High Performance Optimizations and Dynamic Load Balancing for Computational Aerodynamics Solvers

Robin R
Robinraju@cet.ac.in
Shilpa S
shilpasunil997@gmail.com
Blessy Boby
godbless@cet.ac.in
Gmon k
gmonk@cet.ac.in

S8 CSE,
Department of Computer Science and Engineering,
College Of Engineering Trivandrum

Load Balancing[13] is a challenging and interesting problem that appear in all High Performance Computing(HPC)[11] environments. Most of the applications running in HPC environments use MPI [10] based parallel computing. This approach needs load assignment, before starting the computations, which is difficult in many problems where computational load changes dynamically. In VSSC Aeronautics entity, parallel computing is used to solve Computational Fluid Dynamics problems. Memory bandwidth utilization of CFD [12](Computational fluid dynamics is a branch of fluid mechanics that uses numerical analysis and data structures to solve and analyze problems that involve fluid flows.) problems are very low, making them computationally inefficient.

1 Proposed System

We are provided with mesh file[15] represented in VTK format[14] as initial input. In order to use MPI communication[10], we need to convert this mesh file into graph file which is then partitioned using METIS[9]. And this partitioned graph file is further applied to centralized, peer-to-peer and greedy approach and its performance comparison is noted.

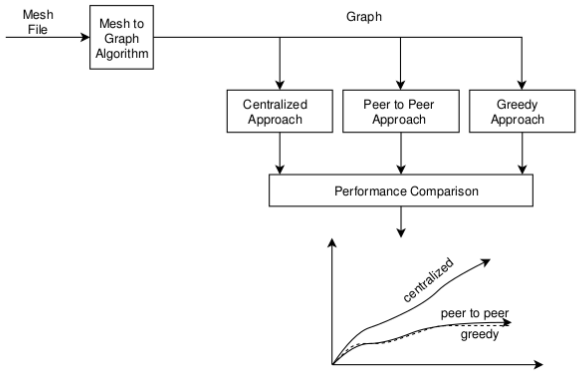

Figure 1: Different Phases of Project

1.0.1 Algorithm to convert mesh file into graph file

1.0.2 Algorithm-I

We are initially provided with a mesh file which contain information about the element used to represent the given input. We have to convert this mesh into graph file represented using adjacency list. Each element in the mesh file will become a vertex in the graph file. For finding whether two vertices are neighbours (connected by an edge) we have to check whether two elements are neighbours. In this algorithm we try to check whether two elements share any common face between them. If so then we construct an edge between them in the graph file using adjacency list.

In order to find the neighbouring vertices the algorithm proceed in following way. It first construct an element to node mapping. The element to node mapping contain the information about the nodes that constituting an element. We have the mesh file input in the standard vtk format. For this format each element has an unique id. For example if the element is

Algorithm 1 Algorithm to convert mesh to graph file-I

```
1. Read Mesh file in .su2 format
2. Check the dimension of the input
3. Store the number of vertices
4. For each element in the mesh file
   do the following
   4.1 Check the type of element
   4.2 Make an element to node mapping
   4.3 Make a node to element mapping
5. From the node to element mapping
   do the following.
   5.1 For each face for an element
       do the following
     5.1.1 Make an intersection among
           all the node to element
           mappings in the current
           face.
     5.1.2 If the cardinality of the
           resultant set is >= 2 then
           create adjacency list
6. Write the adjacency list to a file.
```

tetrahedron then the id is 10, for pyramid it is 14 etc. Consider a tetrahedron as example it contains the nodes 0,1,2,3. Then we make an element to node mapping as tetrahedron1 –> 0,1,2,3.

After the element to node mapping we create a node to element mapping as reverse to element to node mapping. This mapping contain the information about to which elements a node belongs. For example consider if there are two tetrahedron and first tetrahedron has the nodes 0,1,2,3 and the second has nodes 0,1,3,4. Then the corresponding node to element mapping will look like 0 –> tetrahedron1,tetrahedron2, 1 –> tetrahedron1,tetrahedron2, 2 –> tetrahedron1, 3 –> tetrahedron1,tetrahedron2, 4 –> tetrahedron2.

