REPORT: HOMEWORK 2
One-Sided MPI Communication

German Research School for Simulation Sciences GmbH Laboratory for Parallel Programming

Rohan Krishna Balaji.- 403596
rohan.balaji@rwth-aachen.de

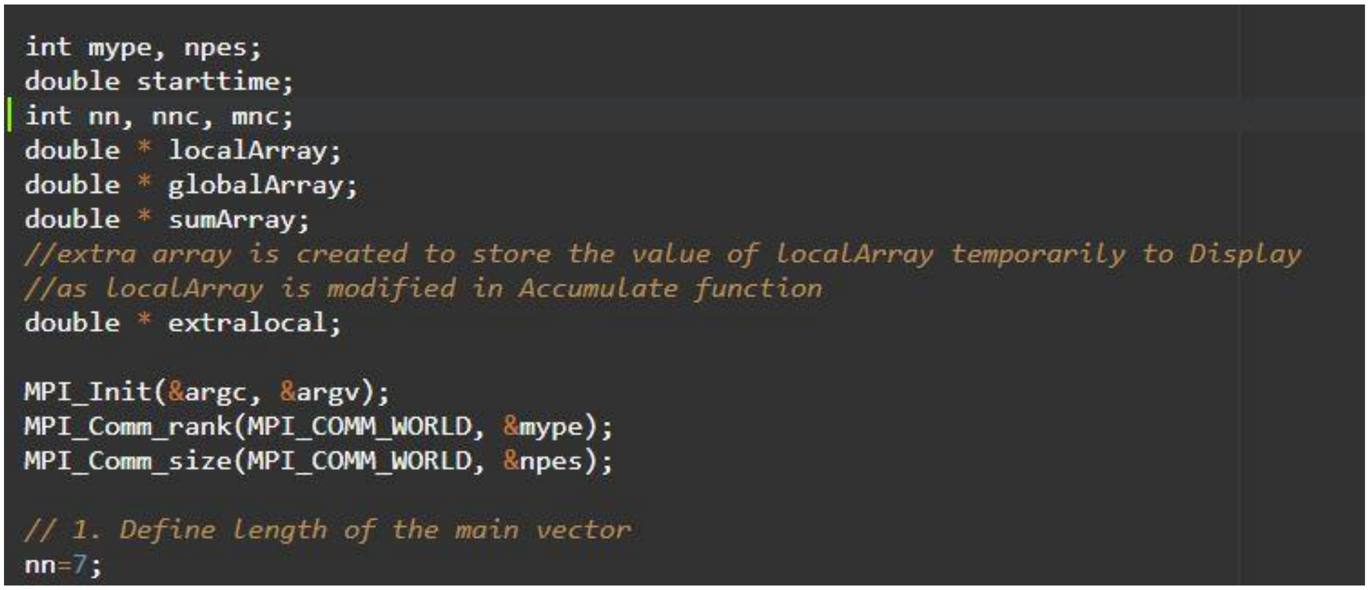
1 INTRODUCTION:

One sided **MPI** (Message Passing Interface) operations are **RMA** (Remote Memory Access) operations, in which there are two main components. **Origin** which is the process that originates the transfer and the **Target** whose memory is being accessed **[1].** Usually in two sided communication there is both send and receive routines. But in RMA operations the process can access the data in a user declared memory area on the target called **Window** which are accessed by all processes in the communicator.

The objective of the project as given in the assignment document **[2],** is to implement the one sided MPI communication. Firstly, the given array is divided and local array are spread across different processes. These local arrays will be initialized with random values, in the first function the values/entries from local arrays of individual processes will be copied to individual global arrays. In the next step, the values in the local arrays are added up in the manner of reduction using accumulate routine. Finally, the sums of these arrays are printed. Detailed working for all these routines are explained in the further sections.

