Parallelizing unsteady 2D heat equation solver with Message Passing Interface (MPI)

Mridul Mani Tripathi

German Research School for Simulation Sciences GmbH
Laboratory for Parallel Programming
mridul.tripathi@rwth-aachen.de

Abstract: The aim of this project is to parallelize the unsteady 2D heat equation solver using Message Passing Interface (MPI). The elements and nodes are distributed among the available processors. The computation is done independently on each processor and can be sent or received to and from other processors when necessary. A brief description of the solver is provided as follows: mesh file is read first and its data are stored. Element matrices M, K, F are calculated after calculating and storing jacobian for each element. Dirichlet boundary condition is applied. For solving the given heat equation, explicit first order time integration scheme is used. Further error is being calculated to check deviation from analytical solution.

1 Introduction

Message Passing Interface (MPI) is a standardized application programming interface that allows one to provide unambiguously the interface with the precise sematic of communication protocols and global calculations routines, among others. Thus, a parallel program using distributed memory can be implemented using various implementations of the MPI interface. This project aims to parallelize the unsteady 2D heat equation solver using MPI. The MPI communication routines used in this project are MPI_Accumulate, MPI_Get along with window creation and free communications such as MPI_Win_create, MPI_Win_fence, and MPI_Win_free. The solver is then tested on multiple cores to get speed-up and efficiency analysis.

This project of developing a solver for a 2D unsteady heat equation is divided in three stages. In project one, solver for a 2D unsteady heat equation was modelled and implemented in a very primitive way. In project two, the solver was sped up using multiple compiler optimizations and then parallelized using OpenMP routines. In project three, Message Passing Interface (MPI) communications between multiple processors is used.

2 Description of problem

The problem at hand is to parallelize the finite element code for a 2D unsteady heat equation solver with Message Passing Interface (MPI). The the file tri.cpp, the elements and nodes are distributed among the available processors as equally as possible. After partitioning all elements and nodes among processors, we get the values of nec, mec, nnc, and mnc.

nec: Number of elements stored on the current PE

mec: Maximum number of elements across all PEs

nnc: Number of nodes stored on the current PE

mnc: Maximum number of nodes across all PE.

Next step is to implement the localization of the coordinate data inside localizeNodeCoordinates(). In the file solver.cpp three functions must be completed. First, we transfer data from node class to an array massG of size nnc. Further, temperature data is transferred to TL from TG along with transferring data from MTnewL to MTnewG. Appropriate communication routine should be added wherever necessary. In the end, the result should be compared with analytical solution to verify its correctness.

3 Methodology

Excerpts of the required parts of the code is presented below along with explanations.

```
// ADD Code To
// Determine nec, mec

nec=ceil((double)ne/(double)npes);          // Dividing elements among PEs as equally as possible
mec=nec;                                    // mec is maximum value of nec

if((mype+1)*mec>ne)
   nec=ne-mype*mec;

if(nec<0)
   nec=0;


// ADD Code To
// Determine nnc, mnc

nnc=ceil((double)nn/(double)npes);          // Dividing nodes among PEs as equally as possible
mnc=nnc;                                    // mnc is maximum value of 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 except 1: Excerpt from readMeshFiles()

All elements (ne) are distributed among the available processors (npes) as equally as possible. nec is chosen such that it is equal to smallest integer greater than or equal to quotient of ne/npes. mec is the equal to maximum value of nec across all processors. Using same logic, all nodes (nn) are also distributed among the processors with mnc being the maximum value of nnc.

```
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);

   //ADD CODE to make the following statements work

   double buffer[mnc*nsd];              // buffer will store x and y  coordinate values of nodes
   int count,inc,num,proc;

   MPI_Win win;
   MPI_Win_create(xyz,nnc*nsd*sizeof(double),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD,&win);
   MPI_Win_fence(0,win);                // Window stores coordinates of nodes to be accessed by all processors

   for(int i=0;i<npes;i++)
      {
      for(int j=0;j<(mnc*nsd);j++)
         {buffer[j]=0;}


      if((mnc*i)>(nn-mnc))              // count variable is used for number of nodes assigned to last processor
         {count=max(0,nn-mnc*i);}
      else
         {count=mnc;}
```

```
    MPI_Get(&buffer[0],nsd*count,MPI_DOUBLE,i,0,nsd*count,MPI_DOUBLE,win);
    MPI_Win_fence(0,win);            // Storing coordinates values of nodes in buffer

    for(int inl=0;inl<nnl;inl++)
      {
      num=nodeLToG[inl];             // Determining global node number of corresponding local node
      proc=num/mnc;                  // Determining processor rank of the node

      if(proc==i)                    // For current pe
        {
        inc=(num%mnc);               // Determining stride
        lNode[inl].setX(buffer[inc*nsd+xsd]);          // Localization of coordinate data
        lNode[inl].setY(buffer[inc*nsd+ysd]);
        }
      }
    }

  MPI_Win_free(&win);
  return;
}
```

Code excerpt 2: Excerpt from localizeNodeCoordinates()

The buffer stores coordinate values i.e. x and y. Window is created to access these coordinate values from any processor. The buffer is initialized during each iteration since in different iterations, the target rank is different and thus new values must be stored in buffer. Variable count stores number of nodes that can be assigned to each processor. We now determine the global node number of the node and if the processor rank matches with the current processor rank, stride is determined and corresponding value from buffer is stored inlNode. Here nnl is the number of nodes associated with the elements in the current processor.

```
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

  int cnnc,proc,in;

  int mnc=mesh->getMnc();
  int nnc=mesh->getNnc();
  int nn=mesh->getNn();
  int nnl=mesh->getNnl();
  double *massG=mesh->getMassG();

  int nodeLToG;                                    // Stores global node number

  double *temp;
  temp=new double[mnc*sizeof(double)];          // Dynamic memory allocation of temp array

  MPI_Win win;                                    // Creating window for global mass
  MPI_Win_create(massG,mnc*sizeof(double),sizeof(double),MPI_INFO_NULL,MPI_COMM_WORLD,&win);
  MPI_Win_fence(0,win);

  for(int i=0;i<npes;i++)
    {
    for(int j=0;j<mnc;j++)
      {temp[j]=0;}

    if(npes==1)                                    // Calculating nnc for each processor
      {cnnc=nn;}
    else
      {
```

```
        if((mnc*i)>(nn-mnc))
          {cnnc=max(0,nn-mnc*i);}
        else
          {cnnc=mnc;}
        }

      for(int l=0;l<nnl;l++)
        {
        nodeLToG=mesh->getNodeLToG()[l];        // Storing global node number of corresponding local node
        proc=nodeLToG/mnc;                      // Determining processor rank of the local node

        if(proc==i)                             // For current pe
          {
          in=(nodeLToG)%mnc;                    // Determining index
          temp[in]=mesh->getLNode(l)->getMass();
          }
        }

      MPI_Win_fence(0,win);                     // Value in temp is added for each processor
      MPI_Accumulate(temp,cnnc,MPI_DOUBLE,i,0,cnnc,MPI_DOUBLE,MPI_SUM,win);
      MPI_Win_fence(0,win);
      }
    free(temp);

    return;
}
```

Code excerpt 3: Excerpt from accumulateMass()

Two arrays are created, temp of datatype double and size mnc, and nodeLToG of datatype int and size nnl. The array nodeLToG stores the global node numbers, thus it should be of size nnl since number of nodes associated with the elements in a processor is nnl as well. massG can be accessed by all processors. MPI_Accumulate is used to add the data in temp into massG and then temp memory is freed.

MPI_Accumulate is used instead of MPI_Put. MPI_Put overwrites the data already present in the array since it always accesses the array memory from index 0. This is not desirable. Whereas MPI_Accumulates adds data without overwriting any previous entries of the array. It basically combines the already present data in array with new data. This is desirable, since we need contribution from all processors.

