REPORT: PROJECT 3

Parallel solver for unsteady 2D Heat Equation using "Message Passing Interface (MPI)"

German Research School for Simulation Sciences GmbH
Laboratory for Parallel Programming

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

1. INTRODUCTION:

The project on the solver of 2D unsteady heat equation illustrates that the effectiveness of parallel programming in scientific simulation and computations. Where multiple processing elements solves the given problem simultaneously, right implementation of parallel computing helps in appropriate utilization of the available computational resources, thus reducing the overall runtime when comparison with the serial code and enables us to solve the complex problems which requires heavy computational load.

In this project, two different types of parallelizing techniques were implemented investigated. In the project one, serial version of 2D Unsteady Heat Equation was modeled and solver was implemented. Taking a step further into the same solver, in the project two the serial solver was more optimized using appropriate compiler flags. Further, the code is parallelized used in OpenMP by avoiding data-race, here while using OpenMP we considered the system to be using virtual shared memory space where all threads can access the virtual uniform memory address space, in this project work sharing techniques was used to distribute workload among threads. Further in the project three, Message Passing interface (MPI) communications between multiple processors is used and compared with the serial code. In MPI each processor will have its own memory address space and the data between these processors are transferred using MPI communication commands. Detailed implementation is presented in the further sections.

2. IMPLEMENTATION:

2.1 tri.cpp

```
nec = (ne-1)/npes+1;
mec = nec;
if ((mype+1)*mec>ne)
  {
nec = ne-(mype*mec);
  }
if (nec<0)
  {
nec=0;
  }
// ADD Code To
// Determine nec,mec
// TODO
// the division of all nodes into nodes per processors
// algorithm is as discussed in lecture
nnc = (nn-1)/npes + 1;
mnc = nnc;
if ((mype+1) * mnc > nn)
  {
nnc = nn - mype*mnc;
  }
if (nnc < 0)
  {
nnc = 0;
  }
cout << "mype:" << mype << " nnc:" << nnc << " mnc:" << mnc << " nec:" << nec << endl;
```

Code excerpt 1

In this part of the code of readmeshfile function of trimesh class, firstly total elements (ne) is divided among all the available processors, but most likely this will not be a properly divisible, so we can check that, if the next processor is more than array size bound , we can allocate those left over elements in that processor. Same logic also applies to the total nodes (nn), when they are distributed among processors. This logic was explained in the lecture **[1].**

```
void triMesh::localizeNodeCoordinates()
{
    int mype, npes;               // my processor rank and total number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);

    //defining a buffer, to store x,y values for nodes
    double buffer[mnc*nsd];
    // creating a window, with the x,y values of all the preocessors
    // which can be accessed by any of the processors with MPI operations
    MPI_Win win1;
    MPI_Win_create(xyz,(nsd*nnc*sizeof(double)),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD, &win1);
    MPI_Win_fence(0, win1); //synchronyzation point
    //for looping over the processors
    for(int ipe = 0; ipe < npes; ipe++)
    {
      //initialising the buffer for each iteration
      for(int k=0;k<(mnc*nsd);k++)
    {
      buffer[k]=0;
    }
```

Code excerpt 2

In this part of the code, the rank and number of active processors are stored, then a buffer is created to store the coordinate values for each processor. In the next step a window (win1) is created with the xyz private array belonging to trimesh class, since localizeNodeCoordinate() function is a member of this class, this array can be accessed directly. This window can be accessed by other processors in the communicator MPI_COMM_WORLD.

```
  int randimax;
  //randimax is used account for the size of nodes for the last pe which might have
  //smaller size
  if ((ipe+1)*mnc>nn)
{
  randimax = max(0,nn-(ipe* mnc));
}
  //for other processes the size is already known which is max num of elements
  else
{
  randimax = mnc;
}
  // MPI Get is used to get the x,y values from the window of 2*randimax elemnts of double
  // datatype, where the target rank is the particular processor ipe
  MPI_Get(&buffer[0],(nsd*randimax),MPI_DOUBLE,ipe,0,(nsd*randimax),MPI_DOUBLE,win1);
  MPI_Win_fence(0,win1); //synchoronization point
```

Code excerpt 3

Then we loop over processors count, and initialize the buffer each time, because every iteration the new processor (ipe) is the new target rank in MPI GET, so the buffer has to store the local values of ipe. Here I define a variable randimax to store the exact number nodes that can be stored for each processor, since the current processor can't know the 'nnc' of other processors. Then from each of these processors (ipe) gets the coordinate values from the window is storea into the buffer.

```
    for(int inl = 0; inl<nnl; inl++)
    {
      //is used to get the global sorted node number for the local node number with respect
      //to elements in the processor
      int node_num = nodeLToG[inl];
      // to identify in which processor the current node falls in
      int mpe = (node_num/mnc);

      if(mpe == ipe)
        {
          //the stride is calculated
          int jump = (node_num%mnc);
          //the x,y coordinate values are stored into tht trinode class
          lNode[inl].setX(buffer[jump*nsd+xsd]);
          lNode[inl].setY(buffer[jump*nsd+ysd]);
        }
    }
    }
    MPI_Win_free(&win1);//freeing the memory associated with the window
    return;
}
```

Code excerpt 4

Now the next for loop meets the real objective of the localizeNodeCoordinate function, where the for loop, loops across all the nodes (nnl) corresponding to the elements in current processor, now we get the global node number of the sorted array, now we check if the outer iterating loop processor has the value that is needed for the current processor (mpe), this logic is given in lecture **[1]**, if it matches we calculate the stride and store the corresponding value from the buffer into the lNode array.