Now consider the faces of an element and take the intersection of the nodes to element mapping for the nodes constituting that face and if the cardinality of the result from the intersection is grater than or equal to 2 then they are neighbours and the corresponding neighbours are the elements of the resultant intersection.

For example consider the above node to element mapping here tetrahedron1 and tetrahedron2 are neighbours since they share a common face 0-1-3. Now take the face 0-2-3 of first tetrahedron, while taking intersection among them cardinality of the resultant set is not greater or equal to 2. But while considering the face 0-1-3 the resulting intersection set has a cardinality = 2 which satisfies our criteria and the corresponding neighbours are tetrahedron1 and tetrahedron2.

1.0.3 Algorithm-II

We are initially provided with a mesh file which contain information about the element used to represent the given input. We have to convert this mesh into graph file represented using adjacency list. Each element in the mesh file will become a vertex in the graph file. For finding whether two

Algorithm 2 Algorithm to convert mesh to graph file-II

```
1. Read Mesh file in .su2 format
2. Check the dimension of the input
3. Store the number of vertices
4. For each element in the mesh file
   do the following
   4.1 Check the type of element
   4.2 Make an element to node mapping
5. From the element to node mapping
   do the following.
   5.1 Find each face of an element
   5.2 For each face for an element
       do the following
       5.2.1 If this face is equal to
             another face of an element
             then make the face as edge
             and add this pair of elements
             to adjacency list.
       5.2.2 Store the face information.
6. Write the face information to file.
7. Write the adjacency list to a file.
```

vertices are neighbours (connected by an edge) we have to check whether two elements are neighbours. In this algorithm we try to check whether two elements share any common face between them. If so then we construct an edge between them in the graph file using adjacency list.

For this algorithm we developed a node to element mapping initially using some data structure. This mapping describes the nodes of an element. From this particular information and using the standards of the vtk format for the mesh file we can find which nodes contributing a face for a particular element. For example consider a tetrahedron in vtk format it consist a total of 4 nodes. Nodes 0-1-2 makes the first face, 1-2-3 makes second face, 0-2-3 makes the third face and finally 0-1-3 makes the fourth face.

By using this information about the faces, in this algorithm we are constructing faces for all the elements in the input file. So there will a face to element mapping for every element in the file. Then we are proceed to check whether two faces of any element is common which means they sharing a face and they are neighbours. If such a relationship exist between any two elements then we add them to our resultant graph as neighbours.

1.1 Communication

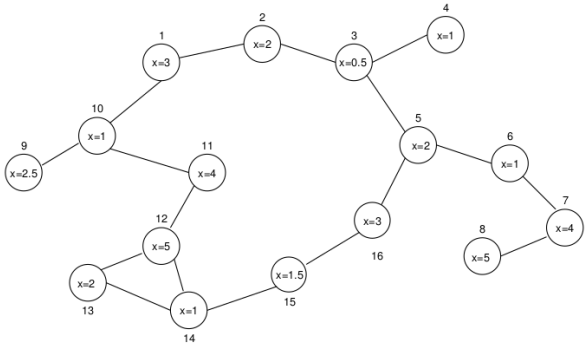

Figure 2: Sample graph

A sample graph and its partition is shown in the figure 2. Graph file consists of 16 vertices which is represented by the circle and value of x inside the circle indicates the data of the vertex. Graph is then partitioned into 4 cores i.e, vertices are distributed among cores. In this example, each core consists of 4 vertices of its own and 2 or more ghost vertices. Ghost vertices are represented by $\oplus$. Ghost vertices are vertices that contributes in the sub-graph of a core but it's ownership doesn't belong to that core. In this example, vertices 10 and 5 are the ghost of core 0 and contributes in its sub-graph but vertex 10's ownership belongs to core 2 and vertex 5's belongs to core 1.

