CS701 - Parallel Programming Assignment - 2

M5 - Message Passing Interface (MPI)

Contents

1 Hello World Example 1
Q1. Hello World Program - Version 1 2
Q2. DAXPY Loop 2
Q3. Hello World Program - Version 2 2
Q4. Hello World Program - Version 3 3
Q5. Calculation of π - `MPI_Bcast` and `MPI_Reduce` 3
Q6. Ocean Kernel 4
Q7. Reduction operation 4
Q8. Collective Communication - Scatter - Gather 6
Q9. MPI Derived Datatypes 7
Q10.Pack and Unpack 9
Q11.Derived Datatype - Indexed 9
Q12.Matrix Dot Product 10
Q13.MIMD Example 10
Q14.Matrix Multiplication on a Cartesian Grid (2D Mesh) using Cannon's Algorithm 13
Q15.Martix Multiplication using Cannon's Algorithm for Large Matrices 16

Note: Deadline: 8pm, October 15. Pack your report, code, screenshots and other files in an archive and mail to `cs701.nitk@gmail.com`. This is a team assignment. At most two students per team. One submission per team.

1 Hello World Example

MPI_Init(int *argc, char **argv); Initializes the MPI execution environment. Should be the first MPI call. After this call the system is setup to use the MPI library.

MPI_Comm_rank(MPI_Comm comm, int rank); Determines the rank of the calling process in the communicator. The first argument to the call is a communicator and the rank of the process is returned in the second argument. Essentially a communicator is a collection of processes that can send messages to each other. The only communicator needed for basic programs is `MPI_COMM_WORLD` and is predefined in MPI and consists of the processees running when program execution begins.

MPI_Finalize(); Terminates MPI execution environment

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

//Include user libraries

//Defines
//Global variables

int main (int argc, char *argv[]) {

//Declare int variables for Process number, number of processes and length of processor name.
int  rank, size, namelen;

//Declare char variable for name of processor
char name[100];

//Intialize MPI
MPI_Init(&argc, &argv);

//Get number of processes
MPI_Comm_size(MPI_COMM_WORLD, &size);
//Get processes  number
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

//Get processesor name
MPI_Get_processor_name(name, &namelen);

printf ("Hello World. Rank %d out of %d running on %s!\n", rank, size, name);

//Terminate MPI environment
MPI_Finalize();
return 0;
}
```

Installation and Compilation

You will need the MPI compiler, (`mpicc`), and the MPI runtime, (`mpirun`) to run MPI programs. On most Ubuntu systems running the following commands will install `mpicc` and `mpirun`. MPI Manual pages are installed by the `openmpi` package. A small installation guide is here[1]. A more detailed installation is here[2]. The MPI complete reference is here[3].

```
$ sudo apt-get update
$ sudo apt-get install openmpi-bin openmpi-common openmpi-doc
```

Compile and run the Hello World progam.

```
$ mpicc -o helloworld ./helloworld.c
$ mpirun -n 4 ./helloworld
```

Q1. Hello World Program - Version 1

Hello World Program - Version 1

Initialize the MPI parallel environment. Each process should identify itself and print out a Hello world message.

Q2. DAXPY Loop

DAXPY Loop

D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size 2^{16} each, P stands for Plus. The operation to be completed in one iteration is X[i] = a*X[i] + Y[i]. Implement an MPI program to complete the DAXPY operation. Measure the speedup of the MPI implementation compared to a uniprocessor implementation. The `double MPI_Wtime()` function is the equivalent of the `double omp_get_wtime()`.

Q3. Hello World Program - Version 2

Hello World Program - Version 2

Initialize the MPI parallel environment. Process with rank $k(k = 1, 2, \ldots, p-1)$ will send a "Hello World" message to the master process (has Rank 0). The master process receives the message and prints it. An illustration is provided in Figure 1. Use the MPI point-to-point blocking communication library calls (`MPI_Send` and `MPI_Recv`). Example usages of the calls follow:

- `MPI_Send(Message, BUFFER_SIZE, MPI_CHAR, Destination, Destination_tag, MPI_COMM_WORLD);`: Send the string of `MPI_CHAR` of size `BUFFER_SIZE` to `Message` to Process with Rank `Destination` in `MPI_COMM_WORLD`;
- Example: `MPI_Recv(Message, BUFFER_SIZE, MPI_CHAR, Source, Source_tag, MPI_COMM_WORLD, &status);`: Receive the string `Message` of `MPI_CHAR` of size `BUFFER_SIZE` from Source belonging to `MPI_COMM_WORLD`. Execution status of the function is stored in `status`.

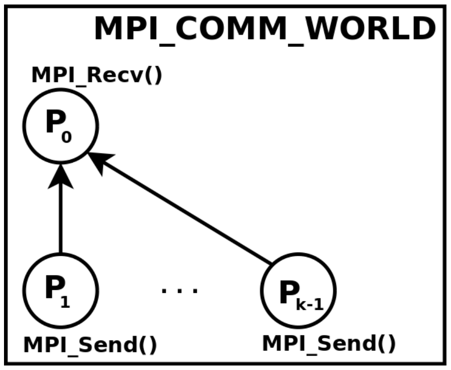


Figure 1: Illustration of the Hello World - Version 2 processes communicating with the root process.

Q4. Hello World Program - Version 3

Hello World Program - Version 3

Create 8 MPI processs. Each process generates a random number. process 0 (the master process) passes its random value to process 1. process 1 prints out a hello world message along with the received random number. process 1 sends the number to process 2 and so on. Each process prints a hello world message along with the number it receives. Note: the message has to printed only after the number is received.

Q5. Calculation of π - `MPI_Bcast` and `MPI_Reduce`

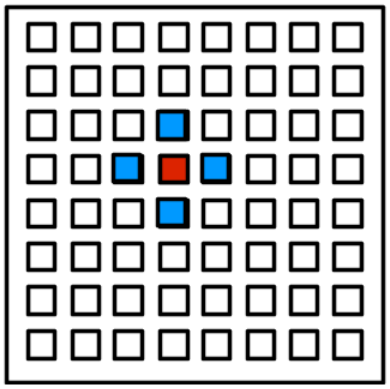
Calculation of π - `MPI_Bcast` and `MPI_Reduce`

This is the first example where many processs will cooperate to calculate a computational value. The task of this program will be arrive at an approximate value of π. The serial version of the code follows.

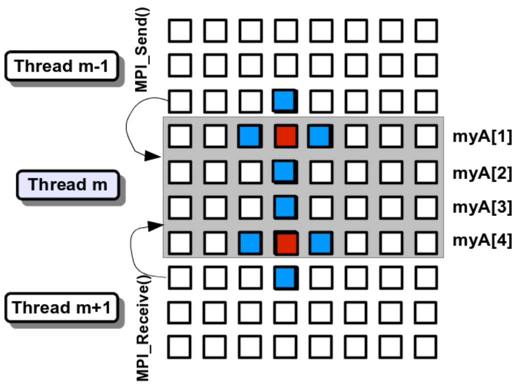
```
static long num_steps = 100000;
double step;
void main ()
{
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double)num_steps;
  for (i=0; i<num_steps; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Your task is to create a parallel MPI version of the π program. For this version, use the `MPI_Bcast` and `MPI_Reduce` functions.