```
void femSolver::localizeTemperature(MPI_Win win)
{
  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

  double *TL=mesh->getTL();

  int mnc=mesh->getMnc();
  int nn=mesh->getNn();
  int nnl=mesh->getNnl();

  int cnnc,proc,in;

  double *temp;
  temp=new double[mnc*sizeof(double)];          // Dynamic memory allocation of temp array

  int nodeLToG;                                 // Stores global node number

  for(int i=0;i<npes;i++)
```

```
    {
    for(int j=0;j<mnc;j++)
      {temp[j]=0;}

    if(npes==1)                                    // Determining nnc for each pe
      {cnnc=nn;}
    else
      {
      if((mnc*i)>(nn-mnc))
        {cnnc=max(0,nn-mnc*i);}
      else
        {cnnc=mnc;}
      }

    MPI_Win_fence(0,win);                          // temp stores the values of TG
    MPI_Get(temp,cnnc,MPI_DOUBLE,i,0,cnnc,MPI_DOUBLE,win);
    MPI_Win_fence(0,win);

    for(int l=0;l<nnl;l++)
      {
      nodeLToG=mesh->getNodeLToG()[l];      // Storing global node number of corresponding local node
      proc=(nodeLToG)/mnc;                  // Determining processor rank for the local node

      if(proc==i)                           // For current pe
        {
        in=(nodeLToG)%mnc;                  // Determining index
        TL[l]=temp[in];
        }
      }
    }
  free(temp);

  return;
}
```

Code excerpt 4: Excerpt from localizeTempature()

In this function, the initial data initialization and data gathering part is similar to the previously explained function accumulateMass(). However, in the end, MPI_Get routine is used to get the global temperature data transferred into temp which is then transferred into local temperature array when global node of receiving processor is same with current loop iteration.

```
void femSolver::accumulateMTnew(MPI_Win win)
{
  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

  double *MTnewL=mesh->getMTnewL();

  int mnc=mesh->getMnc();
  int nnc=mesh->getNnc();
  int nn=mesh->getNn();
  int nnl=mesh->getNnl();

  int cnnc, proc,in;

  double *temp;
  temp=new double[mnc*sizeof(double)];          // Dynamic memory allocation of temp array

  int nodeLToG;                                 // Stores global node number

  for(int i=0;i<npes;i++)
    {
```

```
    for(int j=0;j<mnc;j++)
      {temp[j]=0;}

    if(npes==1)                                          // Determining nnc for each pe
      {cnnc=nn;}
    else
      {
      if((mnc*i)>(nn-mnc))
        {cnnc=max(0,nn-mnc*i);}
      else
        {cnnc=mnc;}
      }
    for(int l=0;l<nnl;l++)
      {
      nodeLToG=mesh->getNodeLToG()[l];          // Storing global node number of corresponding local node
      proc=(nodeLToG)/mnc;                      // Determining processor rank for the local node

      if(proc==i)
        {
        in=(nodeLToG)%mnc;                     // Determining index
        temp[in]=MTnewL[l];
        }
      }

    MPI_Win_fence(0,win);                              // Value in temp is added for each pe
    MPI_Accumulate(temp,cnnc,MPI_DOUBLE,i,0,cnnc,MPI_DOUBLE,MPI_SUM,win);
    MPI_Win_fence(0,win);
    }
  free(temp);

  return;
}
```

Code excerpt 5: Excerpt from accumulateMTnew()

In this function, data is transferred from MTnewL to MTnewG. In temp, we have data from MTnewL and accumulate it with MTnewG data. The logic used to perform this operation is similar to what had been discussed above.

After adding essential instruction in the required functions, we have to implement suitable communication routine in the explicitSolver(). Necessary code excerpts are provided below showing the implementation of MPI communication.

```
// Complete MPI Communication here

MPI_Win winMTnew;          // Window for MTnew
MPI_Win winTG;             // Window for TG

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);
```

Code excerpt 6: Excerpt from explicitSolver() for communication

Two windows i.e. winMTnew and winTG are created with MTnewG and TG of size mnc each. mnc is obtained from the helper function mesh->getMnc().

```
// Add MPI Communication here

localizeTemperature(winTG);    // To get TL
```

Code excerpt 7: Excerpt from explicitSolver() for communication

localizeTemperature(winTG) function is called with passing winTG as an argument since we need local temperature data to determine TL which in turn is needed to get values of MTnewL.

```
// Add MPI Communication here

accumulateMTnew(winMTnew);        // To get MTnewG
```

Code excerpt 8: Excerpt from explicitSolver() for communication

Since MTnewL is already determined, it has to accumulated with MTnewG. This is done by calling function accumulateMTnew(winMTnew) while winMTnew is passed as an argument.

```
// Add MPI Communication here

localizeTemperature(winTG);       // To get TL
```

Code excerpt 9: Excerpt from explicitSolver() for communication

Again localizeTemperature(winTG) is called to get values of TL for next iteration.