2.2 solver.cpp

In the explicitsolver function of the femsolver class, two MPI windows (winMTnew and winTG) are created with MTnewG (of size nnc) and TG (of size nnc) arrays which are members of trimesh class, but they are accessed by public getter functions. These can be accessed by other processors in the communicator. Similarly the mnc is taken from public member function of class trimesh. In the end localizeTemerature function is called with winTG as argument because we need the values of local temperature to calculate the MTnew. So all the data in the TG has to be localized to each processor what contain that element so ML has to have dimension of nnl.

```
// Complete MPI Communication here
// Here, the windows for the MTnew and TG are created
MPI_Win winMTnew;
MPI_Win winTG;
// Add MPI Communication here
//using getter function to get maximum number of nodes across all PE's
int mnc = mesh->getMnc();

//the windows for the MTnew and TG are created
MPI_Win_create(MTnewG,(mnc*sizeof(double)),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD,&winMTnew);
MPI_Win_create(TG,(mnc*sizeof(double)),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD,&winTG);

localizeTemperature(winTG);
```

Code excerpt 5

Once the MTnewL is calculated locally among processors this has to be accumulated again into MTnewG this can be achieved by calling accumulateMTnew function with winMTnew as window. This is needed to calculate the partial error and new global temperature. In the end all the windows are freed from the memory space.

```
// Add MPI Communication here
    accumulateMTnew(winMTnew);
```

```
// Add MPI Communication here
localizeTemperature(winTG);
```

```
// Add MPI Communication here
MPI_Win_free(&winMTnew);
MPI_Win_free(&winTG);
```

Code excerpt 6 (a),(b),(c):

Now each of these functions are explained below,

```
void femSolver::accumulateMass()
{
    int mype, npes;               // my processor rank and total number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    //ADD CODE HERE
    //using getter functions to get necessary data
    int nn = mesh->getNn();int nnl=mesh->getNnl();int nnc=mesh->getNnc();int mnc=mesh->getMnc();

    //Creating a buffer of for size mnc dynamically
    double *buffer;
    buffer=new double[mnc*sizeof(double)];

    //A window is created with global mass
    MPI_Win win; double *massG = mesh->getMassG();
    MPI_Win_create(massG,(mnc*sizeof(double)),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD,&win);
```

Code excerpt 7

In accumulate mass function, initially using public getter function of trimesh class, then requires parameters are obtained. Then dynamic arrays is created, first is buffer of size (mnc), i.e maximum number of node among individual processors. In the end the window is created for global mass, these global mass in can be accessed by all the processors. So when accumulate id used all the values in buffer is added into the GlobalMass.

```
int innc;
for(int ipes=0;ipes<npes;ipes++)
{
   if (npes==1)
   {
      innc=nn;
   }
    //This is the logic for calculating nnc for each pe, so that
    // the value could be accumulated from the window into the buffer
    else if(((ipes+1)*mnc)>nn)
    {
        //same logic as explained in tri.cpp
        innc=max(0,(nn-ipes*mnc));
    }
    else
    {
        innc=mnc;
    }
     //initializing buffer
    for(int i=0;i<mnc;i++)
    {
        buffer[i]=0;
    }
```

Code excerpt 8

In this section of the code, similar to what was done in localizenodecoordinate function in tri.cpp file, first we loop across the number of processors, g_node is used to store the global node numbers obtained by getNodeLToG function. Then we calculate the number of nodes corresponding to each processor (innc), analogous to randimax in tri.cpp, then the buffer is initialized.

```
        for(int inode=0; inode<nnl; inode++)
        {
            //getting the global node number
            int g_node= mesh->getNodeLToG()[inode];
            //as explained in lecture, seeing in which PE the node is in.
            int ipe= g_node/mnc;
            if(ipes==ipe) //if its in current PE
            {
                int index= g_node%mnc; //Finding the index as shown in lecture 11
                buffer[index]=  mesh->getLNode(inode)->getMass();
            }
        }
    MPI_Win_fence(0,win);
    // the value in the buffer is added to the Global mass, this will be done be every PE
    MPI_Accumulate(&buffer[0],innc,MPI_DOUBLE,ipes,0,innc,MPI_DOUBLE,MPI_SUM,win);
    MPI_Win_fence(0,win); //Synchronization Point
    }
    free(buffer); //Freeing dynamic array
    return;
}
```

Code excerpt 9

Here, again we loop over all the nodes corresponding to elements in the current processor and global node number is obtained, again we check if the processor of the loop match with the global node residing processor, then the value from local mass array is transferred into the buffer, then **MPI_Accumulate** is used to **add** the values in the buffer into global mass array which is in the window, which corresponds to the target rank defined by ipe. After this, the memory of the dynamically allocated arrays is freed