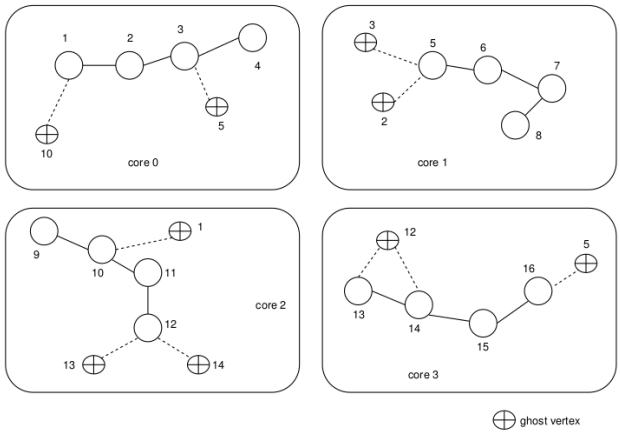

Figure 3: Partitioned graph

1.1.1 Centralized approach

In centralized approach, core 0 acts as master and other cores as slaves. Core 0 contains all the information about every vertices in the graph. Whenever, any core is in need of the details of any vertex, it is communicated with core 0.In order to balance the load dynamically between the cores, each core requires data of all vertices constituting it's subgraph. However, some of vertices are ghost and the data is only available to the owner of that vertices so each core request ghost vertex data from core 0 (owner of all vertices) through mpi call and core 0 sends the ghost vertex details through another mpi call.

Algorithm 3 Algorithm for centralized approach

```
1. Create a record structure for the vertex
   that contains
      1.1 Vertex number
      1.2 Vertex data
      1.3 Adjacency list
2. Initialize MPI communication environment
3. Do the following in Core 0.
      3.1 Read the graph file
      3.2 Read the partition file
      3.3 For each partition send the vertex
          corresponding to that partition
          one by one
4. For each core other than zero , do the
   following
      4.1 Construct the subgraph from the vertex
          sent by the zeroth core
      4.2 Find the ghost vertices from the subgraph
      4.3 Pack together the ghost vertices and send
          to the zeroth core.
5. On the zeroth core do the following
      5.1 find the data corresponding to each ghost
          vertices sent by each other partition
      5.2 send the data to corresponding partition
```

Vertex are actually represented in struct format and it consists of a adjacency list (array containing neighbours of the vertex). Initially, core 0 reads partition file and graph file parallely and checks the ownership of each vertex using partition file. Core 0 then sends object of each vertex to its corresponding core and using this object, we can access adjacency list

of the vertex. Core start to construct subgraph after all the vertices belonging to it's ownership arrived. Vertex data is available to both core 0 and the core in which it belongs. After constructing subgraph, cores checks whether data of all vertices are available. If any ghost vertex exists, core then requests its data from core 0 which then sends back corresponding data.

SHORTCOMING

While analysing and testing the above program, We came across some issues.

- This approach causes performance bottleneck at core 0. Since core 0 is the master in this approach, it is in charge of attending to all information transfer requests, it is overloaded and it become difficult to keep pace with rest of the system and thus lead to slowing overall performance.
- In this approach each vertex requires 2 MPI call,which in term needs that much amount of system call, which is so expensive and inefficient.
- In case of large graph file which may contain millions of vertices, zeroth core alone is insufficient in handling such large files. Even if it was possible for core 0 to handle such files, it may cause delay in the communication.
- Since the zeroth core acts as a master, all other cores have to wait until core 0 completes its task. So at this time other cores remains idle which is a wastage of time.

1.1.2 Peer-to-peer approach

Considering the shortcomings of centralised approach, we moved on to peer to peer approach. In this approach there is no centralised node. Each node is provided with subgraph of its own. Here, each core is parallely constructing subgraph of its own and hence, at that time amount of communication is zero therefore, cores does not suffers from idle state problem. Since there is no unnecessary transfer of information, the issue of bottleneck is resolved. Also because of less transfer, MPI communication is less frequent and thus it is no more expensive.