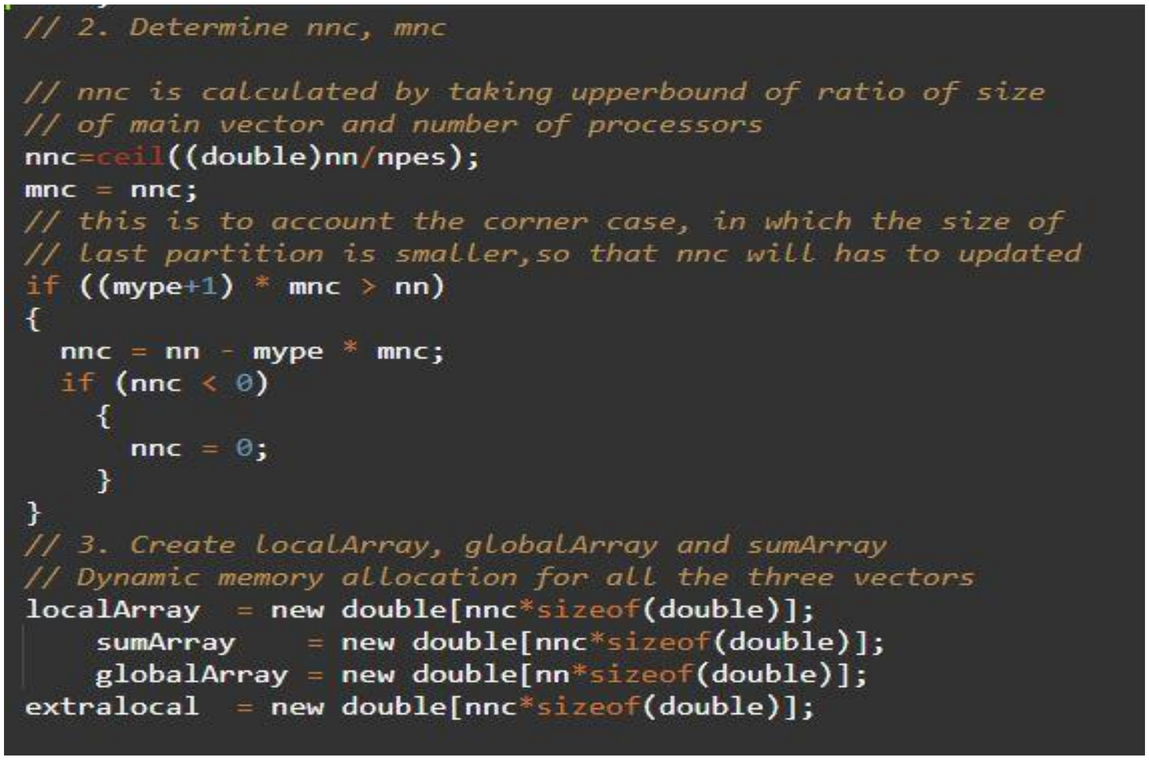
2 METHODOLOGY

The given main vector is split into parts across the processes and are named as localArrays in which these separated values are stored,



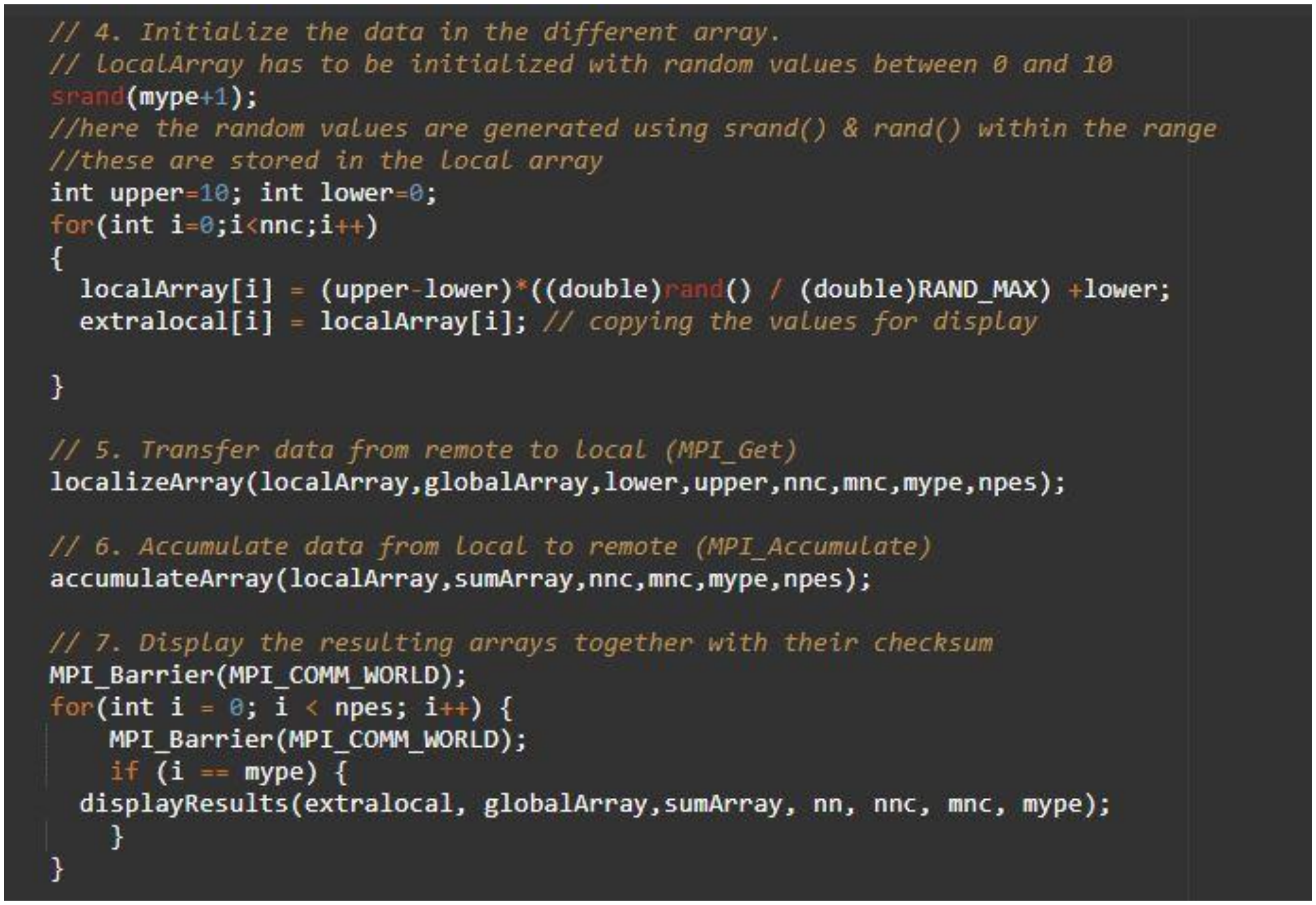
```
int mype, npes;
double starttime;
int nn, nnc, mnc;
double * localArray;
double * globalArray;
double * sumArray;
//extra array is created to store the value of localArray temporarily to Display
//as localArray is modified in Accumulate function
double * extralocal;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mype);
MPI_Comm_size(MPI_COMM_WORLD, &npes);

// 1. Define length of the main vector
nn=7;
```