As an answer to the question asked in the Assignment PDF, we cannot use MPI_Put here because put overwrites the values, whereas here we need the combined added values/contribution from all the processors, so when we use accumulate the buffer values are added to global mass in the window. By this we get the total sum of all the mass associated with the particular processor. Same argument holds for using the accumulate command in the accumulateMTnew function.

```
void femSolver::accumulateMTnew(MPI_Win win)
// Noted that the window created before is taken as argument which has global mass
{
    int mype, npes;               // my processor rank and total number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    //ADD CODE HERE
    //using getter functions to get necessary data
    int nn = mesh->getNn();int nnl=mesh->getNnl();int nnc=mesh->getNnc();int mnc=mesh->getMnc();

    //Creating a buffer of for size mnc dynamically
    double *buffer;
    buffer=new double[mnc*sizeof(double)];

    //pointer to array MTNewL
    double *MTnewL = mesh->getMTnewL();
```

Code excerpt 10

The operations performed in the accumulateMTnew function is similar to that of accumulatemass function, but here window object win is passed as operator with MTnewG as pointer, and we obtain another pointer to MTnewL. Next part is also similar to previous function, where number of nodes is calculated.

```
int innc;
for(int ipes=0;ipes<npes;ipes++)
{
   if (npes==1)
   {
      innc=nn;
   }
    //This is the logic for calculating nnc for each pe, so that
    // the value could be accumulated from the window into the buffer
    else if(((ipes+1)*mnc)>nn)
    {
        //same logic as explained in tri.cpp
        innc=max(0,(nn-ipes*mnc));
    }
    else
    {
        innc=mnc;
    }
     //initializing buffer
    for(int i=0;i<mnc;i++)
    {
        buffer[i]=0;
    }
```

Code excerpt 11

```
        for(int inode=0; inode<nnl; inode++)
        {
            //getting the global node number
            int g_node= mesh->getNodeLToG()[inode];
            //as explained in lecture, seeing in which PE the node is in.
            int ipe= g_node/mnc;
            if(ipes==ipe) //if its in current PE
            {
                int index= g_node%mnc; //Finding the index as shown in lecture 16
                buffer[index]=  MTnewL[inode];
            }
        }
    MPI_Win_fence(0,win);
    // the value in the buffer is added to the Global mass, this will be done be every PE
    MPI_Accumulate(&buffer[0],innc,MPI_DOUBLE,ipes,0,innc,MPI_DOUBLE,MPI_SUM,win);
    MPI_Win_fence(0,win); //Synchronization Point
    }
    free(buffer); //Freeing dynamic array
    return;
}
```

Code excerpt 12

Here again its similar to earlier function but in the buffer we collect the MTnewL value of the matching node and accumulate with MTnewG vector.

Again for localizeTemperature function the getting values and initialization is similar to other two functions, in accumulatemass section it's explained in detail.

```
void femSolver::localizeTemperature(MPI_Win win)
{
    // Noted that the window created before is taken as argument which has global mass
    int mype, npes;               // my processor rank and total number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    //ADD CODE HERE
    //using getter functions to get necessary data
    int nn = mesh->getNn();int nnl=mesh->getNnl();int nnc=mesh->getNnc();int mnc=mesh->getMnc();

    //Creating a buffer of for size mnc dynamically
    double *buffer;
    buffer=new double[mnc*sizeof(double)];

    //Pointer to local temperature array
    double *TL = mesh->getTL();
```

Code excerpt 13

```
int innc;
for(int ipes=0;ipes<npes;ipes++)
{
   if (npes==1)
   {
      innc=nn;
   }
    //This is the logic for calculating nnc for each pe, so that
    // the value could be accumulated from the window into the buffer
    else if(((ipes+1)*mnc)>nn)
    {
        //same logic as explained in tri.cpp
        innc=max(0,(nn-ipes*mnc));
    }
    else
    {
        innc=mnc;
    }
     //initializing buffer
    for(int i=0;i<mnc;i++)
    {
        buffer[i]=0;
    }
```

Code excerpt 14

In the last section it's different from other two functions above; here the using MPI_Get the value of the global temperature is transferred into the buffer, and then when the global node of the receiving processer match with the current loop iteration, the value in the buffer is transferred into the local temperature array.

```
        MPI_Win_fence(0,win);
        //Into the buffer the values of temperature TG are transferred, which are in window
        MPI_Get(&buffer[0],innc,MPI_DOUBLE,ipes,0,innc,MPI_DOUBLE,win);
        MPI_Win_fence(0, win);
        for(int inode=0; inode<nnl; inode++)
        {
            //getting the global node number
            int g_node= mesh->getNodeLToG()[inode];
            //as explained in lecture, seeing in which PE the node is in.
            int ipe= g_node/mnc;
            if(ipes==ipe) //if its in current PE
            {
                int index= g_node%mnc; //Finding the index as shown in lecture 16
                TL[inode]=  buffer[index];
            }
        }
    }
    free(buffer); //Freeing dynamic array
    return;
}
```

Code excerpt 15

3.1 TEMPERATURE DISTRIBUTION for Serial and Parallel Code:

The accuracy of the result is the most important aspect; the improvement in performance is useful only if the results fall in the acceptable range. One way to verify the parallelization is by observing and comparing the temperature distribution of serial and parallelized solution.

Here, I will consider coarse, fine mesh and present the temperature distribution obtained by serial code, then show that in parallel code by using 4 tasks for same meshes the temperature distribution remains the same. Same can be extended to finest mesh.