Algorithm 4 Algorithm for peer-to-peer approach

```
1. Define Vertex structure that contains.
   1.1 Vertex number
   1.2 Vertex data
   1.3 Adjacency list
   1.4 Local indexing list
2. Initialize MPI environment.
3. For each core do the following.
   3.1 Read the adjacency list
   3.2 Read the partition file
   3.3 Create subgraph on each core
   3.4 Create ghost list on each core
4. Create a local index for vertices on each
   core.
5. Prepare the ghost vertices for other cores
   as follows.
   5.1 Pack the ghost vertices for other cores
   into an array
   5.2 Send the ghost vertices as packed to
   other cores.
6. On the receiving side do the following.
   6.1 Receive the ghost vertices in packed
   manner
   6.2 Fill the data in the list corresponding
   to the ghost
   6.3 Send the filled data to the cores from
   which ghost received
7. Do the required calculations on the cores.
8. Finalize the MPI environment.
```

SHORTCOMING

While analysing this approach, we came across some issues

- Here we are using constant buffer and receiving core is not aware of size of the ghost so that it should allocate a constant buffer size which will lead to inefficient space allocation.
- Peer-to-peer approach may face deadlock in situations where all cores are sending mpi send to all other cores. In such conditions, mpi_send act as a blocking send and blocks communication between cores.
- Peer-to-peer approach may incur more delay in communication because in peer-to-peer, communicating cores are selected at random without considering any ordering.

1.1.3 Greedy approach

In this approach, Synchronization message is sent between all the cores. Hence each core will be aware of the size of ghost sent from other cores. For this, a matrix in which each cell representing the size of information send from i'th node to j'th node is created. From the matrix of communication load, communication with maximum load is selected and is added to slot 0 and repeat the process by excluding row and column of selected load.

Greedy approach incur less delay compared to peer-to-peer approach. That is in greedy approach, communicating cores are selected through maximum to minimum communication load ordering and these cores are then arranged in slots depending on their dependency. That is, each slot contains the maximum number of cores, which does not cause delay in the communication between any other cores in that slot.

Algorithm:

In greedy method, initially a synchronization message is send between each core. fig 3 shows that core 1 and core 2 requires data of vertices which are belonging to core 0 for constructing their sub-graph. That's why in fig 4 core 0 is sending 0110 to every other core indicating that it contains vertex that core 1 and core 2 are in needed.