The basic idea is that the array is divided by number of processes to and each of these will have its own local values, these will be used in next part to implement get and accumulation routines. In the next part the arrays are dynamically allocated.



```
// 2. Determine nnc, mnc

// nnc is calculated by taking upperbound of ratio of size
// of main vector and number of processors
nnc=ceil((double)nn/npes);
mnc = nnc;
// this is to account the corner case, in which the size of
// last partition is smaller,so that nnc will has to updated
if ((mype+1) * mnc > nn)
{
  nnc = nn - mype * mnc;
  if (nnc < 0)
    {
      nnc = 0;
    }
}
// 3. Create localArray, globalArray and sumArray
// Dynamic memory allocation for all the three vectors
localArray  = new double[nnc*sizeof(double)];
    sumArray    = new double[nnc*sizeof(double)];
    globalArray = new double[nn*sizeof(double)];
extralocal  = new double[nnc*sizeof(double)];
```

Code Snippet 1.1 and 1.2: Vector splitting and arrays creation in main function.

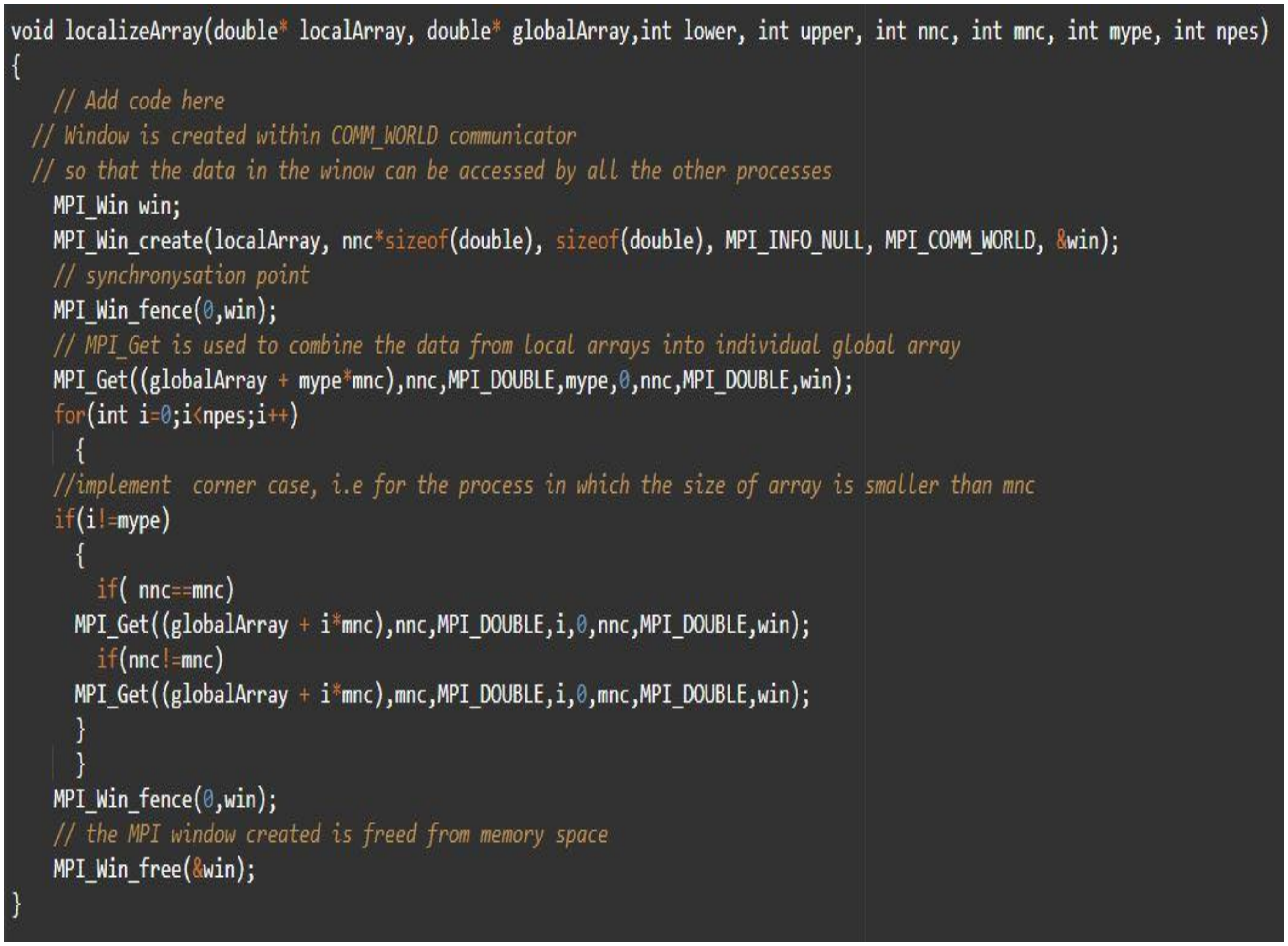


```
// 4. Initialize the data in the different array.
// localArray has to be initialized with random values between 0 and 10
srand(mype+1);
//here the random values are generated using srand() & rand() within the range
//these are stored in the local array
int upper=10; int lower=0;
for(int i=0;i<nnc;i++)
{
  localArray[i] = (upper-lower)*((double)rand() / (double)RAND_MAX) +lower;
  extralocal[i] = localArray[i]; // copying the values for display

}

// 5. Transfer data from remote to local (MPI_Get)
localizeArray(localArray,globalArray,lower,upper,nnc,mnc,mype,npes);

// 6. Accumulate data from local to remote (MPI_Accumulate)
accumulateArray(localArray,sumArray,nnc,mnc,mype,npes);

// 7. Display the resulting arrays together with their checksum
MPI_Barrier(MPI_COMM_WORLD);
for(int i = 0; i < npes; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (i == mype) {
  displayResults(extralocal, globalArray,sumArray, nn, nnc, mnc, mype);
    }
}
```

Code Snippet 2: Local Array initialization and Function calls

In this part of the code, the local array is initialized with random values and these are individual for each process, it is important to note that after MPI initialization the entire kernel is run individually as process on each processor with its own variables and address space. Thus these arrays are used in localizeArray and accumulateArray functions to combine and accumulate the data respectively.
Note that here I have used a copy of local array with the name of extralocal, the purpose of that is, in the accumulateArray function, the local array is modified, but while displaying we need localArray, so I just copied it and displayed. If localArray is need in further again it can be copied back.

In the end before the display of results is made, the **MPI_Barrier** function is called, it Blocks until all processes in the communicator have reached this routine. It blocks the caller until all processes in the communicator have called it; that is, the call returns at any process only after all members of the communicator have entered the call.