- Broadcast the total steps value (`num_steps`) to all the processs (process 0 could broadcast). The syntax and usage of `MPI_Bcast` follow:
  - `MPI_Bcast(void *message, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`: Broadcast a message from the process with rank "root" to all other processes of the group.
  - Example: `MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);`. process 0 broadcasts `n`, of type `MPI_INT`, to all the processs MPI_COMM_WORLD.
- Each process calculates its partial π value. Partial $\pi = \frac{4.0}{1.0+x^2} \times$ `step`).
- Use `MPI_Reduce` to reduce the partial π values generated by each process. `MPI_Reduce` is executed by every process. Syntax and an example follow:



(a) Array layout showing the elements involved in the averaging step.



(b) Array layout showing some of the communication required by the processes in the Ocean kernel.

Figure 2: Illustrations for the Ocean kernel question (Q6.).

- MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Operator op, int root, MPI_Comm comm);: Reduce values on all processes to a single value. Count refers to the number of operand and result values.
- Example: MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);: Perform MPI_SUM on mypi from each process into a single pi value and store it in process 0.

Q6. Ocean Kernel

Ocean Kernel

The task for this question is to parallelize the ocean kernel.

The kernel works on a large array of floating numbers. For every element, the average of the element with its neighbors is calculated. The difference between the average and the original element is recorded. The sum of differences over the entire matrix is calculated. The kernel ends if the sum is below a threshold. The ocean kernel pseudocode follows:

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i <-- 1 to n do
         for j <-- 1 to n do
            temp = A[i,j];
            A[i,j] <-- 0.2 * (A[i,j] + neighbors);
            diff += abs(A[i,j] - temp);
         end for
      end for
      if (diff < TOL) then done = 1;
  end while
end procedure
```

Implement the ocean kernel in MPI. MPI_Send and MPI_Recv will be useful. Note: While you are free to choose any strategy to distribute work among processes, one way would be to divide the rows of the array equally amongst the processes. Depending on the strategy used MPI_Barrier may also be useful.

Q7. Reduction operation

Reduction operation

The task for this question is to write two versions of an MPI program to find sum of n integers on n processors without using the MPI_Reduce API. The illustration of the last 4 iterations of the reduction operation are shown in Figure 3. The pseudocode for the entire operation is also presented. For more explanation you may visit Page 521 of the PH-CoD-2014 book.

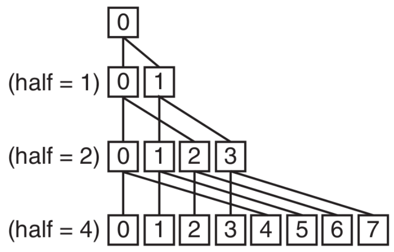


Figure 3: The last four levels of a reduction that sums results from each processor, from bottom to top. For all processors whose number i is less than half, add the sum produced by processor number (`i + half`) to its sum (Figure 6.8 from Patterson and Hennessy, Computer Organization and Design. MK. 2014.).

The basic idea is to bisect the available processors into two sets, say (P_0, ..., $P_{\frac{n}{2}-1}$) and ($P_{\frac{n}{2}}$, ..., P_{n-1}). One half sends its number to its peer in the other half - Process $P_{i+\frac{n}{2}}$ sends its integer to P_i. The receiving process computes the partial sum – process P_i calculates ($P_i + P_{i+\frac{n}{2}}$). This process is repeated till the last process (P_0 in the pseudocode) remains with the accumulated sum. Point to note - what if there are an odd number of processors at the beginning of this step? (the pseudocode handles this situation too).

```
half = numprocs;
do
  synch(); /*wait for partial sum completion*/

  /* Conditional sum needed when half is odd;
     Processor0 gets missing element. */
  if (half%2 != 0 && Pn == 0)
     sum[0] += sum[half-1];

  half = half/2; /*dividing line on who sums */

  if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

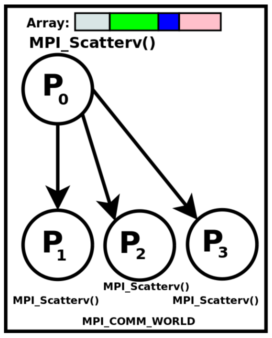
Implement two versions of this program.

Ver. 1. Use blocking point-to-point communication API - `MPI_Send`, `MPI_Recv`.

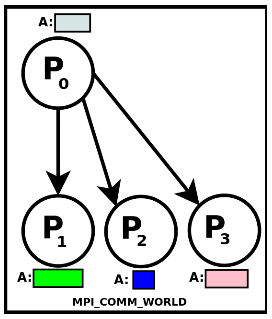
Ver. 2. Use non-blocking point-to-point communication API - `MPI_ISend`, `MPI_IRecv`. `MPI_Wait` or `MPI_Waitall` is useful in this scenario.

- `MPI_ISend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`: Begins a non-blocking send.
- `MPI_IRecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Reque *request);`. Both `ISend` and `Irecv` return a request parameter that can be used in the `Wait` and `Waitall` functions.
- `MPI_Wait(MPI_Request *request, MPI_Status *status);`: Waits for a non-blocking MPI send or receive to complete. The request parameter corresponds to the request parameter returned by `MPI_Isend` or `MPI_Irecv`.
- `MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`: Waits for all given communications to complete and to test all or any of the collection of nonblocking operations. Example usage:

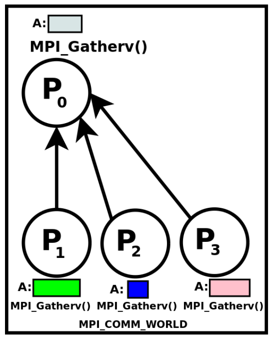
```
...
MPI_Status  *status;
MPI_Request request[200];
...

/* Receiver - Non Blocking calls */
MPI_Irecv(&value_to_recv, 1, MPI_INT, Source, Source_tag, MPI_COMM_WORLD, &request[i]);
```



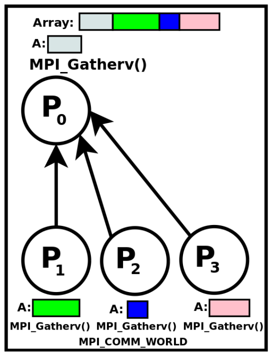
(a) Root process decides non-uniform segments of `Array` to be sent to processes P0, P1, P2, and P3.