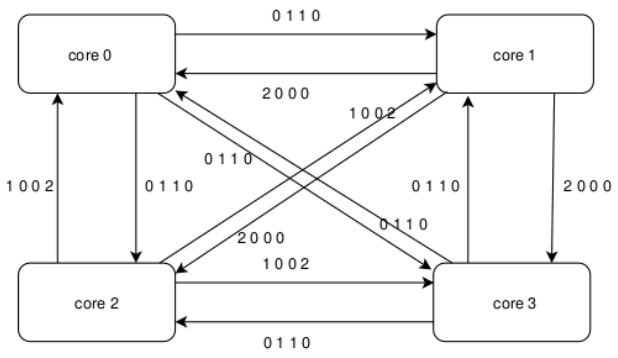

Figure 4: Sending synchronization message

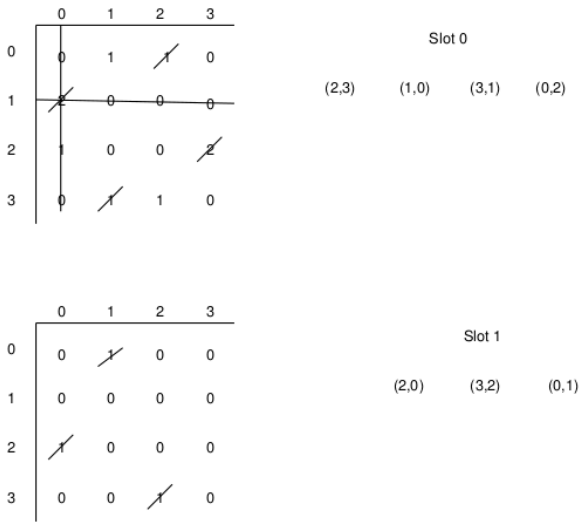

Figure 5: Process of taking maximum load from matrix and forming slots

Also, here before sending ghost data, communication load (number of ghost vertex send between each core) is arranged in matrix 5 and from that matrix different slots for communicating is derived i.e, from matrix maximum communication load is chosen and added in first slot and removed its corresponding rows and columns in further selection of load for slot 0. and this process repeats until there is no non-conflicting entry.

After completing slot 1, loads are selected for slot 2 through the same process. and this process of constructing slots repeats until all entries in the matrix becomes zero. Slots are represented by vector pairs that is each entry in slot is a pair of communicating cores. Cores are then communicated in order that is first pair in slot 0 is communicated first.

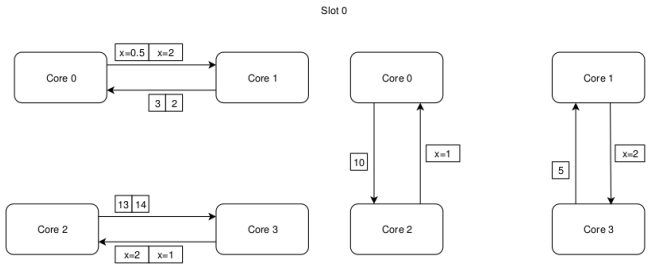

Figure 6: Representation of slot-0

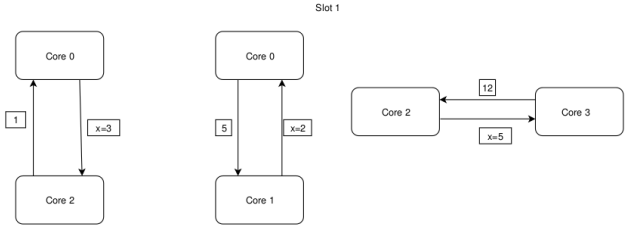

Figure 7: Representation of slot-1

Algorithm 5 Algorithm for greedy approach

1. Define Vertex structure that contains.
   - 1.1 Vertex number
   - 1.2 Vertex data
   - 1.3 Adjacency list
   - 1.4 Local indexing list
2. Initialize MPI environment.
3. For each core **do** the following.
   - 3.1 Read the adjacency list
   - 3.2 Read the partition file
   - 3.3 Create subgraph on each core
   - 3.4 Create ghost list on each core
4. Create a local index **for** vertices on each core.
5. Create and broadcast synchronization message from each core 'i' that contains,
   - 5.1 Size of the ghost from the core 'i' to **all** other cores.
6. For each core 'i' **do** the following
   - 6.1 Receive the synchronization message from **all** other cores.
   - 6.2 Construct a matrix 'synch' in **which** synch[i][j] contains **size** of the ghost to send from i'th core to j'th core.
7. Prepare the cores **for** communication as follows.
   - 7.1 Take the current largest entry in the 'synch' matrix and assign corresponding 'i' and 'j' to slot 0.
   - 7.2 Avoid the previously taken 'i' and 'j', now **find** next largest entry assign the corresponding 'j' and 'j' to slot 0.
   - 7.3 Continuing in ths way **fill** out slot 0 and then proceed to slot 1 and so on.
8. Initiate the communication.
   - 8.1 Pack the ghost vertices **for** the cores in slot 0 and initiate their communication.
   - 8.2 After completing **all** communication in slot 0 initiate the slot 1 and so on.
7. Do the required calculations on the cores.
9. Finalize the MPI environment.