```
// Add MPI Communication here

MPI_Win_free(&winMTnew);      // To free the created window
MPI_Win_free(&winTG);
```

Code excerpt 10: Excerpt from explicitSolver() for communication

In the end, all windows are freed from memory space.

4 Results

The runtimes for all coarse and fine mesh are recorded till convergence while varying number of CPUs. For finest mesh, runtime was noted only till 1001 iterations to get speedup and efficiency.

	Runtime		
No of CPUs	**Coarse mesh** (till convergence)	**Fine mesh** (till convergence)	**Finest mesh** (till 1001 iterations)
	(seconds)	**(seconds)**	**(seconds)**
1	1.28811	772.24621	155.96905
2	1.35204	572.32525	93.62658
4	2.95156	372.41635	68.54197
8	7.82233	493.9096	75.86980
16	14.4823	664.52085	93.53615
32	53.83076	1117.12460	159.06960

Table 1: Effect of number of CPUs on runtime

For 4 number of CPUs, the runtime is least for fine mesh. The reason for this is, if fewer CPUs are used, proper MPI communication won't happen whereas if number of CPUs are increased, the overhead generated for communication will be too large and thus it will adversely affect the runtime.

For comparing the correctness of the implementation, this parallelize solution is compared with serial solution. The serial solution was obtained in project one.

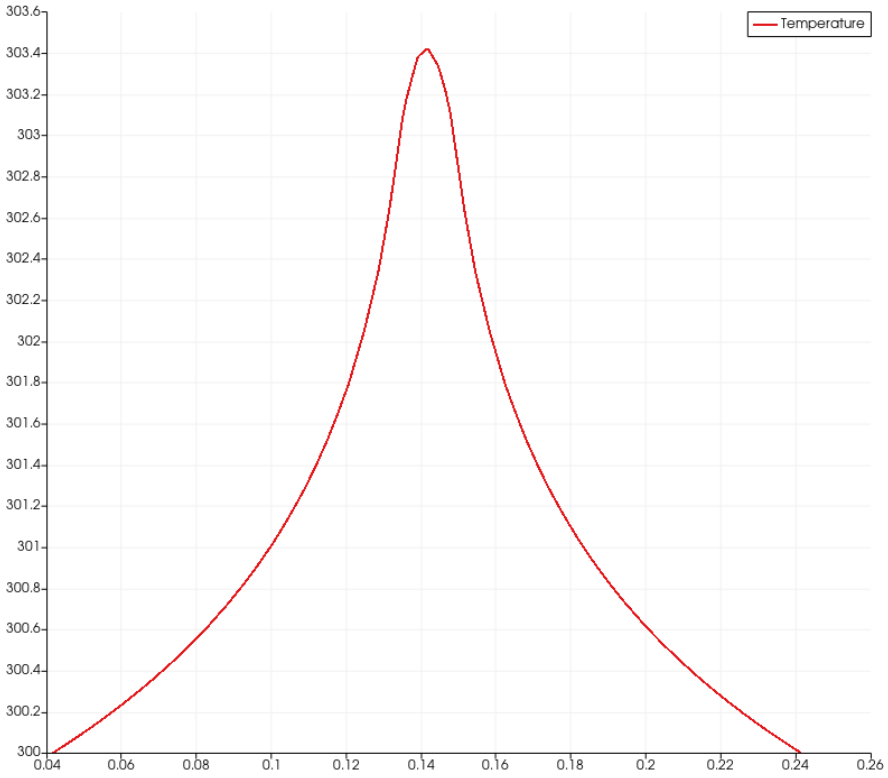

Figure 1: Temperature distribution for parallelized code (coarse mesh)

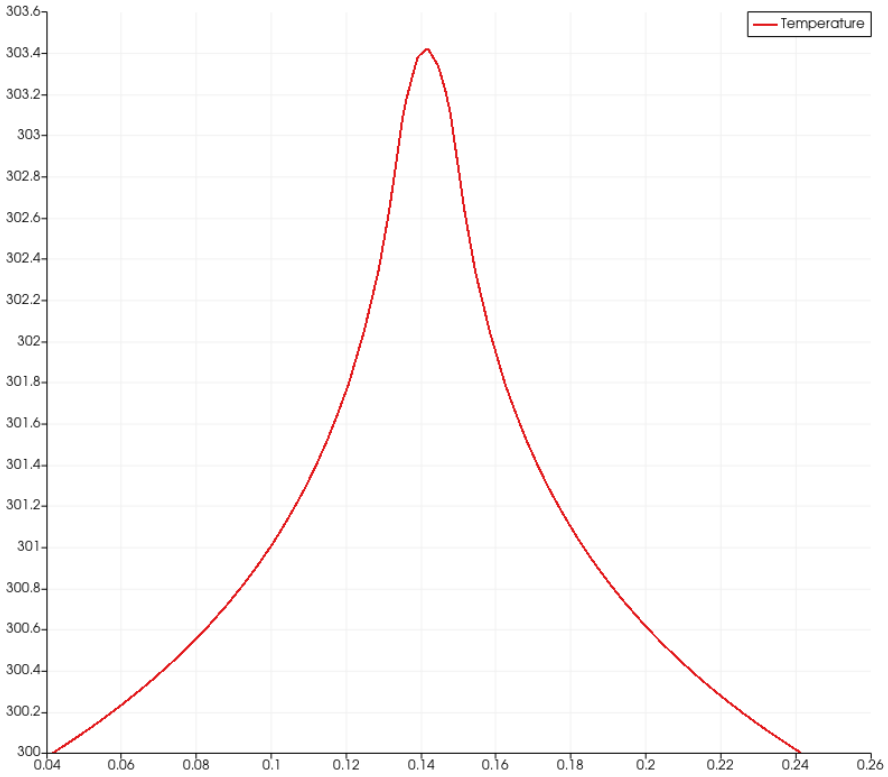

Figure 2: Temperature distribution for serial code (coarse mesh)

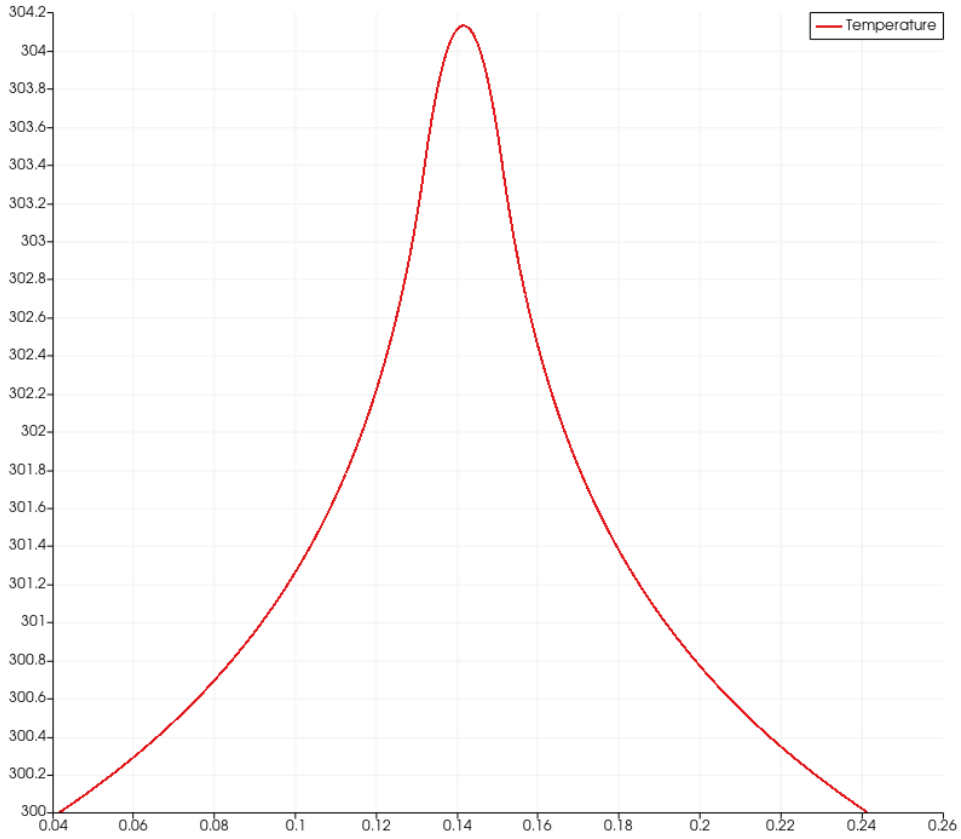

Figure 3: Temperature distribution for parallelized code (fine mesh)

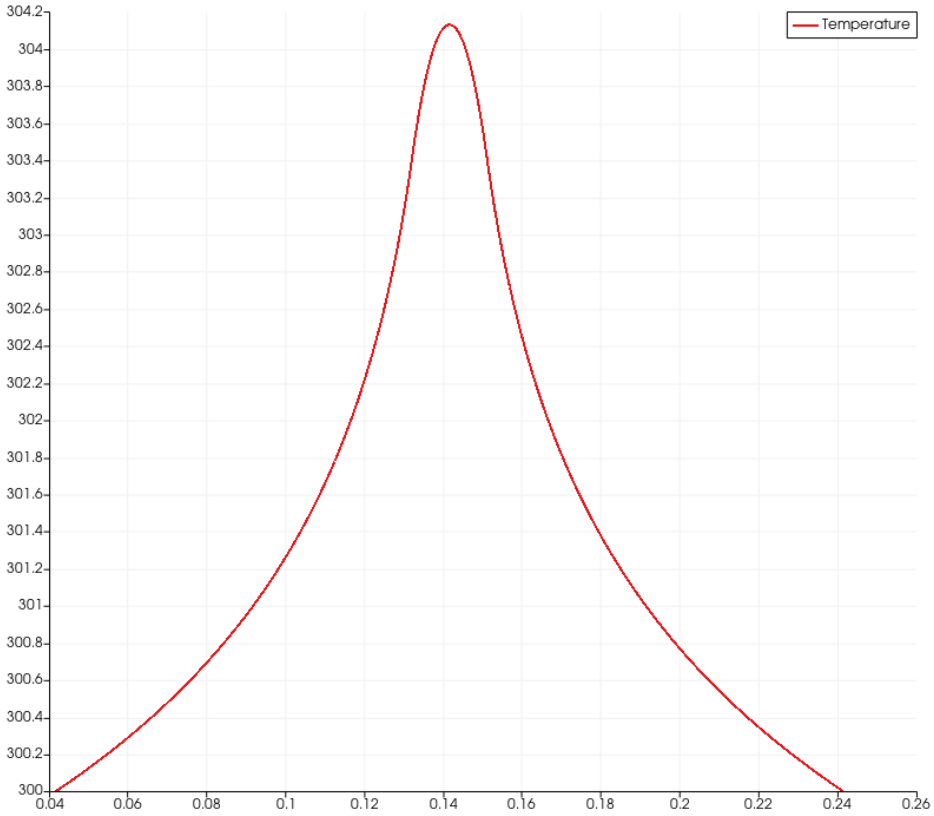