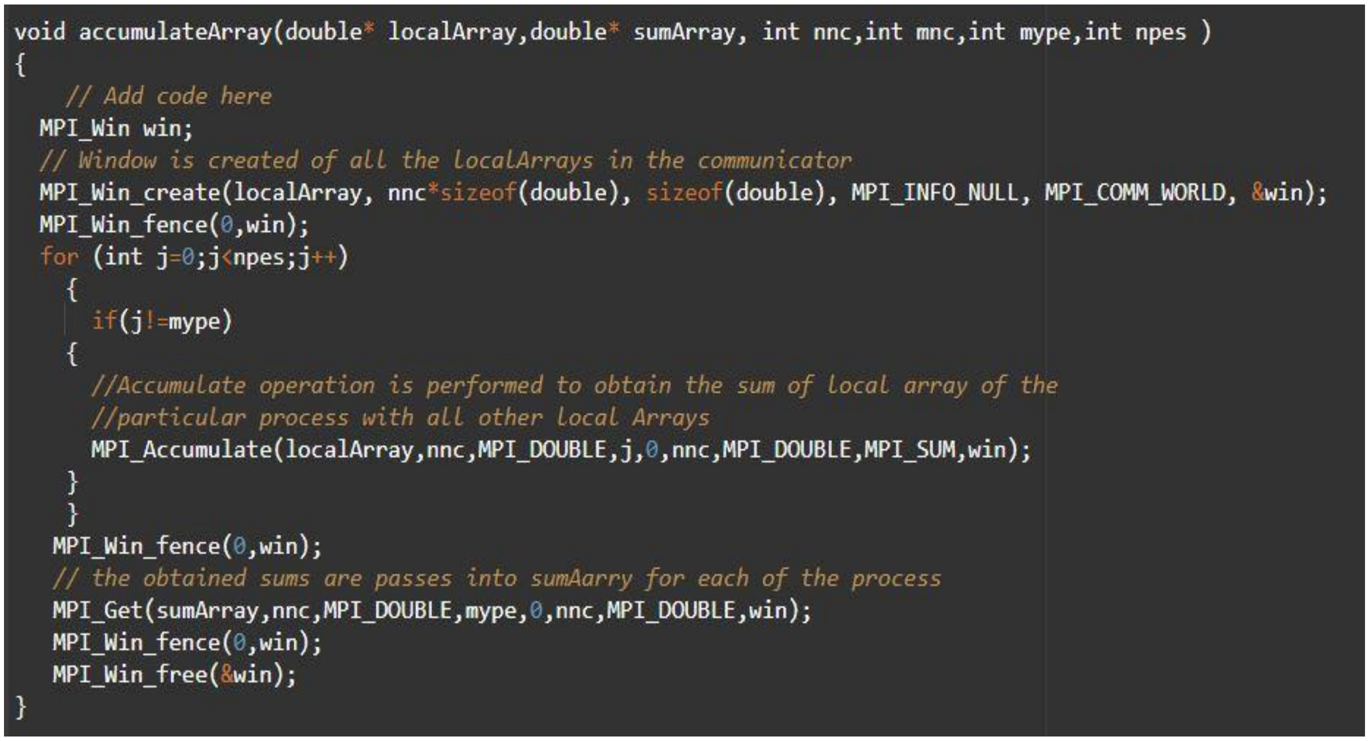
```
void localizeArray(double* localArray, double* globalArray,int lower, int upper, int nnc, int mnc, int mype, int npes)
{
    // Add code here
  // Window is created within COMM_WORLD communicator
  // so that the data in the winow can be accessed by all the other processes
    MPI_Win win;
    MPI_Win_create(localArray, nnc*sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
    // synchronysation point
    MPI_Win_fence(0,win);
    // MPI_Get is used to combine the data from local arrays into individual global array
    MPI_Get((globalArray + mype*mnc),nnc,MPI_DOUBLE,mype,0,nnc,MPI_DOUBLE,win);
    for(int i=0;i<npes;i++)
      {
    //implement  corner case, i.e for the process in which the size of array is smaller than mnc
    if(i!=mype)
      {
        if( nnc==mnc)
      MPI_Get((globalArray + i*mnc),nnc,MPI_DOUBLE,i,0,nnc,MPI_DOUBLE,win);
        if(nnc!=mnc)
      MPI_Get((globalArray + i*mnc),mnc,MPI_DOUBLE,i,0,mnc,MPI_DOUBLE,win);
      }
      }
    MPI_Win_fence(0,win);
    // the MPI window created is freed from memory space
    MPI_Win_free(&win);
}
```

Code Snippet 3: LocalizeArray function used to combine elements from local to global array

In this localizeArray function, the wincreate command is used to create the window, **MPI_Win_create** allows each process to specify a window in its memory that is made accessible to accesses by remote process. Thus in this part of code an object win of MPI WIN type is created, and window of local arrays of processes in communicator MPI_COMM_WORLD is created. Here the win would have localArray for each process which can be accessed by other remote processes in comm.

Then **MPI_Win_fence** is used to synchronize on the MPI window, All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns.