(b) All processes execute `MPI_Scatterv()`, receive segments of `Array`.



(c) Each process the computation step. Each process executes `MPI_Gatherv()` to send its segment to the Root process.



(d) Root process receives segments of the processed array (including its own).

Figure 4: The Scatter-Gather sequence for Q8..

```
{\tt MPI\_Waitall(1,&request[i],status);}
...

/* Sender - Non Blocking calls */
MPI_Isend(&value_to_send, 1, MPI_INT, Destination, Destination_tag, MPI_COMM_WORLD,&request[i]);

/* wait for the Isend corresponding to request[i] to complete */
/* alternatively MPI_Wait could have been used here */
MPI_Waitall(1,&request[i],status);
...
```

Q8. Collective Communication - Scatter - Gather

Collective Communication - Scatter - Gather

The task for this question follows:

- The root process (Rank 0) distributes an array in a non-uniform manner to all the processes in the communicator (including itself). This step is called *Scatter*.
- Each process (including the root) finds the square root of each element it receives.
- After the computation, all the processes return the modified array back to the root process. This step is called *Gather*.

The sequence of operations are illustrated in Figure 4.

MPI provides `MPI_Scatter`, `MPI_Scatterv`, `MPI_Gather`, and the `MPI_Gatherv` primitives for use in situations where scatter-gather sequences appear.

- `MPI_Scatterv`: Scatters a buffer in parts to all tasks in a group.

  ```
  int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  ```

  - The root process scatters the array `sendbuf`.
  - Size of each segment is recorded in the array `sendcounts`. `sendcounts[0]` is the number of elements to be sent to P0, `sendcounts[1]` is the number of elements to be sent to P1, and so on.
  - `displs`: Integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i. `sendbuf[ displs[0] ]`, `sendbuf[ displs[0]+1 ]`, . . ., `sendbuf[ displs[0]+ sendcounts[0]-1 ]` are sent to P0, `sendbuf[ displs[1] ]`, `sendbuf[ displs[1]+1 ]`, . . ., `sendbuf[ displs[1]+ sendcounts[1]-1 ]` are sent to P1, and so on.
  - Each receiving process is returned its segment in the `recvcount` array.
  - `root`: The sending processes in the communicator (`comm`).

  Note: The `sendbuf`, `sendcounts`, `displs` parameters are meaningful only in the root process.

- `MPI_Gatherv`:Gathers varying amounts of data from all processes to the root process.

  `int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)`

  Each of the parameters in the `MPI_Gatherv` function have similar meaning as in the `MPI_Scatterv` function.

  - Each sending process populates its `sendbuf` with `sendcount` elements of type `MPI_INT`.
  - The root process receives `recvbuf` array which is the union of all the sendbuf arrays. The elements are filled in `recvbuf` in the order of ranks of the subprocesses.

  Note: The `recvbuf`, `recvcounts`, `displs` parameters are meaningful only in the root process.

  Example usage:

  ```
  MPI_Gatherv(My_gatherv, Send_Count, MPI_INT,
              Total_gatherv, recv_count, Displacement, MPI_INT,
              Root, MPI_COMM_WORLD);
  ```

  Each subprocess has its copy of `My_gatherv` which contains `Send_Count` elements. The `Root` process receives `Total_gatherv`. Each subprocess's segments start from the indices recorded int he Displacement array.

- `MPI_Scatter`, `MPI_Gather`: These primitives are used to scatter (gather) arrays uniformly (all segments of the array have equal number of elements) between subprocesses.

Q9. MPI Derived Datatypes

MPI Derived Datatypes

The task to complete:

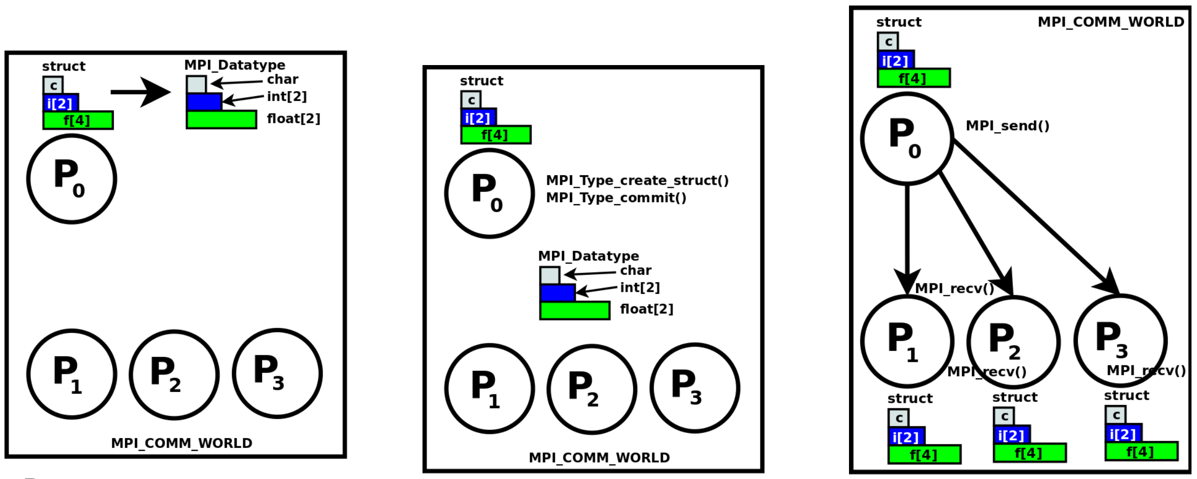
- Build a custom datatype (it is called a *derived datatype*) in the MPI program.
- The example derived datatype for this question is a `struct`. The `struct` contains a char, an int array with 2 elements, and a float array with 4 elements.
- The root process broadcasts one data item of the derived datatype (the `struct` in this case) to all other subprocesses in the communicator.
- The root process sends the data item of the derived datatype to each subprocesses through a point-to-point communication call.

The sequence of steps to be followed to implement this MPI program are shown in Figure Q9.. The steps are described below.

- To create a derived datatype that mirrors a `struct`, use the `MPI_Type_create_struct` function.

  Prototype: `int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displaceme MPI_Datatype array_of_types[], MPI_Datatype *newtype)`

  The following example creates a derived datatype of the structure shown below.



(a) Root process creates a new `MPI_datatype` reflecting the mirror of the `struct`.

(b) After the `MPI_Commit`, all processes register the derived datatype.

(c) Root sends data in the structure to all the subprocesses. The subprocesses now identify the data as an item of the derived datatype.

Figure 5: The sequence of steps in the MPI program for the derived datatype question (Q9.).