Figure 4: Temperature distribution for serial code (fine mesh)

The same concept can be extended to finest mesh as well. It is seen that temperature distribution is same, thus implementation of code is correct.

To show that solution stays unaltered for any number of cores, line plot of temperature distribution is shown for four cores.

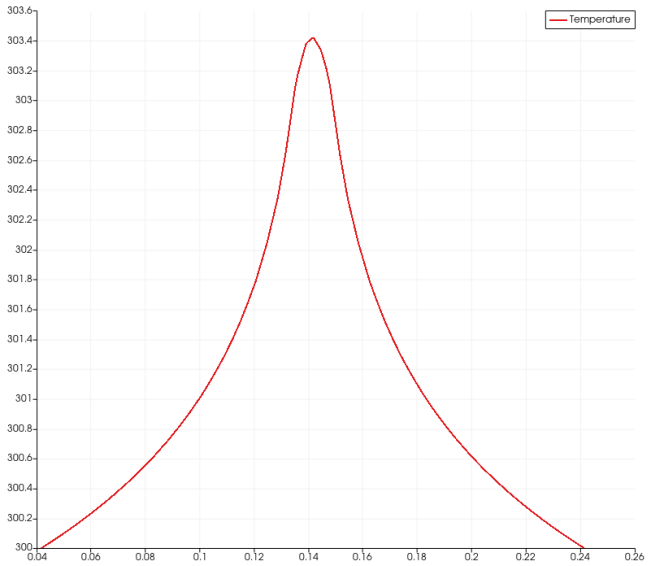

Figure 5: Temp. distribution (coarse mesh-2 cores)

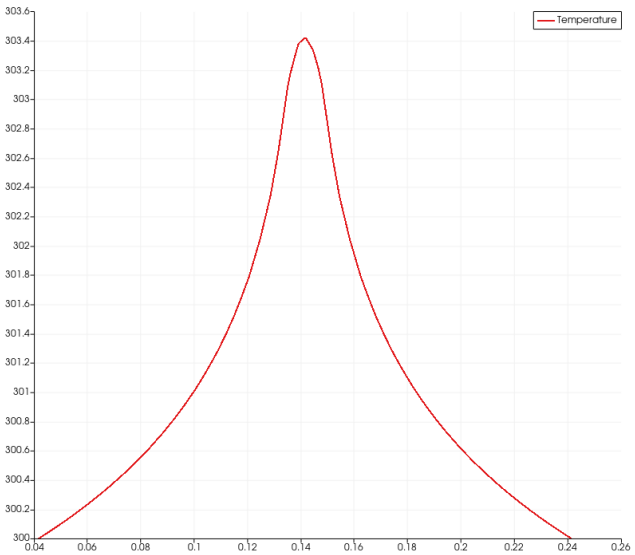

Figure 6: Temp. distribution (coarse mesh-4 cores)

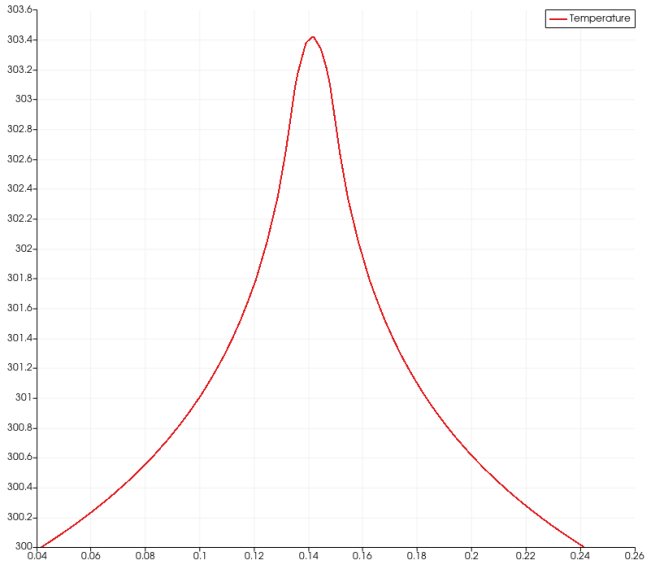

Figure 7: Temp. distribution (coarse mesh-8 cores)

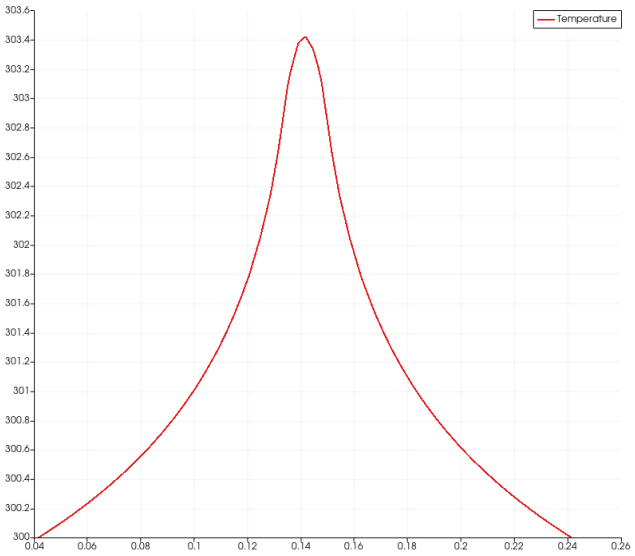

Figure 8: Temp. distribution (coarse mesh-16 cores)

It can be seen that, temperature distribution for coarse mesh is remains unaltered for 2, 4, 8, and 16 cores.

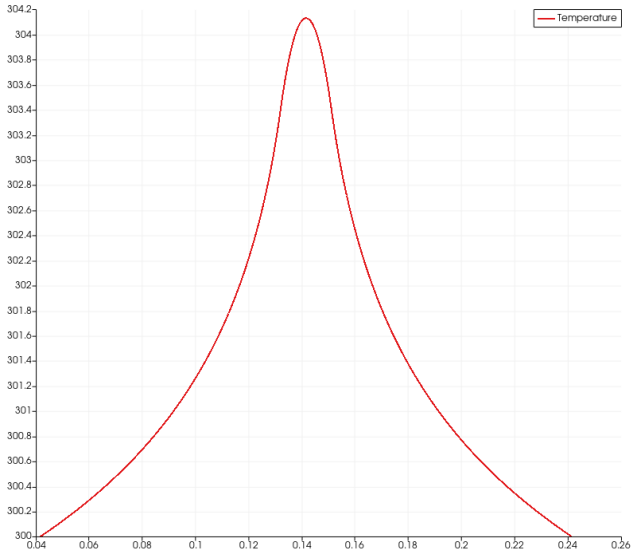

Figure 9: Temp. distribution (fine mesh-2 cores)

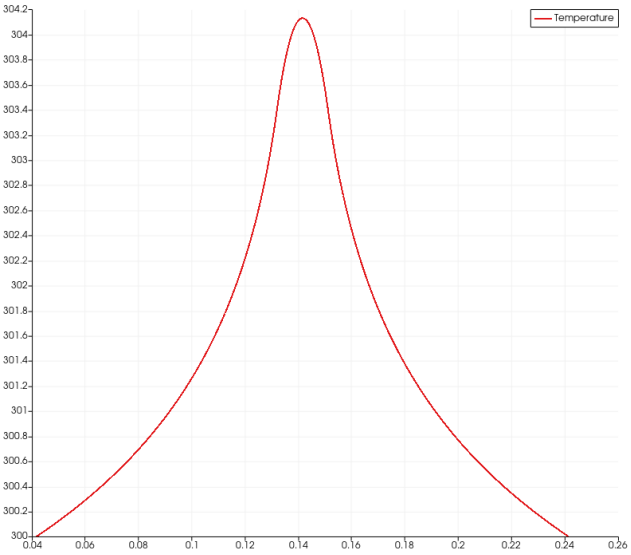

Figure 10: Temp. distribution (fine mesh-4 cores)

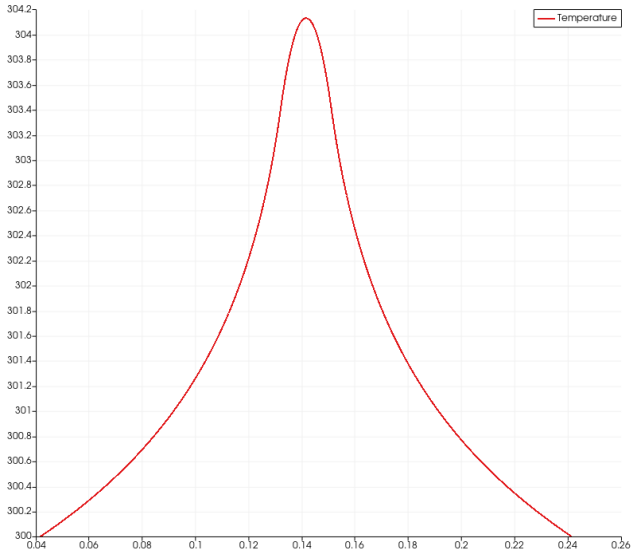

Figure 11: Temp. distribution (fine mesh-8 cores)

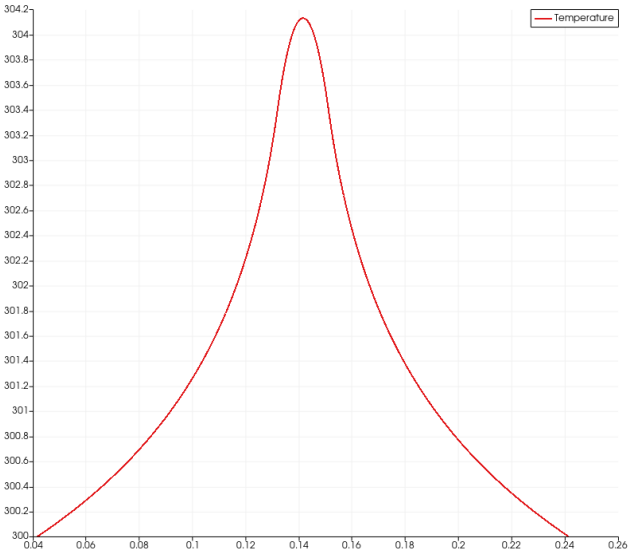

Figure 12: Temp. distribution (fine mesh-16 cores)

Similarly, for fine mesh as well, the temperature distribution remains unaltered for 2, 4, 8, and 16 cores.

Further to prove this, we can plot the RMS error value for coarse and fine mesh varying cores.

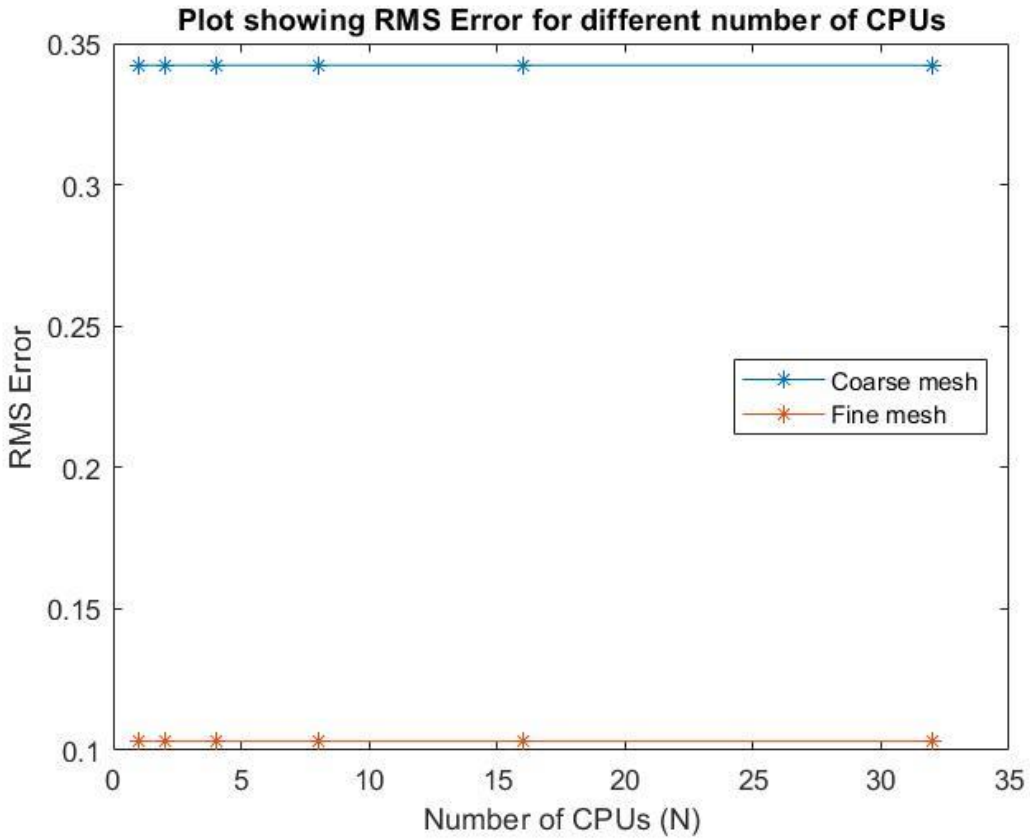

Figure 13: RMS error vs Number of CPUs for coarse and fine meshes

This figure shows that RMS error remains same even if the number of CPUs is varied.