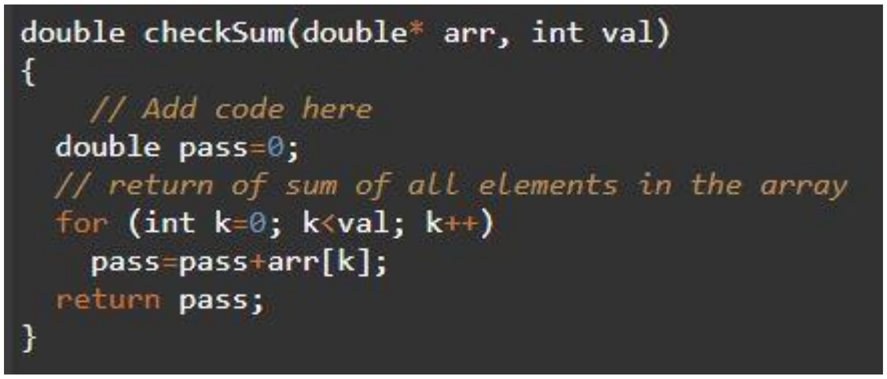
Next **MPI_Get** is used to receive data into the globalArray from the win, the first Get in the code snippet 3 is implemented to copy the data from the win which is localArray of the process that is executing the function, this is done be setting target rank as mype (the current process). After that, we loop across all the processes and again Get is used, but this time copy the values into global array of the entries of other processes localArrays which are in the window. After which again it is synchronized and memory is freed.



```
void accumulateArray(double* localArray,double* sumArray, int nnc,int mnc,int mype,int npes )
{
    // Add code here
  MPI_Win win;
  // Window is created of all the localArrays in the communicator
  MPI_Win_create(localArray, nnc*sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
  MPI_Win_fence(0,win);
  for (int j=0;j<npes;j++)
    {
      if(j!=mype)
    {
      //Accumulate operation is performed to obtain the sum of local array of the
      //particular process with all other local Arrays
      MPI_Accumulate(localArray,nnc,MPI_DOUBLE,j,0,nnc,MPI_DOUBLE,MPI_SUM,win);
    }
    }
   MPI_Win_fence(0,win);
   // the obtained sums are passes into sumAarry for each of the process
   MPI_Get(sumArray,nnc,MPI_DOUBLE,mype,0,nnc,MPI_DOUBLE,win);
   MPI_Win_fence(0,win);
   MPI_Win_free(&win);
}
```

Code Snippet 4: accumulateArray function used to sum elements of local array

In this section of code, the window is created as explained in previous section, then for all the processes other than the current process, the **MPI_Accumulate** is used, MPI Accumulate is used to sum the origin with the target. Here we can observe that the target rank is changed as we traverse across the loop and the process running through function will sum with the localArrays of the other processes.
In the end Get is used again to copy the values into the sumArray, which will be used in the next stage. Also note that since in accumulate the values are added in the localArrays of window, local array will be modified so copy of local array is made in main function.



```
double checkSum(double* arr, int val)
{
    // Add code here
  double pass=0;
  // return of sum of all elements in the array
  for (int k=0; k<val; k++)
    pass=pass+arr[k];
  return pass;
}
```

Code Snippet 5:Check sum function

Check sum function is used in display section, this function add the globalArray and sumArray, which are passed as arguments and returns the sum.

3 RESULTS

To get the results, the main vector is set to length 7 and number of processes to 3 (in run time),