```
struct dd{
  char c;
  int i[2];
  float f[4];
};
```

- `count` indicates the number of blocks – 3 for the structure above (char, int, float).
- `array_of_blocklengths`: Number of elements in each block. [1, 2, 4] for the structure above. Indicates that first block contains one element, the second block has 2 elements and the third block has 4 elements. The code snippet:

  ```
  array_of_block_lengths[0] = 1;
  array_of_block_lengths[1] = 2;
  array_of_block_lengths[2] = 4;
  ```

- `array_of_displacements`: Byte displacement of each block (array of integers). The first value is always 0. The second displacement value is the difference between the addresses of the second block and the first. The third displacement value is the difference between the addresses of the third block and the first. The `array_of_displacements` is of type `MPI_Aint`. Use the `MPI_Get_address` function to retrieve the addresses of type `MPI_Aint`. The following snippet shows the second displacement calculation.

  ```
  MPI_Get_address(&s.c, &block1_address);
  MPI_Get_address(s.i, &block2_address);
  array_of_displacements[1] = block2_address-block1_address;
  ```

  `block1_address` and `block2_address` are of type `MPI_Aint`. `array_of_displacements` is an array of size total blocks and is of type `MPI_Aint`. `s` is of type `struct dd`(shown above).

  Prototype of `MPI_Get_address`: `int MPI_Get_address(void *location, MPI_Aint *address);`
- `array_of_types`: Type of elements in each block. For the structure above, the first block contains characters, the second is integer and the third block contains floating point numbers. The `array_of_types` is of type `MPI_Datatype`. The values will be `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`.
- `MPI_Type_create_struct` returns handle to the derived type in `newtype`.

- Commit the `newtype` using the `MPI_Type_commit` call. Prototype:

  `int MPI_Type_commit(MPI_Datatype *datatype)`

- *Collective communication*: The root broadcasts the filled structure to all the processes in its communicator. Print out the structure in all the processes.

- *Point-to-point communication*: The root sends the filled structure to each process in its communicator. The receiving process prints out the structure.

Q10. Pack and Unpack

Pack and Unpack

The previous question illustrated the creation of a derived datatype to help communicate compound datastructures. The same objective can be achieved by *packing* elements of different types at the sender process and *unpacking* the elements at the receiver process. For this question, achieve the same effect as the previous question using `MPI_Pack` and `MPI_Unpack` routines. Prototypes follow:

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
    void *outbuf, int outsize, int *position, MPI_Comm comm)
```

`incount` number of items from `inbuf`, each of type `datatype` are stored contiguously in `outbuf`. The input/output parameter `position` mentions the offset to start writing into (0 for the first pack). On return, the next write can begin from `position`. `outsize` is the size of `outbuf`. The following code shows packing a char, an int array and a float array in the sender process.

```
MPI_Pack(&c, 1, MPI_CHAR, buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(iA, 2, MPI_INT,  buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(fA, 4, MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
```

To send (receive) the packed `buffer` use the datatype `MPI_PACKED` in `MPI_Send`(`MPI_Recv`). The prototype for `MPI_Unpack`, to be used by the receiver process, follows.

```
int MPI_Unpack(void *inbuf, int insize, int *position,
     void *outbuf, int outcount, MPI_Datatype datatype,
     MPI_Comm comm)
```

`inbuf` is the packed buffer of size `insize`. Unpacking starts from `position` (semantics of `position` are the same as `MPI_Pack`. `outcount` number of elements of type `datatype` are read from `inbuf` and placed in `outbuf`.

Q11. Derived Datatype - Indexed

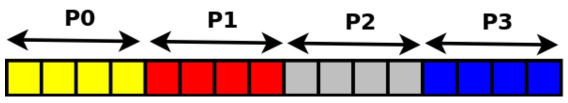
Derived Datatype - Indexed

The objective of this question is to create an indexed derived datatype. The task of this question is to create an indexed derived datatype that stores the upper triangle of a 2 dimensional matrix and sends it to another process. Use the `MPI_Type_indexed` call to create the "upper_triangle" derived type. Prototype:

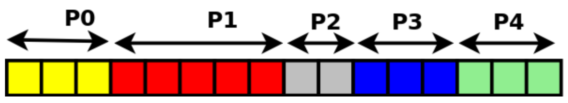
```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
     int *array_of_displacements, MPI_Datatype oldtype,
     MPI_Datatype *newtype)
```

The `MPI_Type_indexed` creates an indexed derived datatype with the handle `newtype`. `newtype` contains `count` number of blocks. Number of items of type `oldtype` in each block is stored in `array_of_blocklengths`. `array_of_displacements` stores the displacement for each block, in multiples of oldtype extent. An example is shown below.

Consider A = $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$, populated by the Root process. An indexed datatype has to be created that stores the elements in the upper triangular matrix. The upper triangular matrix A_upper = $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 10 & 11 \\ 0 & 0 & 0 & 15 \end{bmatrix}$ has to be indexed and sent to process with rank 1. In each row, the upper triangular matrix elements start from the $(N \times row) + row$



(a) Equal distribution of array elements to 5 processes.



(b) Unequal distribution of array elements to 5 processes.

Figure 6: Distribution of array elements in the two versions of the program (Q12.).

position in a matrix of size N×N. For this example, `array_of_displacements` = [0,5,10,15]. There are as many blocks as there are a rows in the matrix (`count` = N. A block from a row contains (N - row) elements (where row is the row index of the matrix) `array_of_blocklengths`=[4,3,2,1]. `oldtype` is `MPI_INT`. The call could look like this:

```
MPI_Type_indexed(N, blocklen, displacement, MPI_INT, &upper_triangle);
```

The task of this program is to create this `MPI_Type_indexed` derived datatype, commit it, and send an upper triangular matrix to the Rank 1 process. Note that blocks in `MPI_Type_indexed` could be from any arbitrary row in any sequence. The `MPI_Type_indexed` facilitates sending an irregular arrangement of elements from multi-dimensional matrices. To communicate a regular section of the matrix `MPI_Type_vector` is used. An example wouldbe to distribute columns of an N×N array to N processes.