Next task was to determine the speedup and efficiency analysis of the code. The runtime is provided in the table 1 above.

Speedup and efficiency are calculated for each case and tabulated using the following formulas:

$$Speed\ up\ (S) = \frac{T_s}{T_p}$$

$$Efficiency\ (E) = \frac{S}{P}$$

Where, Ts = Runtime with one processor

Tp = Runtime with multiple processors

P = Number of CPUs

The speedup and efficiency are calculated and noted down in the tables.

	Speedup (S)		
No of CPUs	**Coarse mesh**	**Fine mesh**	**Finest mesh**
1	1	1	1
2	0.9527	1.3493	1.6658
4	0.4364	2.0736	2.2755
8	0.1646	1.5635	2.0557
16	0.08894	1.1621	1.6674
32	0.02393	0.69128	0.9805

Table 2: Speedup values for all meshes

	Efficiency (E)		
No of CPUs	**Coarse mesh**	**Fine mesh**	**Finest mesh**
1	1	1	1
2	0.4763	0.6746	0.8329
4	0.1091	0.5184	0.5688
8	0.0205	0.1954	0.2569
16	0.0055	0.0726	0.1042
32	0.0007	0.0216	0.0306

Table 3: Efficiency values for all meshes

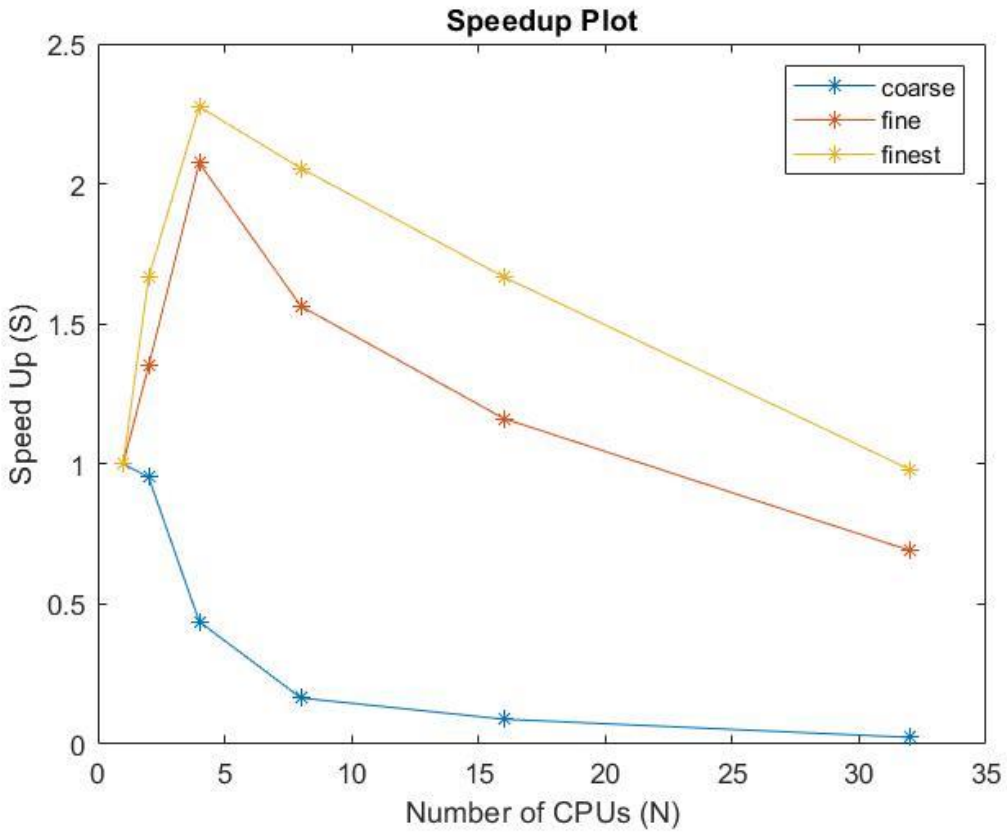

Figure 14: Speed up plot

Speedup can be seen to follow a decreasing trend after a very short increasing trend. This is due to increase in overhead that accompanies with increase in CPUs. The short increasing trend was seen due to less overhead when number of processors is 4. Each processor has its parallel overhead. When number of processors increases, the total parallel overhead increases to a very large magnitude which in turns decreases speedup.

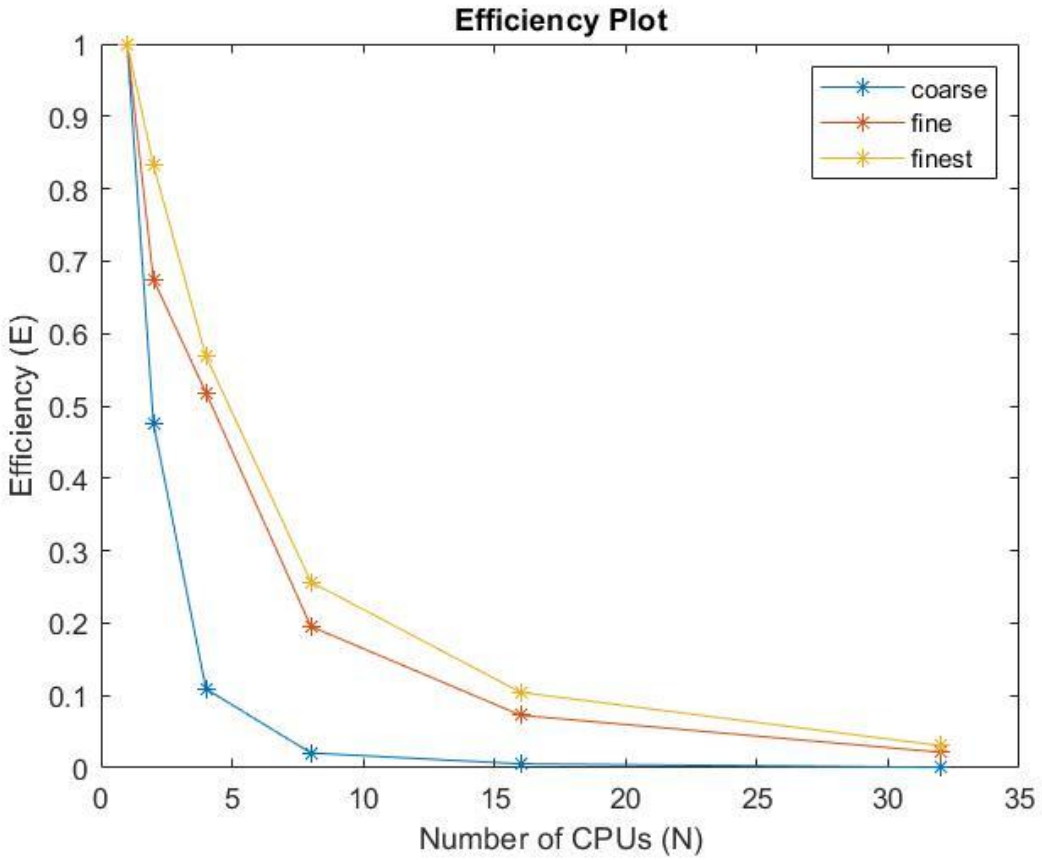

Figure 15: Efficiency plot

Efficiency is following a decreasing trend with increase in number of CPUs. This is since, efficiency is directly dependent on the speedup. As seen before, speedup in decreasing with increase in number of CPUs, thus efficiency also decreases. In other words, the parallel overheads of the processors are responsible for the decrease in efficiency when processor count increases.