Anyhow in the next section it will be shown that the parallel code is independent of number of cores.

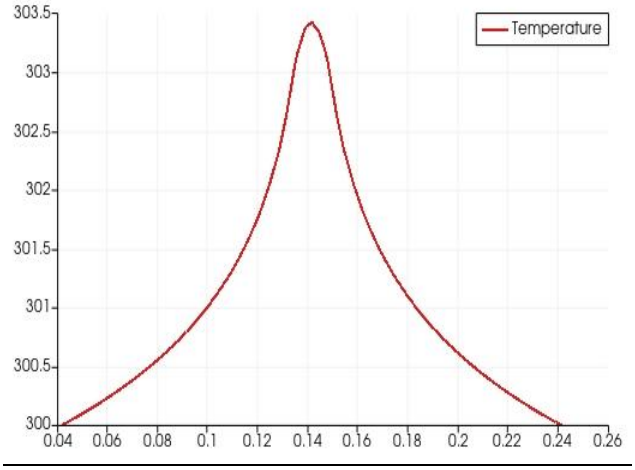


Fig 1: Coarse mesh with serial code from project 1.

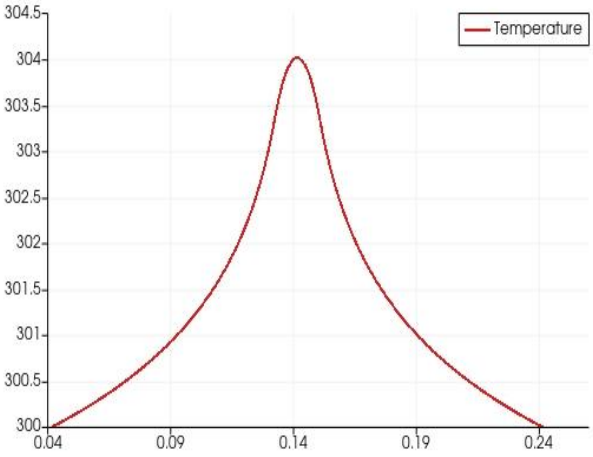


Fig 2: Fine mesh with serial code from project 1.

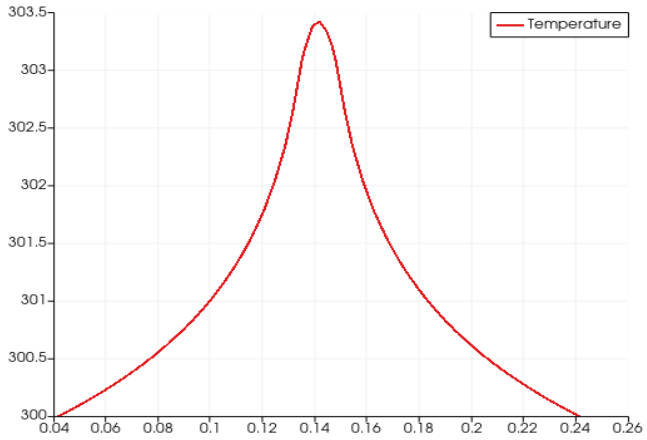


Fig 3: Coarse mesh with Parallel code.

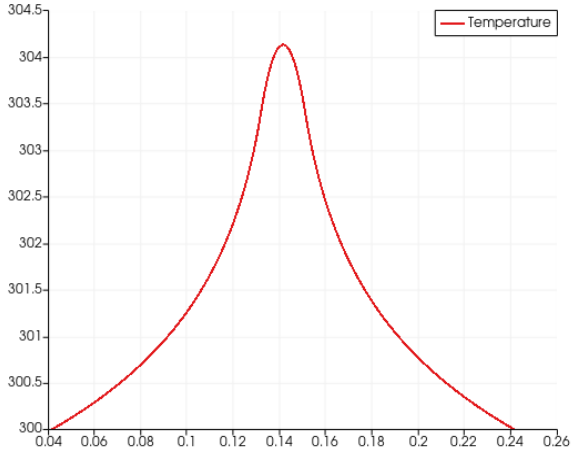


Fig 3: Fine mesh with Parallel code.

From these graphs we can conclude that, by parallelizing the code the final temperature distribution is not changed by parallelizing, thus we can proceed with further investigation.

3.2 SOLUTION WITH DIFFERENT NUMBER OF CORES

For the coarse and fine meshes the temperature distribution for varying number code is shown below, we can see that it is unaltered, this is not a surprise because using MPI Communications properly, and we can divide the tasks and workload among processors and get correct solution.