Q12. Matrix Dot Product

Matrix Dot Product

The task for this question is calculate the dot product of two arrays and to perform reduction on the product array. The working follows:

- The root process populates two arrays, each of size N.
- The arrays are distributed amongst P processes in the communicator.
- Each process calculates the dot product and the reduction operation.
- Root performs the final reduction operation to obtain the final answer.

The values for N and P are your choice. The arrays can be filled with random numbers. Implement two versions of this code (Figure 6).

1. Each process gets an equal sized chunk of both the arrays. (using `MPI_Scatter`).
2. Each process gets an unequal sized chunk of both the arrays. (using `MPI_Scatterv`).

Q13. MIMD Example

MIMD Example

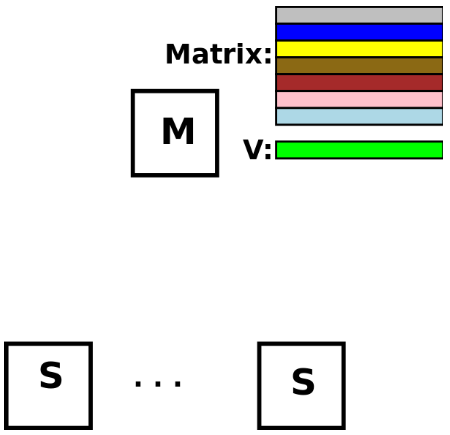
The objective in brief:

- Create 2 instruction streams - a master and a slave. Master has a large amount of data to be processed (Eg. 1000 x 1000 matrix).
- Master distributes a small set of data to each slave. (Eg. A row from the matrix to each slave)
- Each slave process computes a value and returns the result to the master.
- On receiving the computed value for a row, the master distributes the next row to the slave.
- After all the data is exhausted, Master notifies each slave.

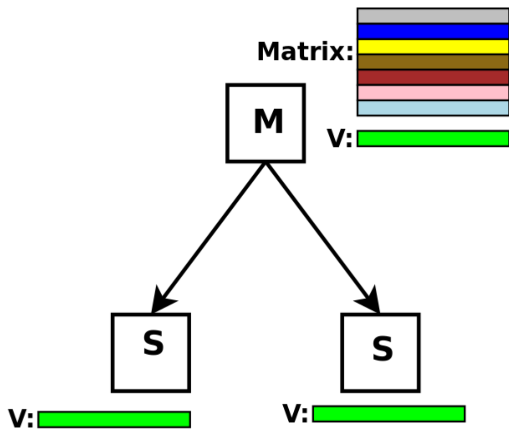
Master-slave parallel algorithms where the master acts as a manager and schedules tasks to the slave processes is also called the *self-scheduling* parallel algorihtm.

The programming tasks are detailed below (Figure 7). Assume: Root process (Rank 0) is the master and slave processes have Rank > 1.

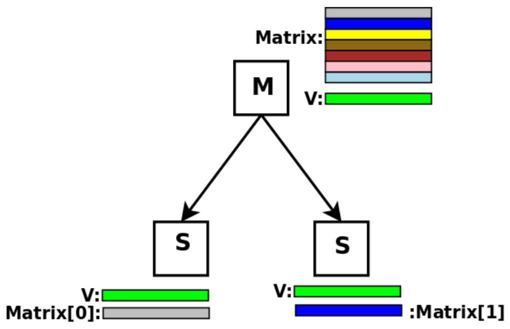
- Create 2 MPI programs - master and slave. Create two arrays in the master.



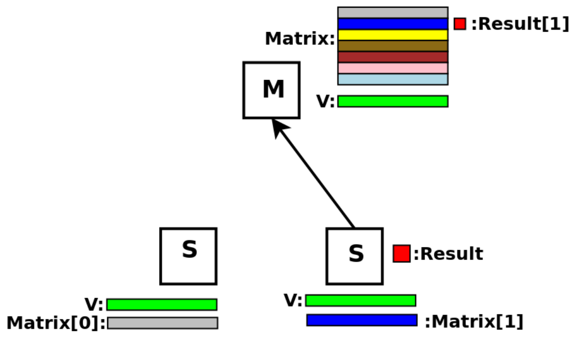
(a) The master process and slave processes. The master process populates a matrix and a vector.



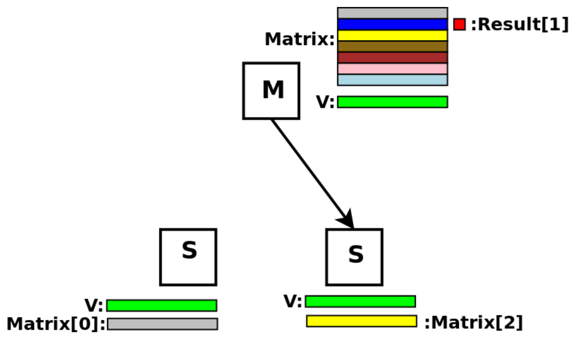
(b) Each slave receives a copy of the vector.



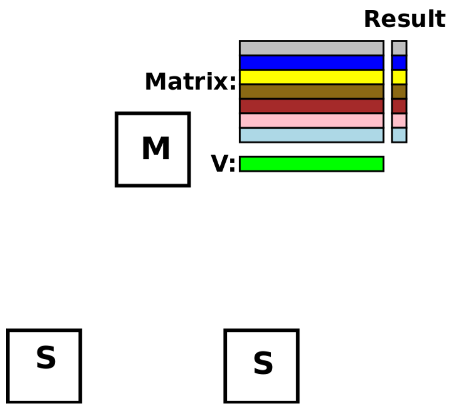
(c) Each slave is assigned a row from the matrix.



(d) The elements of the row are processed and returned to the master. The result is recorded in a result vector. The result is stored in the position corresponding to the original row position.



(e) The next row from the matrix is assigned to the slave that returned the most recent result. Every row of the matrix is processed in this manner.



(f) The result vector is available with the master process. The slave processes can safely exit after this point.

Figure 7: The sequence of events for Q13..