In order to visualize the communication pattern, code is instrumented to gather trace information using SCALASCA and VAMPIR. Color coding is used to highlight the steps of communication and computation. The screenshot from VAMPIR is attached below. This graphics is generated for just one iteration.

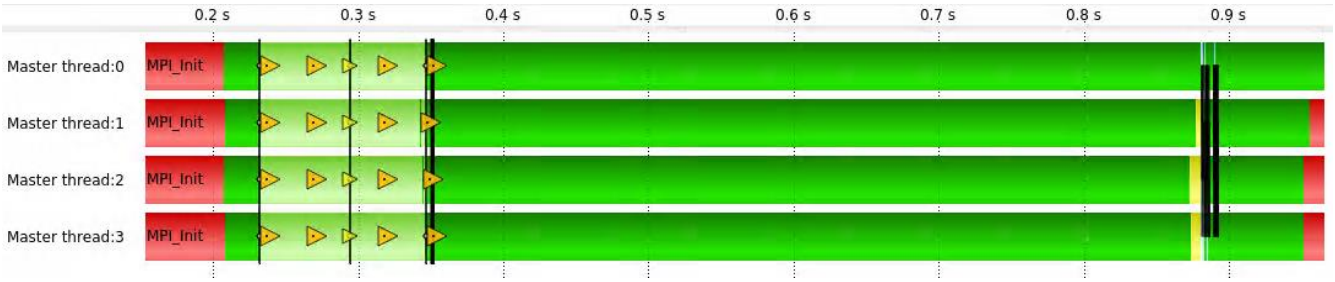

Figure 16: Timeline of communication pattern

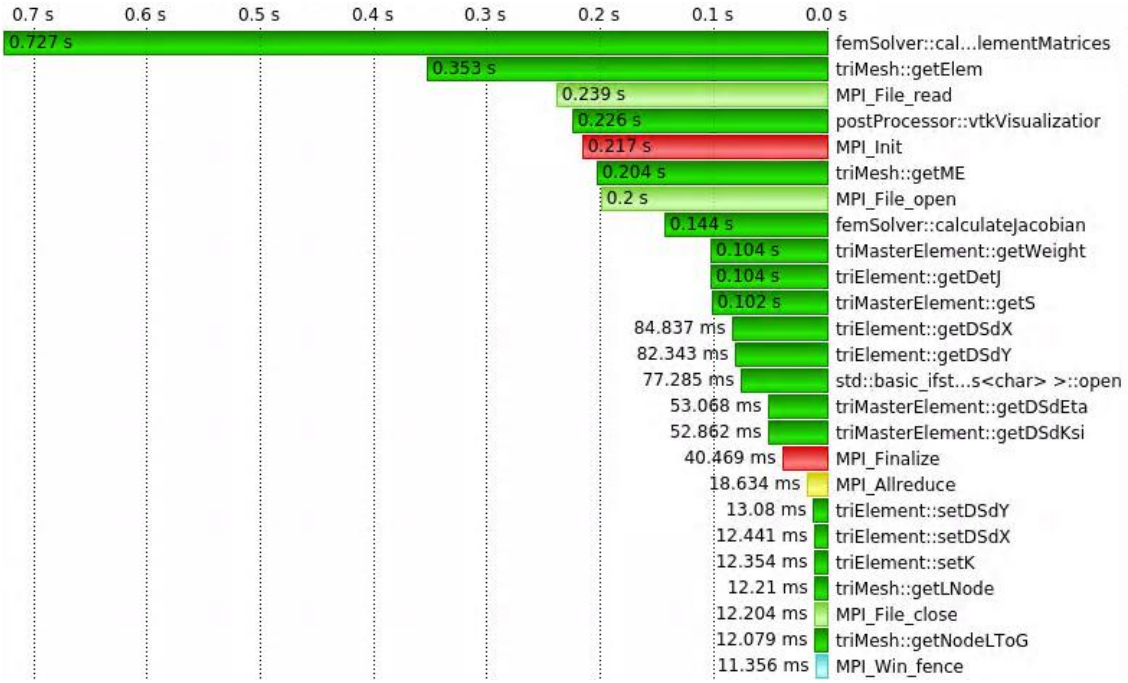

Figure 17: Time taken by processes

Figure 18: Legend

The graphics is divided in multiple time intervals to visualize better.

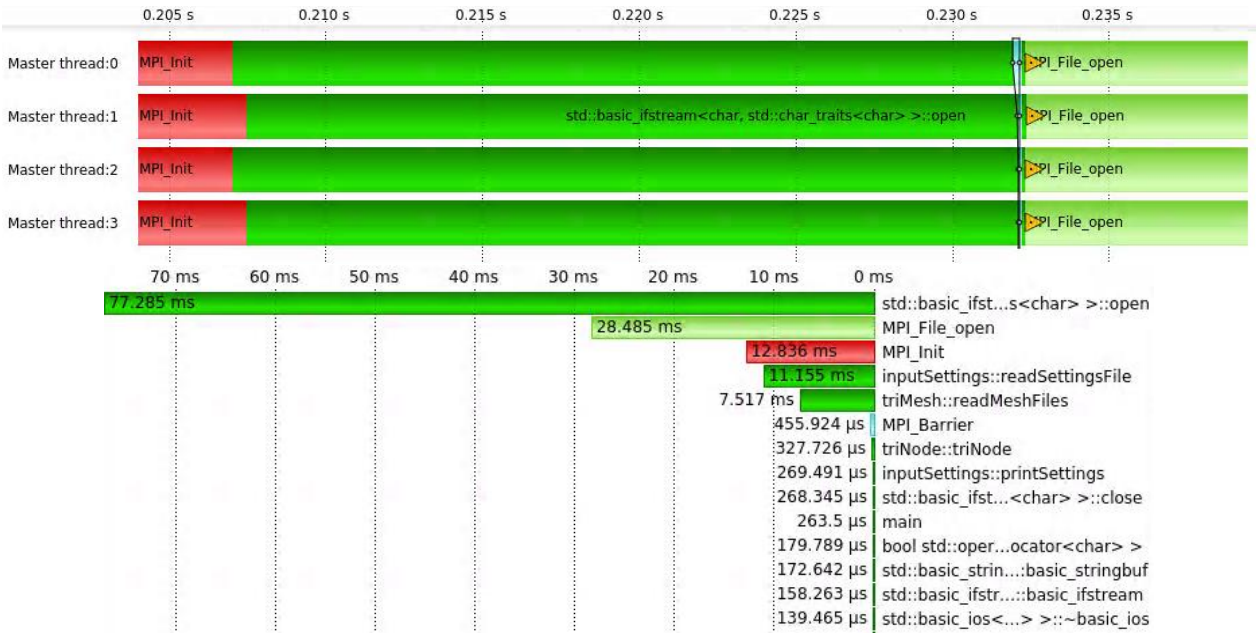

Figure 19: Communication timeline for interval (0.205s to 0.235s)

MPI environment is initialized first and file is then opened after synchronization using MPI_Barrier.

Figure 20: Communication timeline for interval (0.23s to 0.29s)

The opened file is then read and closed followed by synchronization using barrier routine. Every time a file is needed to be opened, it follows the synchronization routine.

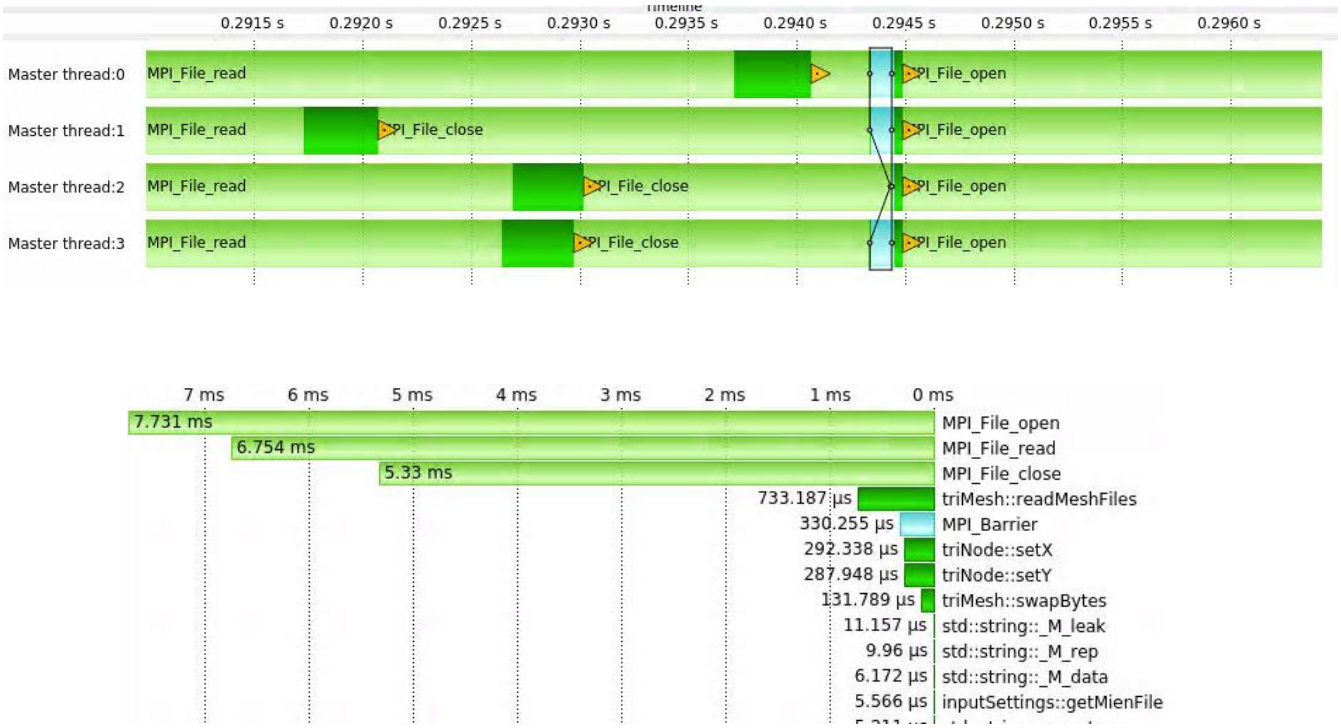

Figure 21: Communication timeline for interval (0.2915s to 0.2960s)

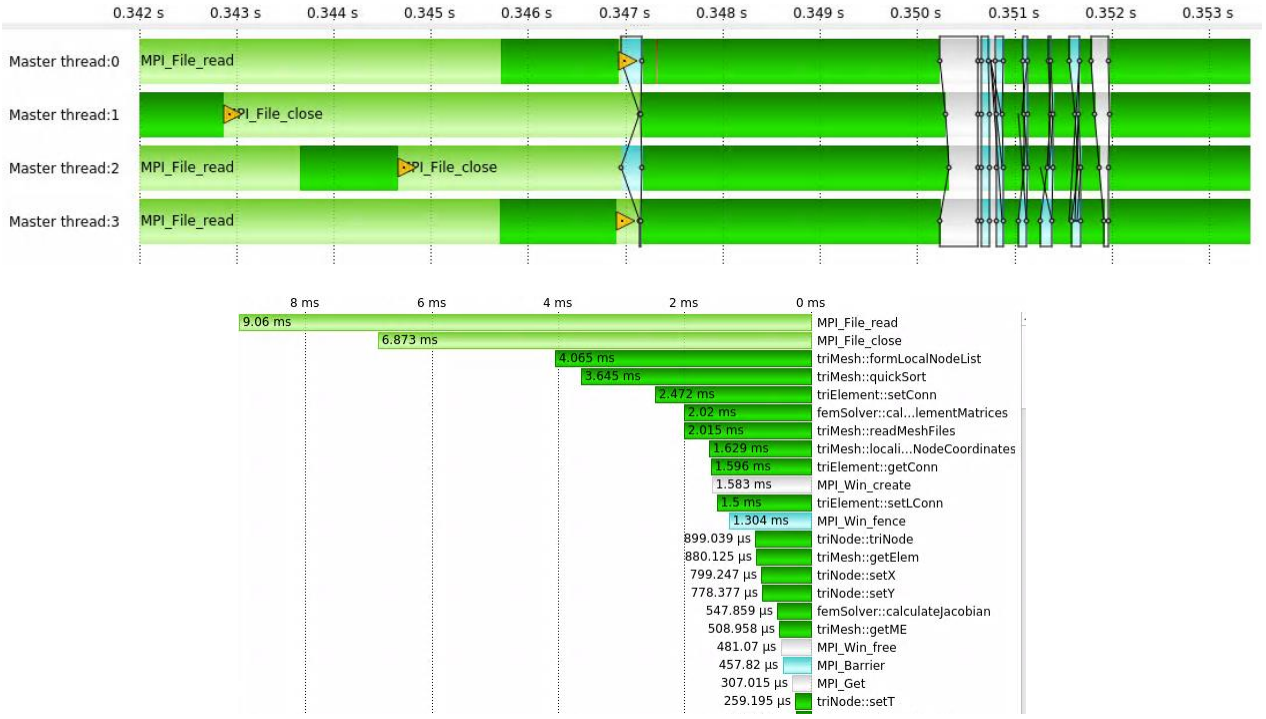

Figure 22: Communication timeline for interval (0.342s to 0.353s)

Window creation is followed by fence routine to enable effective synchronization. Meanwhile barrier routine is used along with MPI_Get.

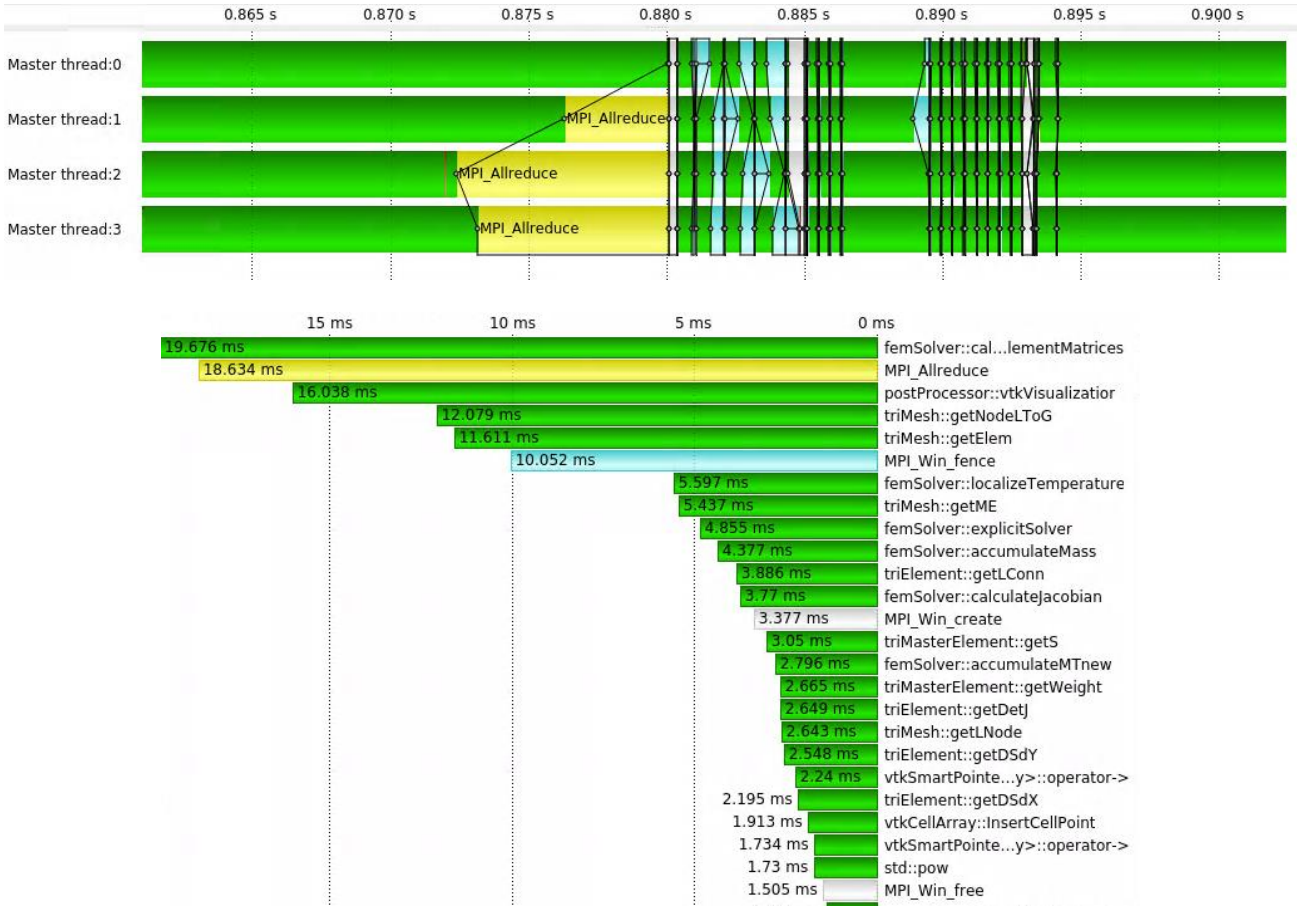

Figure 23: Communication timeline for interval (0.865s to 0.9s)