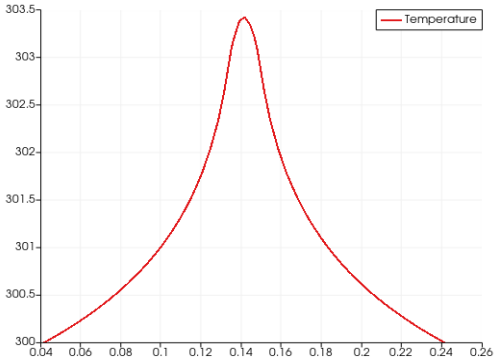


Fig 1: Coarse mesh with 4 cores

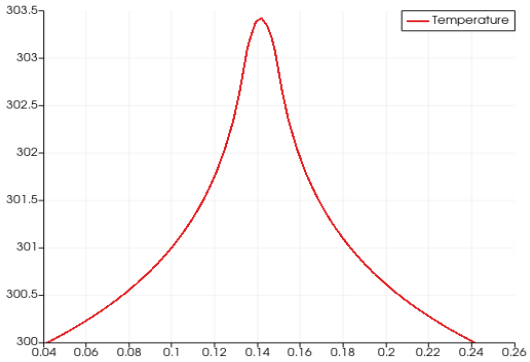


Fig 2: Coarse mesh with 8 cores

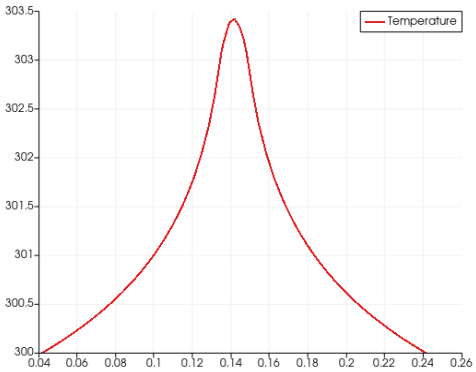


Fig 3: Coarse mesh with 16 cores

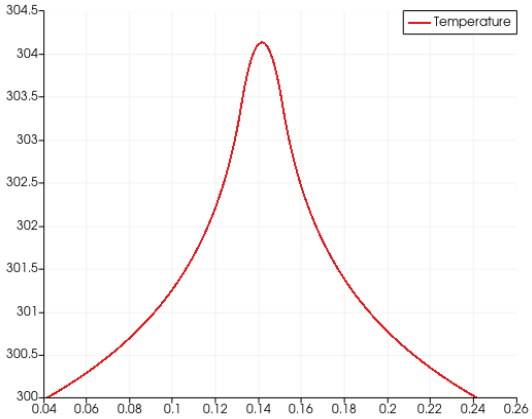


Fig 4: Fine mesh with 4 cores

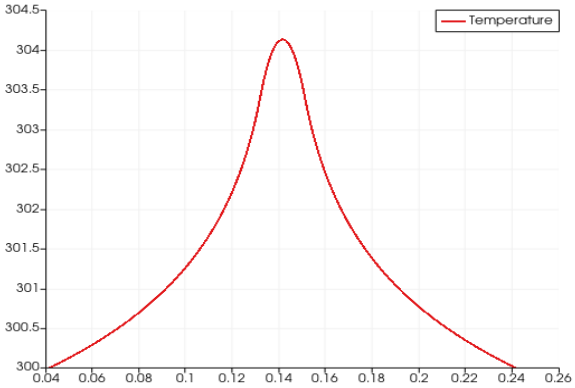


Fig 5: Fine mesh with 8 cores

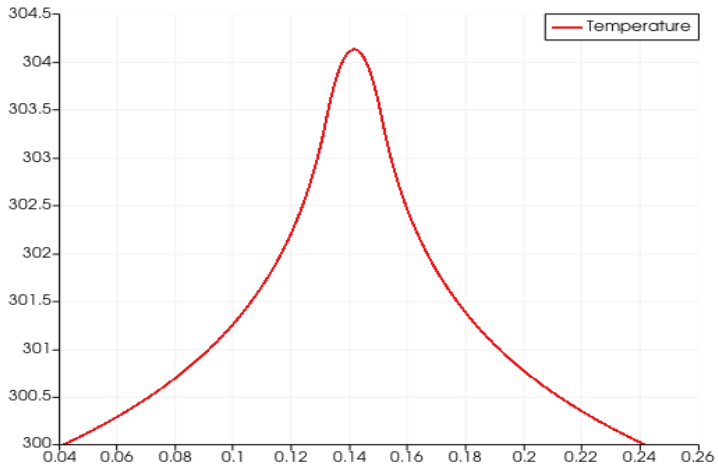


Fig 6: Fine mesh with 16 cores

Further, here we can see that the RMS Error is the same for a particular mesh irrespective of using different number of cores. So solution is same irrespective of number of cores employed.