1. A vector of N floating point numbers. (N = 1000, for example)
2. A matrix of N × N floating point numbers. (both the vector and matrix have N elements in each dimension - this is required for the computation)

- Master: Broadcast the vector to each slave.
- Master: For each slave, send 1 row from the matrix. Eg. Rank 1 gets row 0, Rank 2 gets row 1, and so on.
- Slave: Each slave now has 2 vectors of length N - One from the first vector, and another is the row from the matrix. Each slave process, after receiving the vector, enters a loop in which it receives row from the matrix, processes the row, and sends back the result to the master. Task for the slave is to elementwise multiply both vectors and reduce the product matrix. Simply this: `Sum += V[i] * R[i];` where V is the vector, R is the received row.
- The slave sends the reduced value `Sum` to the master. The row number to which the reduced sum corresponds to is encoded in the `tag` parameter in `MPI_Send`. Example is shown later.
- Master recieves the processed value and stores it in a result vector under the index equal to the original row number. The master process sends the next row (from the original matrix) to the slave that returned the most recent processed value.
- The sequence continues till all N rows in the matrix have been sent to the slave processes. After all the rows have been exhausted in the master process, all the slaves are yet to return the processed values of the last batch of rows.
- After the slave process finishes processing its last row, it has to be notified by the master. This is done by the master by sending a dummy row and a special tag value. The slave can terminate safely after this point. You may follow any strategy you wish to implement this. An example is shown below.
- At the end of these steps, the master process now has the result vector. The master can print the result vector in a output stream and safely exit.

A few code snippets that might help:

- The master process sends a row from the matrix to each slave. The row number has to be recorded throughout the communication sequence (master → slave → master) so that the returned value can be put in the right place in the final vector. One of the strategies to encode the row number of the matrix in the tag parameter. The master sends the row along with this tag to the slave; the slave sends it back to the master process along with the processed value. Example:

  ```
  MPI_Send(RowBuffer, N, MPI_FLOAT, SlaveRank, RowNumber+1, MPI_COMM_WORLD);
  ```

  `Matrix[RowNumber]` has been stored in `RowBuffer` and is sent to process with `SlaveRank`. The `RowBuffer` contains `N` elements of type `MPI_FLOAT`. The tag value is set to 1 plus the RowNumber in this code (a non-zero value means the slave has received a valid row). The tag value 0 is reserved to indicate that all rows have been exhausted.

  For the above code to work, each slave process should know `N` and must have required amount of memory allocated for RowBuffer (what is the required size?).

- The slave will have to accept messages of any tag value. This can be achieved by setting the Source tag parameter of `MPI_Recv` to `MPI_ANY_TAG`. The slave sends the processed value back to the master using the tag it received.
- The computation in the slave processes can complete in any order. The master process should be able to receive messages from any source. Reception of a message from any source can be achieved by setting the Source parameter in `MPI_Recv` to `MPI_ANY_SOURCE`. Example:

  ```
  MPI_Recv(&ReducedValue, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  ```

  The `status` parameter can be used to retrieve the identity of the source and the exact tag value:

  ```
      SenderRank = status.MPI_SOURCE;
      Tag = status.MPI_TAG;
  ```

- The above code can also be used by the slave process to check if the tag equals 0 (thereby indicating that the matrix processing is complete).

Compile each program individually. To start one master process and 4 slave processes run the following mpirun command (master.x and slave.x are the master and slave executables):

```
$ mpirun -np 1 ./master.x : -np 4 ./slave.x
```

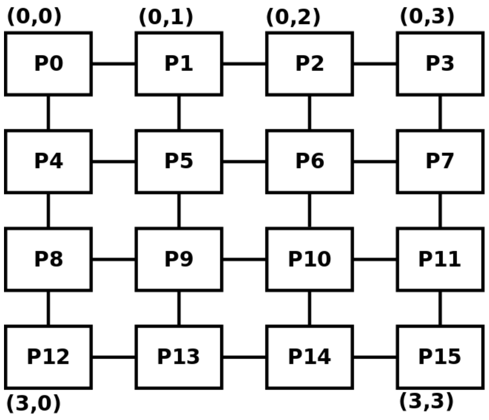


Figure 8: For this question, assume that the processors are organized in the form of a 2D Mesh. The addressess of the processors and the coordinates are also shown.

Q14. Matrix Multiplication on a Cartesian Grid (2D Mesh) using Cannon's Algorithm

Matrix Multiplication on a Cartesian Grid (2D Mesh) using Cannon's Algorithm

For this question, we will assume that the multiprocessor is interconnected in the form of a grid (a 2-dimensional Mesh; Dimensions are X-axis and Y-axis). An example 4×4 Mesh is shown in Figure 8. For this questin, the mesh contains equal processors in both dimensions as shown. The 16 processors shown in the figure can be thought of being arranged in a Cartesian grid - the coordinates of such a grid are also shown in the figure.

Assume that one process is executing on one processor. For the purpose clarity the ranks of the processes equals the identies of the processor it is executing on. The objectives of this question are:

- Create N× N processes. Arrange the processors in the grid fashion.
- The input arrays are distributed equally amongst all processes (scatter operation). The product array calculated using Cannon's multiplication algorithm[4]. The root process gathers the product array and displays it on the output.

Read the details of Cannon's multiplication algorithm here: Parallel Matrix Multiplication page. An example is provided in the Cannon's Matrix Multiplication section (below).

Details of the program implementation follow. The program can be divided into two stages.

Stage 1 Creation of a Grid of processes.

Stage 2 Cannon's Matrix Multiplication.

Creation of a Grid of processes

- Create 16 processes (`-np 16` in `mpirun`).
- Create a new communicator to which a Cartesian 4× 4 grid topology is attached. The API for this in MPI is `MPI_Cart_create`. Prototype:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
     int *periods, int reorder, MPI_Comm *comm_cart)
```

  – `comm_old` is the communicator whose processes are to be arranged in a grid. If all the processes in the MPI program are to be put in the grid, `comm_old` is equal to `MPI_COMM_WORLD` (the global communicator). The grid is now attached to the new communicator pointed to by `comm_cart`.
  – `ndims`: The grid contains `ndims` dimensions. For this program `ndims`=2. The X and Y dimensions.
  – `dims` is an array. Each element records the number of processes per dimension. dims[0] is the number of processes in the X dimension. dims[1] is the number of processes in the Y dimension. Both values are 4 for this question.
  – periods is the logical array of size `ndims` specifying whether the grid is periodic (true) or not (false) in each dimension. Both values are True for this question.

  - `reorder` is a flag that indicates if the process ranks can be reordered in the new communicator. The newly created grid should have the same ordering. This value is false for this question.

- Example:

  ```
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &grid_comm));
  ```

  `grid_comm` is the handle to the new communicator.

- For each process in the grid, the following data will be useful to record.
  - Grid info: Size of the grid, No. of processors per dimension.
  - Rank in the grid.
  - Position in the grid - row and column (X and Y coordinates).
  - All communicator handles to which the process belongs.