MPI_Allreduce combines values from all processors and distributes the result back to all processors. This is followed by MPI one sided communication and synchronization routines.

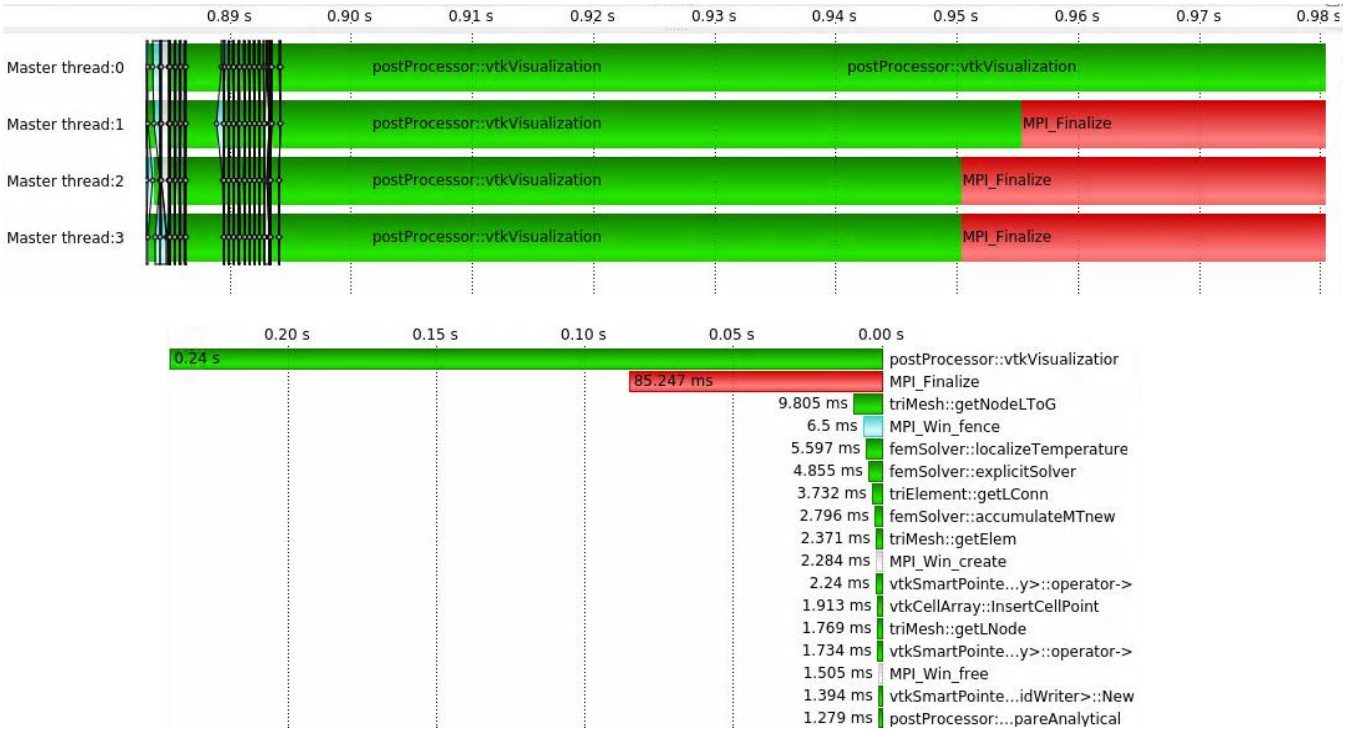

Figure 24: Communication timeline for interval (0.89s to 0.98s)

We have selected the coarse mesh to obtain the VAMPIR generated graphics and ran it using 4 cores. The four master threads in the picture above represent the four cores used.

Now, a question might arise: how can I further improve the code?

The code implementation can further be improved if we you both OpenMP and MPI routine together instead of using them separately. We have seen that in selected circumstances, one-sided communication offers several advantages, but overall using two-sided communication will improve the code. Most of the MPI routines that we used are from MPI-2 standard except concept of windows, which is from MPI-3 standard. To further improve the code, more routines from MPI-3 standard should be implemented, concept of RMA to be specific.

5 Conclusion

The MPI routines were implemented and solver was parallelized. The parallelization results in decrease in run time but since overall parallel overheads increased, the speedup decreased with increasing number of cores. The parallelized solution is same as serial solution and has very little deviation from analytical solution. Even though, the number of cores increased, the solution remained unaltered. Due to very large overhead, the speedup and efficiency followed a decreasing trend.