2 Theoretical proof

2.1 Centralized approach

Let number of vertices be **n** and number of cores be **m**. Here, core 0 act as **master** and contains all details about every other vertex. Therefore for creating sub-graph, each vertex requires **2 mpi call.**

2.1.1 Worst case :

Assume that each core contain only 1 vertex and remaining all vertices are ghost for that vertex. In such condition, algorithm takes n-1 mpi send for requesting ghost data from 0th core and n-1 mpi receive for receiving ghost details from 0th core. In total there are m-1 cores, therefore amount of communication will be,

$$\mathbf{2n + 2(m\text{-}1) * (n\text{-}1)}$$

2.1.2 General case:

In cases other than worst cases, amount of computation will be proportional to the number of ghost for that core.

$$\mathbf{2n + 2 * \sum_{i=1}^{m-1} ki}$$

where ki is the number of ghost in ith core

2.2 Peer-to-peer and greedy approach

In these approaches, there is no centralised node. Each node is provided with sub-graph of its own. Here each core invokes at most 2 mpi call with every other core.

2.2.1 Worst case:

Each core has to communicate with all other cores. since there are **m** cores, total amount of communication will be

2(m-1) * m

3 Results

Through our experimental results and observations, we came across a result which is depicted in the following table

Table 1: Comparison between different communication approaches

	Total amount of communication	
	General Case	Worst Case
Centralized	$2n + 2*\sum_{i=1}^{m-1}$ ki	**2n + 2(m-1) * (n-1)**
Peertopeer		**2(m-1) * m**
Greedy		**2(m-1) * m**

That is, Greedy approach is more efficient in terms of computational time than peer-to-peer and centralized approach.

In centralized approach each vertex requires 2 MPI call, which in term needs that much amount of system call, which is so expensive and inefficient but in peer-to-peer and greedy approach, MPI communication is less frequent and thus these are no more expensive.

Peer-to-peer approach may incur more delay compared to greedy approach because in peer-to-peer, communicating cores are selected at random without considering any ordering. However in greedy approach, communicating cores are selected through maximum to minimum communication load ordering and these cores are then arranged in slots depending on their dependency.

4 Conclusion

DFS/BFS technique will not reduce the communication volume(represented as edgecuts in metis) which is why we used METIS here. but, still there is a proper alternative for METIS which is space filling curves [17]. in our input mesh file, vertex is more of a fluid mechanics computation and mass, energy and momentum fluxes will be computed on graph edges and also conserved fluid properties will be updated on each vertex looking at the fluxes on its edges. A close computation can be the one which appear in the paper [2][3].

MPI is not only the standard for communication. There are many other communication libraries like ARMCI [4], Portals [5] etc. which are used for special cases. But here we work on MPI as it was the library which was suggested to us.

Partitioned graph file is communicated through centralized, peer-to-peer and greedy approach and their results are compared and best approach is chosen. These approaches are implemented in C++ but it can also be implemented in charm++ [6] or PETsc[7] through dynamic programming [8].

5 REFERENCES

1. https://community.dur.ac.uk/suzanne.fielding/teaching/BLT/sec1.pdf
2. https://www.comsol.com/multiphysics/fluid-flow-conservation-of-momentum-mass-and-energy
3. https://wiki.mpich.org/armci-mpi/index.php/Main_Page
4. http://www.nwchem-sw.org/index.php/Release66:ARMCI
5. http://charm.cs.illinois.edu/research/charm
6. https://www.hindawi.com/journals/mpe/2015/147286/
7. Vassil Alexandrov, Ken Chan, Alan Gibbons, Wojciech Rytter (2005). On the PVM/MPI computations of dynamic programming recurrences. *springer*
8. George Karypis (2013). METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0
9. William Gropp. Tutorial on MPI: The Message-Passing Interface.
10. https://searchdatacenter.techtarget.com/definition/high-performance-computing-HPC.
11. https://www.simscale.com/docs/content/simwiki/cfd/whatiscfd.html.
12. https://www.nginx.com/resources/glossary/load-balancing/
13. https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf
14. https://su2code.github.io/docs/Mesh-File/
15. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
16. http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/.
17. *https://insidehpc.com/2006/06/what-are- alternatives-to-mpi*