  Storing the data in a `struct` similar to the one below is suggested.

  ```
  typedef struct{
    int      N;        /* The number of processors in a row (column). */
    int      size;     /* Number of processors. (Size = N*N           */
    int      row;      /* This processor's row number.                */
    int      col;      /* This processor's column number.             */
    int      MyRank;   /* This processor's unique identifier.         */
    MPI_Comm comm;     /* Communicator for all processors in the grid.*/
    MPI_Comm row_comm; /* All processors in this processor's row   .  */
    MPI_Comm col_comm; /* All processors in this processor's column.  */
  }grid_info;
  ```

- The coordinates of the current process can be obtained using the following API call. These are needed during multiplication.

  ```
    MPI_Cart_coords(grid->comm, grid->MyRank, dims, Coordinates);
  ```

  `Coordinates` is the array that records the position of the process in the grid. `Coordinates[0]` stores the X position and `Coordinates[1]` stores the position in the Y axis.

  Note: You can verify that the grid has been created by printing out the coordinates of each process.

- Create communicators for individual rows and columns. This will ease the implementation of the algorithm (in the skewing, and rotating steps).

Cannon's Matrix Multiplication

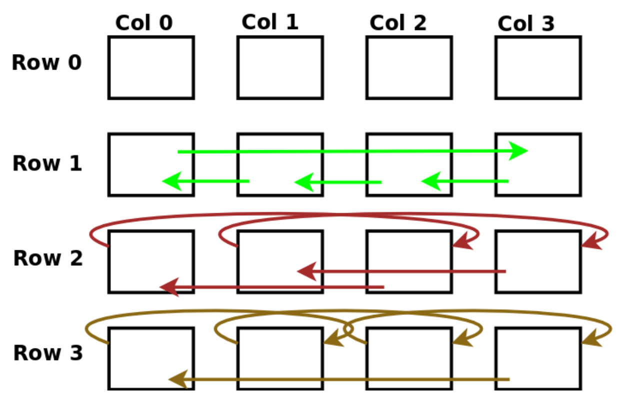
Before explaining the implementation details for the MPI program, an example Multiplication of two 3×3 matrices using Cannon's algorithm is shown.

- Input Matrices are A and B. C is the product matrix.

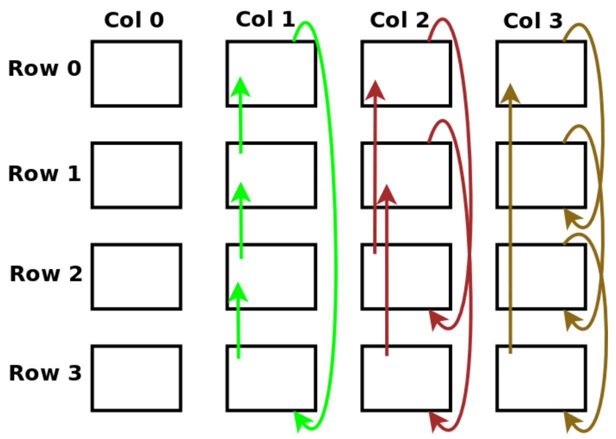
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- In A, left rotate (left shift with wrap around) elements in row i by i positions. In B, north rotate (up shift with wrap around) elements in column i by i positions. Eg. A[2][1] will be shifted to A[2][2]. B[2][1] will be shifted to B[1][1]. This step is called *skewing.* The rowwise and columnwise skewing steps are demonstrated in Figure 9. The skewed matrices are:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 4 \\ 9 & 7 & 8 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 5 & 9 \\ 4 & 8 & 3 \\ 7 & 2 & 6 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



(a) Row-wise Skewing. Each rectangle represents a block of the input matrix.



(b) Column-wise Skewing. Each rectangle represents a block of the input matrix.

Figure 9: Skewing illustrations for Q14. and Q15..

- Multiply elements of A and B in position (elementwise) and add elementwise to C. This operation: C[i][j] += A[i][j]*B[i][j]. The new C matrix is shown.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 4 \\ 9 & 7 & 8 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 5 & 9 \\ 4 & 8 & 3 \\ 7 & 2 & 6 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 10 & 27 \\ 20 & 48 & 12 \\ 63 & 14 & 48 \end{bmatrix}$$

- Left rotate every element in A. North rotate every element in B. Repeat the multiplication step. The status of A and B after the rotate steps are shown in the first line. The status of C after the multiplication is shown next.

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 6 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 8 & 3 \\ 7 & 2 & 6 \\ 1 & 5 & 9 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 10 & 27 \\ 20 & 48 & 12 \\ 63 & 14 & 48 \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 6 & 4 & 5 \\ 7 & 8 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 8 & 3 \\ 7 & 2 & 6 \\ 1 & 5 & 9 \end{bmatrix} \qquad C = \begin{bmatrix} 9 & 34 & 30 \\ 62 & 56 & 42 \\ 70 & 54 & 129 \end{bmatrix}$$

- Repeat the previous step till all the multiplications are done. In this example there are 3 multiplications to complete. The next is the last step. C is the product matrix.

$$A = \begin{bmatrix} 3 & 1 & 2 \\ 4 & 5 & 6 \\ 8 & 9 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 7 & 2 & 6 \\ 1 & 5 & 9 \\ 4 & 8 & 3 \end{bmatrix} \qquad C = \begin{bmatrix} 30 & 36 & 42 \\ 64 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}$$

The core of the algorithm is the element wise multiplication and addition step made possible by skewing and rotating steps. Consider that each element from A and B were stored in a single processor. The skewing and rotating steps correspond to communication of elements between the processors. The multiplication and accumulation step is completed inside the processor. In other words, Process (i,j) in the grid, calculates the element C[i][j]. This idea can be scaled for larger arrays - divide the large arrays into submatrices (blocks) such that each processor now contains a submatrix. Every processor now computes the submatrix of the product instead of a single element.

The details of the implementation of this stage are presented below. The matrices are populated and multiplication algorithm is completed in this stage.

- In the root process, populate the multiplicand and multiplier arrays (A and B) with random numbers. Assume 4×4 arrays for now.
- Divide the array into equal sized blocks and scatter to each process in the grid. For this example we have a convenient 4×4 array and a 4×4 grid. Each process will get one element each of arrays A and B corresponding to its position in the grid. (Process (2,1) in the grid will get A[2][1] and B[2][1]), etc.