From VAMPIR, for 4 threads, complex communication pattern takes place. All threads communicate with each other. Using MPI_Win_fence and MPI_Barrier routines synchronized the communication very effectively. Out of all the MPI routines used, maximum time was taken by MPI_Finalize, followed by MPI_Allreduce. MPI communication routine with least time taken is MPI_Accumulate, followed by MPI_Get. However, maximum and majority of the time was taken by MPI_Init, followed by calculateElementMatrices(int) function.

During the series of three projects, we learnt about coding the finite element solver, and then improving it further. The main methods used to improve the code were implementing serial and parallel optimizations. In serial optimization, multiple compiler flags were added. Code was later parallelized using both OpenMP and MPI routines separately. Code timings can be improved if we use two-sided communication instead of one-sided communication. Further code timings can be improved if we use both OpenMP and MPI routines together.

References

[1] https://crd.lbl.gov/assets/Uploads/FTG/Projects/DEGAS/RetreatSummer13/DEGAS-retreat-Ibrahim.pdf

[2] http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture34.pdf

[3] https://www.mpich.org/static/docs/latest/www3/MPI_Allreduce.html

[4] https://www.mcs.anl.gov/uploads/cels/papers/P1731.pdf

[5] https://www.mcs.anl.gov/~thakur/papers/onesided-iba.pdf

[6] https://warwick.ac.uk/research/rtp/sc/rse/training/advancedmpi/05_one_sided_.pdf

[7] http://hpac.rwth-aachen.de/teaching/pp-18/Hoefler.pdf

[8] https://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html#pgfId-999136

[9] https://stackoverflow.com/questions/9799087/where-is-the-point-at-which-adding-additional-cores-or-cpus-doesn-t-improve-the