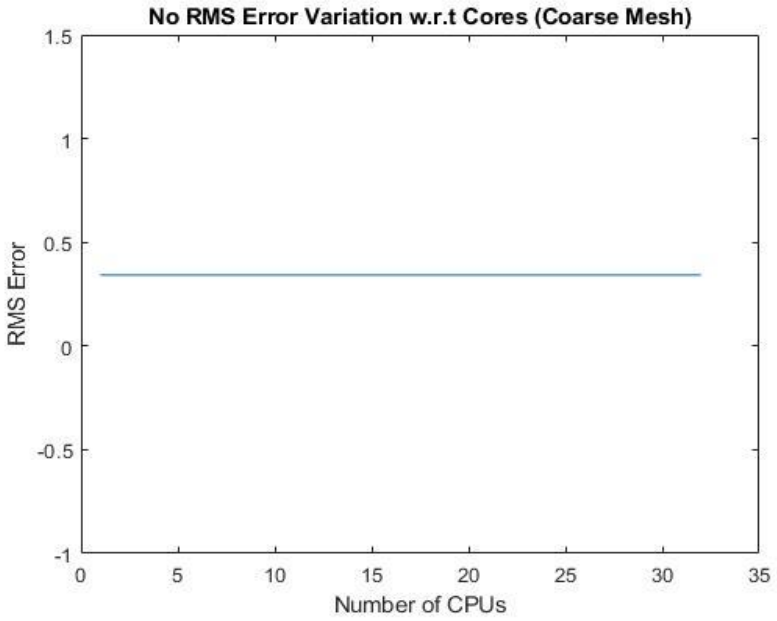


Fig 4: RMS Error for coarse mesh with different number of nodes

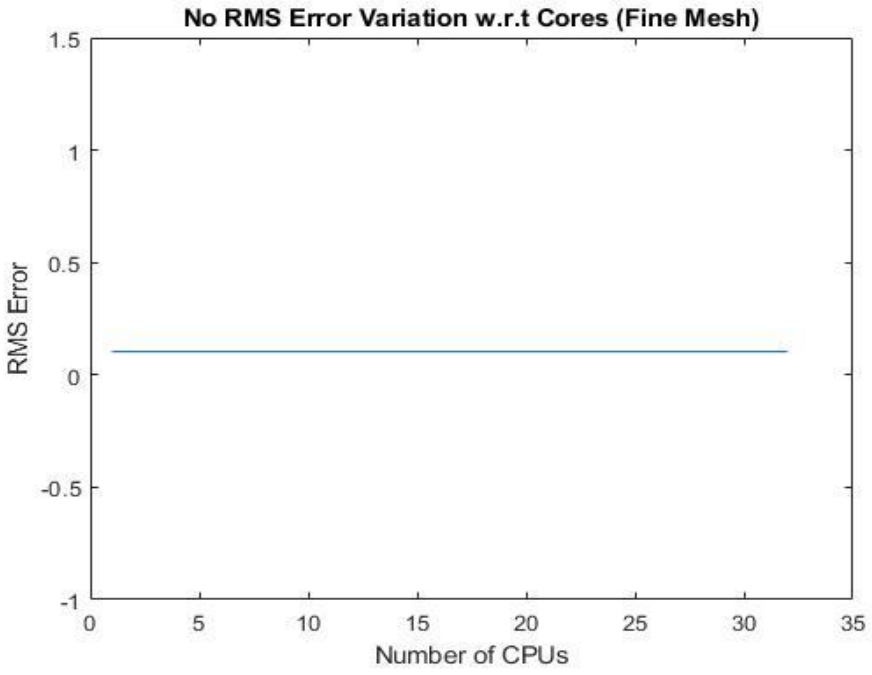


Fig 5: RMS Error for Fine mesh with different number of nodes

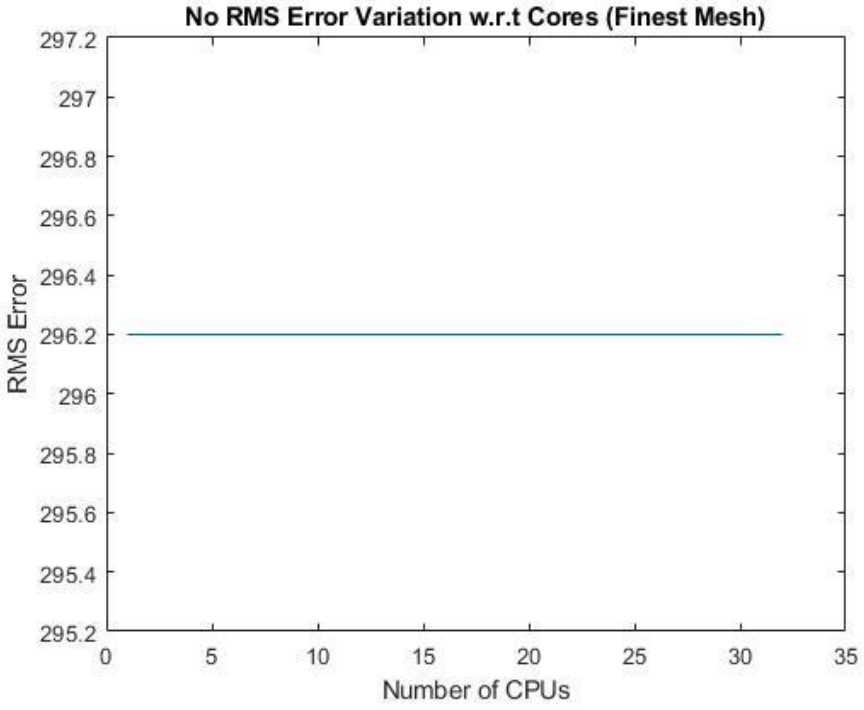


Fig 6: RMS Error for Finest mesh with different number of nodes

3.3 SCALING AND EFFICIENCY PLOTS:

The time taken for the parallel code on various cores is listed in table 1, below. Using this data the plots for scaling/speed up and efficiency are made using Matlab.

Speed Up: $S_p = T_1/T_p$, where 'T_1' is time required by the code on one processor and 'T_p' is time required by the code on 'p' processor

Efficiency: $E_p = S_p/p$ with same notations.