- Create communicators containing processes from a each row and each column. Use the `MPI_Cart_sub` call for this purpose. The call partitions a communicator into subgroups. A subgroup in a grid is a lower-dimensional cartesian subgrid. In the case of a 2-dimensional grid the subgroups are groups of process from the same row or the same column. Prototype and example:

  ```
  int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *comm_new)
  ```

  The ith entry of `remain_dims` specifies whether the ith dimension is kept in the subgrid (true) or is dropped (false). `comm_new` is the handle of the new subgrid that contains the calling process. Example below shows the call that returns the communicator handle (`row_comm`) to which all the processes in the same row as the calling process are attached. This is stored in the `struct` (`grid`) defined earlier.

  ```
     remain_dims[0] = 1;
     remain_dims[1] = 0;
     MPI_Cart_sub(grid->Comm, remain_dims, &(grid->row_comm));
  ```

- Skew the input arrays A and B. Skewing involves left rotating each element in the ith row by i positions in Matrix A. Skewing north rotates each element in the ith column by i positions in Matrix B. One way to implement would be to use `MPI_Send` and `MPI_Recv` calls. If you choose to do this, extra care should be taken to avoid deadlocks. The easier and better way of doing this would be to use the `MPI_Sendrecv_replace` call. The call is used to send data in a buffer to a sender process and receive data into the same buffer from another process. Prototype and example:

  ```
   int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
      int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
      MPI_Status *status)
  ```

  The array `buf` containing `count` items of type `datatype`. The contents of this buffer are sent to the process `dest` and is tagged with `sendtag`. The same buffer, `buf`, is filled with a max of `count` elements from the process `source` tagged with `recvtag`. An example:

  ```
    MPI_Sendrecv_replace(item, 1, MPI_FLOAT, destination, send_tag,
                  source, recv_tag, grid.row_comm, &status);
  ```

  The processes in a single row of the grid are attached to the `row_comm` communicator. The current process sends 1 item of type `MPI_FLOAT` to destination and receives 1 `MPI_FLOAT` item from destination.

- Perform the Cannon's Multiplication algorithm. The rotation steps can be implemented in the same manner as the skewing step.

- At the end of the multiplication, gather the product matrix in the root process. Root prints out the result matrix.

Q15. Martix Multiplication using Cannon's Algorithm for Large Matrices

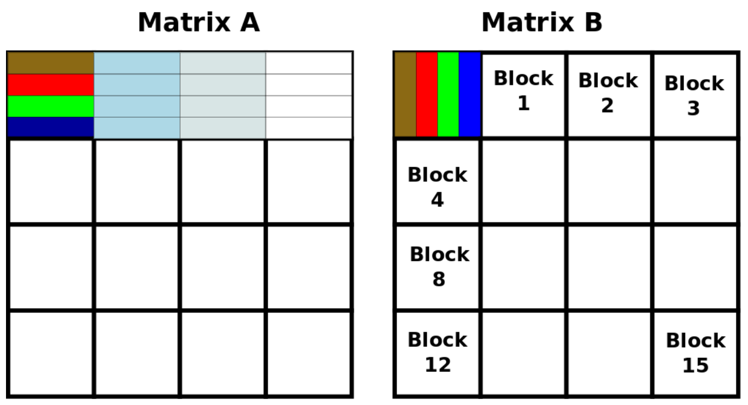
Martix Multiplication using Cannon's Algorithm for Large Matrices

Extend the solution from the previous question for matrices larger than the grid. In this version, each process contains a subblock of the input matrices (instead of a single element). Assume input matrices of size larger than 1024 × 1024. Assume a grid of 16 × 16 processes. A few differences with the previous question are:

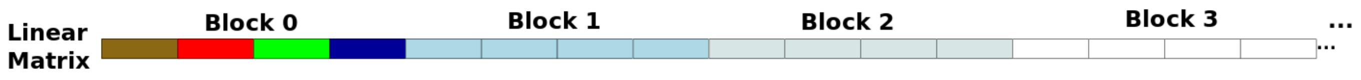
- Each process now gets a submatrix. The root process has to scatter the submatrix to each process. The 2-dimensional matrices have to linearized before using `MPI_Scatter`. The linear submatrices can be scattered to each process. It will be useful to linearize the first matrix's submatrices rowwise and the second matrix's submatrices columnwise.

- The subprocesses do the elementwise multiplication of these linear submatrix arrays. The result will be a row-wise linear version of the product submatrix with each process. The gather operation will put together the linear version of the product array. This has to be converted back to the 2D array.

Figure 10 shows the linearization steps involved in this question.



(a) Blocks in input matrices A and B in the root process. To scatter the blocks to all the processes in the communicator, the blocks in Matrix A is linearized row-wise and the blocks in Matrix B are linearized columnwise.



(b) Linear version of the 2-D matrices. Blocks of Matrix A are linearized row-wise and concatenated. Blocks of Matrix B are linearized column-wise and concatenated. Linear Matrix A is shown here.

Figure 10: Linearizing input matrices for Q15..

Epilogue

This document borrows heavily from the excellent *hyPACK 2013* workshop by CDAC, Pune. The MPI Quick Reference Guide[3] will be useful in implementing the questions in this assignment. The manual pages for MPI are an excellent source of syntax and semantics of the API calls. The recommended MPI Tutorial website[5]. An excellent collention of online books and tutorials[6]. MPI Wikibooks page[7]. Beginning MPI page is a good source[8]. The list of recommended books on MPI is maintained by the MPITutorial website[9]. Other references[10],.

References

[1] A. Dashin, “MPI Installation Guide,” https://jetcracker.wordpress.com/2012/03/01/how-to-install-mpi-in-ubuntu/, Jetcracker.

[2] D. G. Martínez and S. R. Lumley, “Installation of MPI - Parallel and Distributed Programming,” http://lsi.ugr.es/~jmantas/pdp/ayuda/datos/instalaciones/Install_OpenMPI_en.pdf.

[3] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, “MPI Quick Reference Guide,” http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/SPRING-2006/mpi-quick-ref.pdf, netlib.org.

[4] J. Demmel, “Parallel Matrix Multiplication,” http://www.cs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html, CS267.

[5] “MPI Tutorial Webpage,” http://mpitutorial.com.

[6] S. Baden, “MPI Online books and Tutorials Page,” https://cseweb.ucsd.edu/~baden/Doc/mpi.html.

[7] “Message Passing Interface,” https://en.wikibooks.org/wiki/Message-Passing_Interface.

[8] “Beginning MPI,” http://chryswoods.com/beginning_mpi/.

[9] “Recommended MPI Books,” http://mpitutorial.com/recommended-books/.

[10] B. Barney, “MPI,” https://computing.llnl.gov/tutorials/mpi, Lawrence Livermore National Laboratory.