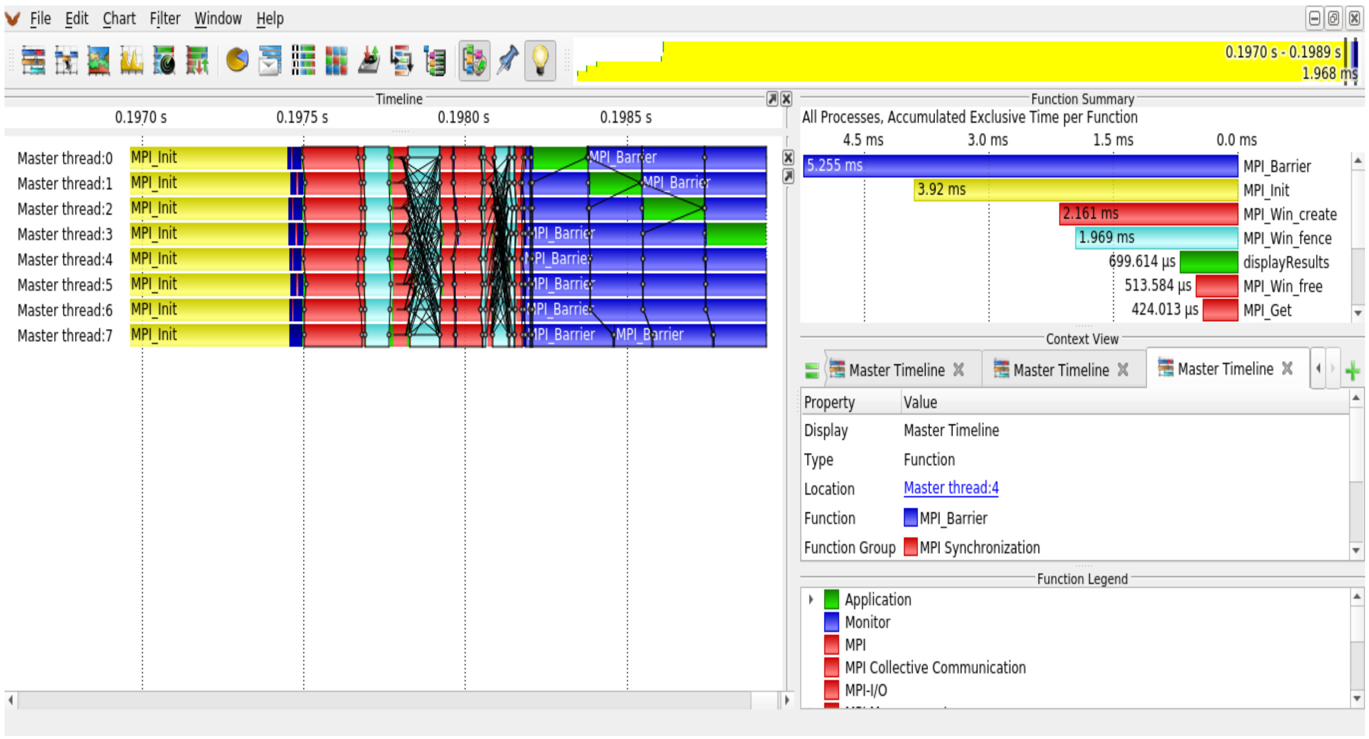
```
##################################################################
mype:0 nnc:3 mnc:3
mype: 0 localArray:
[ 8.40188 3.94383 7.83099  ]
mype: 0 globalArray:
[ -> 8.40188 3.94383 7.83099 <- 7.00976 8.09676 0.887955 5.6138 ]
mype: 0 sumArray[ 21.0254 12.0406 8.71895 ]

mype: 0 checkSum globalArray: 41.785
mype: 0 checkSum sumArray: 41.785

##################################################################
mype:1 nnc:3 mnc:3
mype: 1 localArray:
[ 7.00976 8.09676 0.887955  ]
mype: 1 globalArray:
[ 8.40188 3.94383 7.83099 -> 7.00976 8.09676 0.887955 <- 5.6138 ]
mype: 1 sumArray[ 21.0254 12.0406 8.71895 ]

mype: 1 checkSum globalArray: 41.785
mype: 1 checkSum sumArray: 41.785

##################################################################
mype:2 nnc:1 mnc:3
mype: 2 localArray:
[ 5.6138  ]
mype: 2 globalArray:
[ 8.40188 3.94383 7.83099 7.00976 8.09676 0.887955 -> 5.6138 ]
mype: 2 sumArray[ 21.0254 ]

mype: 2 checkSum globalArray: 41.785
mype: 2 checkSum sumArray: 21.0254
```

4 CONCLUSION

From the results we can see that, in this implementation the main vector is split with maximum uniform distribution among threads, i.e the 7 elements are divided into 3,3,1 elements for each of the processes respectively, this splitting is better than splitting up into 2,2,3 because in first case two processors handles maximum data compared to second case where only one processor handles maximum data. Thus gives better performance.

Further, we can see that local array is copied to global array as required which will serve the purpose of splitting up and adding. Also we can see that the sum of globalArray and sumArray using check sum matches when mnc is equal to nnc. This is because, when accumulation is performed when nnc and mnc are equal all the elements of both arrays will be summed if not few elements are not taken into sum.

In the bonus task, Vampire timeline software is used to find the communication pattern among various processes. It gives a description/visualization about which MPI commands/ routines are progressing along timeline and what communication pattern is happening amongst them.