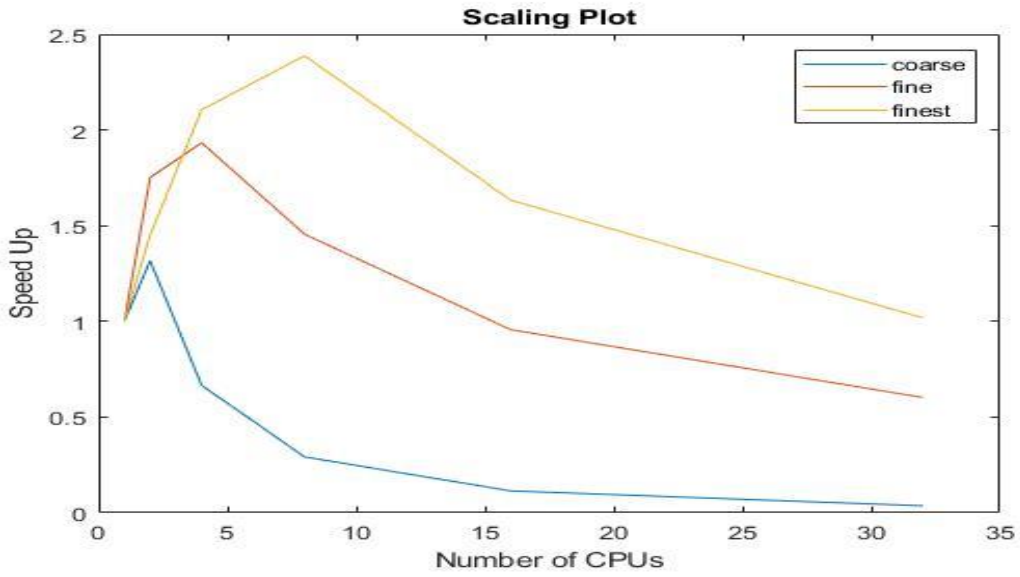


Fig 1: Speedup v/s CPUs plot for various meshes.

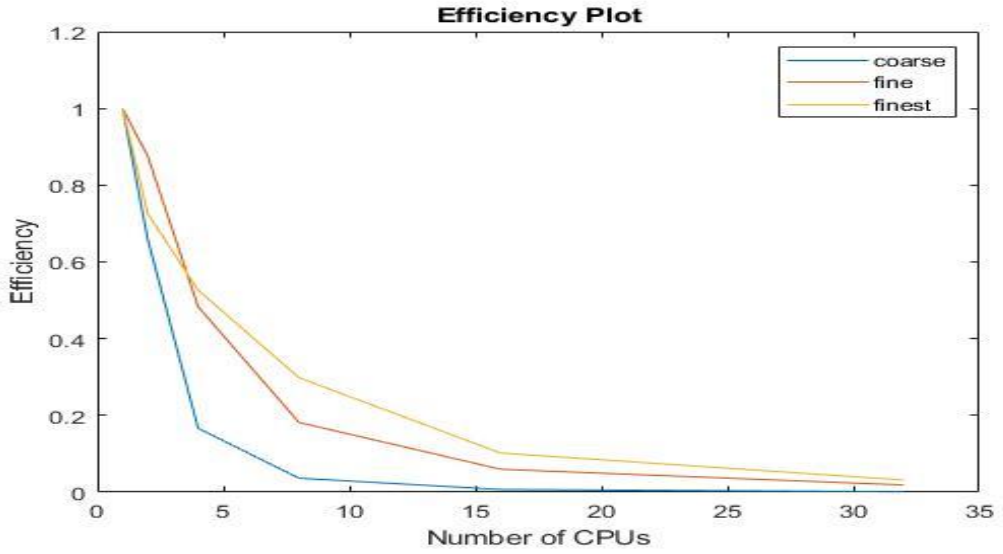


Fig 2: Efficiency v/s CPUs plot for various meshes.

From Scale-up plot figure 1, we can observe that there is speedup in spite of increasing few cores, but speedup decreases when number of cores increases more than 8, this is maybe because, even after parallelizing, there is not significant reduction in runtime due to high overhead by communication and synchronization . On the other had for fine and finest meshes this effect is observed slightly towards larger number of cores, because they have more number of elements, the work is well distributed. Thus workload per processor dominates overhead caused by communication and synchronization.

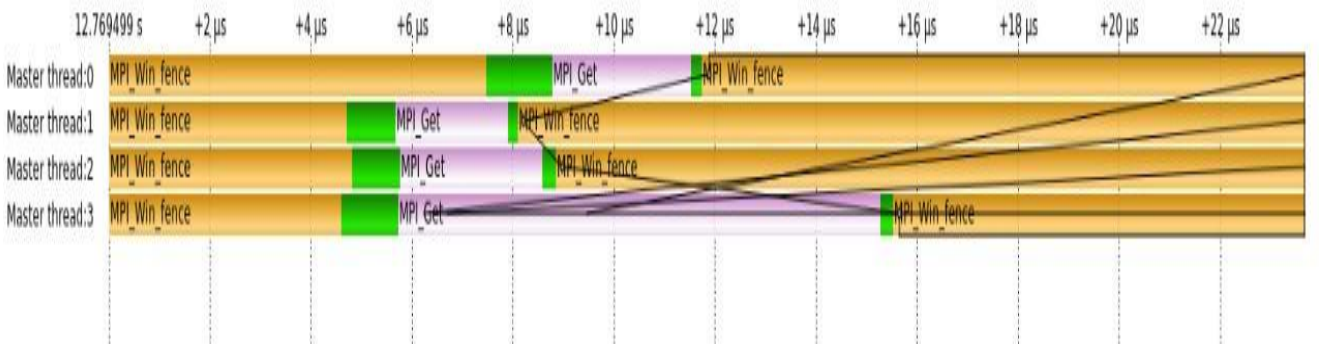
In the Efficiency plot figure 2, the general trend is that as the number of CPU's increase, the efficiency decreases. Because as the number of processes and cores increases, the synchronization becomes an important issue, even though the speed up is observed in the fine and finest meshes, the ratio of that with cores is not high as number of cores increases.

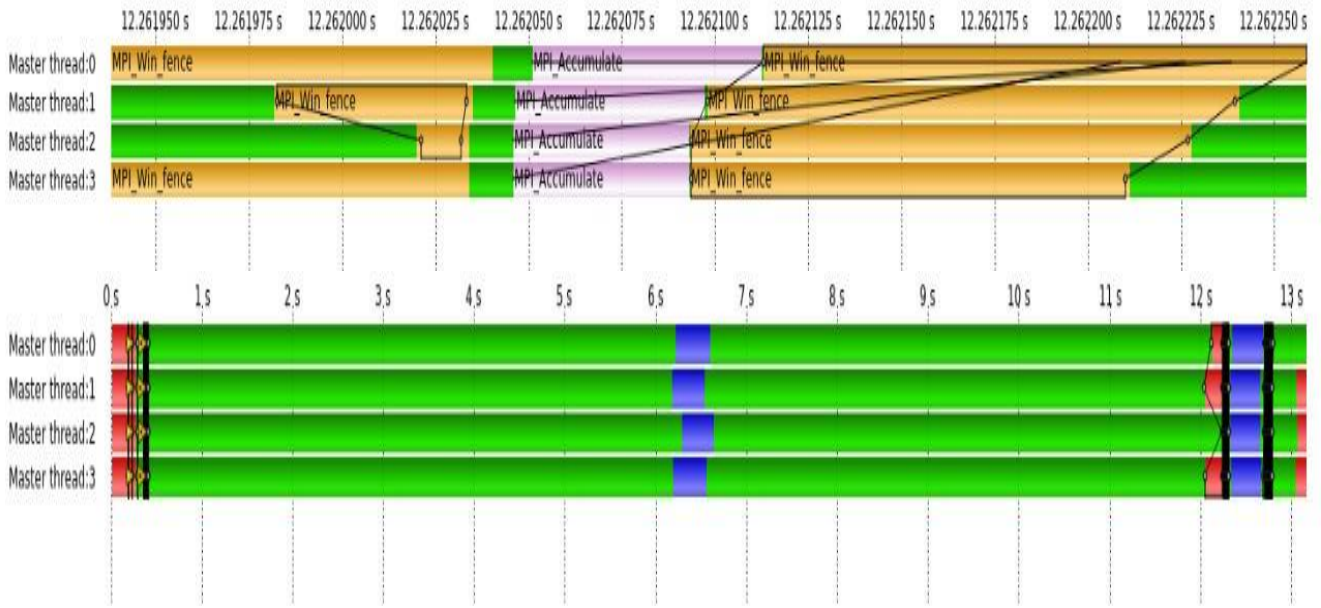
Cores	Coarse	Fine	Finest (1001 Iterations)
1	1.2	627.63	157.8
2	0.91	358.43	108.8
4	1.80	324.67	74.98
8	4.12	431.34	66.12
16	10.51	646.00	96.64
32	32.87	1041.75	154.9

Table 1: Time for different cores and meshes (in Seconds).

Further, as an ***answer to the question*** in the PDF, we can further improve the code by using two sided communication, but we have to be careful for not causing deadlocks and more communication overhead. Further, in the assembly of residuals, we can maybe use the two steps partitioning to make it run faster.

In this section we can see the communication pattern obtained from vampire with 4 processors,





From the picture above we can observe that, there is lot of communication happening among the processors, these is high overhead created from the MPI_Barrier and MPI_Win_fence, We can see that all the communication happening with Master thread when MPI_Accumulate is used. Further is some synchronyzation happening with MPI_Get.

4 CONCLUSIONS:

From the analysis shown above we can say that, by parallelizing the code we get overall similar temperature distribution, in general similar solution but we would use the available hardware effectively, efficiently and get reduction in overall runtime, thus parallelizing the serial code using techniques such as MPI and OpenMP (previous project) is very advantageous for especially larger problems with high computational load. But parallelizing the code in particular with MPI is a bit complex and confusing. Further, we can say once the code is parallelized, we can employ as many cores/CPU's as needed for workload distribution. None the less, adding more CPU's does not always result in speed up. After a point more cores than need will result in decrease in overall runtime, due to increase in communication overhead and synchronization. Also from the speed up graph we can see that for finest meshes (more computation) which results in better load to count ratio per processor, will result in better speedup. Also with increasing cores there is decrease in efficiency. Finally it can be concluded there is a significant increase in performance of parallel code, in terms of run time when optimal number of cores are used depending on workload.

5 REFERENCES

1. Lecture Notes of course parallel computing in computational mechanics- sisc 2020, moodle
2. https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node237.htm
3. https://vampir.eu/tutorial