All Processes, Accumulated Exclusive Time per Function

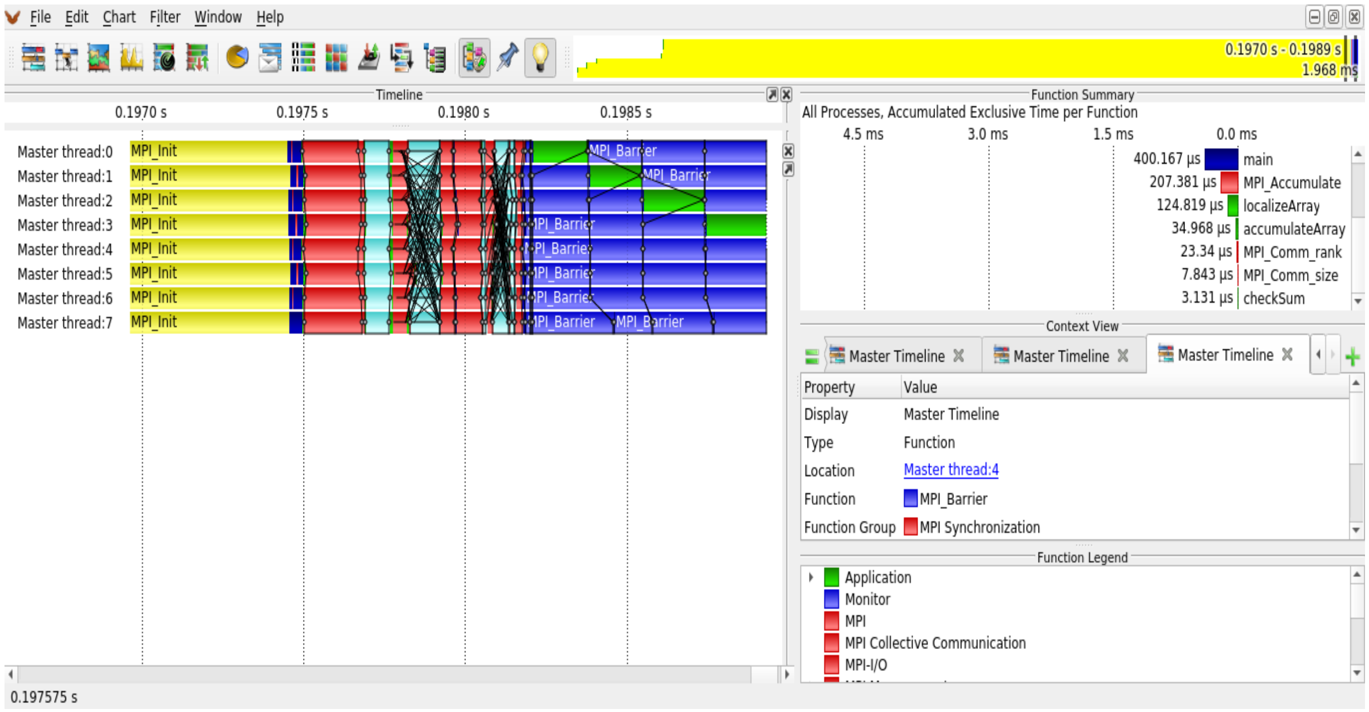
Time	Function
5.255 ms	MPI_Barrier
3.92 ms	MPI_Init
2.161 ms	MPI_Win_create
1.969 ms	MPI_Win_fence
699.614 µs	displayResults
513.584 µs	MPI_Win_free
424.013 µs	MPI_Get

Master Timeline | Master Timeline | Master Timeline

Property	Value
Display	Master Timeline
Type	Function
Location	Master thread:4
Function	MPI_Barrier
Function Group	MPI Synchronization

Function Legend

- 🟩 Application
- 🟦 Monitor
- 🟥 MPI
- 🟥 MPI Collective Communication
- 🟥 MPI-I/O



All Processes, Accumulated Exclusive Time per Function

Time	Function
400.167 µs	main
207.381 µs	MPI_Accumulate
124.819 µs	localizeArray
34.968 µs	accumulateArray
23.34 µs	MPI_Comm_rank
7.843 µs	MPI_Comm_size
3.131 µs	checkSum

Master Timeline | Master Timeline | Master Timeline

Property	Value
Display	Master Timeline
Type	Function
Location	Master thread:4
Function	MPI_Barrier
Function Group	MPI Synchronization

Function Legend

- 🟩 Application
- 🟦 Monitor
- 🟥 MPI
- 🟥 MPI Collective Communication
- 🟥 MPI-I/O

Fig 1.1 & 1.2: Vampire picture of timeline and communication

From the above pictures we can see that there are 8 threads/processes running and initially the main function is executed almost at the same time by all the threads we can see that there is no significant communication here, after that, around 0.190 seconds we can observe the complex communication among the threads, this is due to localizeArray function, since we use MPI fence a band is created due to synchronization, similar process happen again due to accumulateArray. The commands of wincreate, get and accumulate will have high complex communication but most of them happen simultaneously. Once this is done there is a Barrier before display, so program has to wait till all the processes reach this point which causes significant delay so it take 5.22 ms to run and display has to be given by each process one at a time. It will take considerable time. Apart from these, time is also taken for creation of window and etc. Over all we can see that although maximum communication happens in the functions does not take significant time, but MPI Barrier takes lot of time due to synchronization.

5 REFERENCES

1. https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-onesided.html
2. Assignment pdf uploaded for the Homework in moodle.
3. https://www.open-mpi.org/doc/v3.0/man3/MPI_Win_fence.3.php
   https://www.hpc2n.umu.se/documentation/compilers/flags
4. Lecture Notes of course parallel computing in computational mechanics- sisc 2020,moodle