

High Performance Optimizations and Dynamic Load Balancing for Computational Aerodynamics Solvers

A Project Report

submitted by

**ROBIN.R(LTVE15CS068), SHILPA.S(TVE15CS055), BLESSY
BOBY(TVE15CS020), GMON.K(TVE15CS024)**

to

*APJ Abdul Kalam Technological University
in partial fulfilment of the requirements
for the award of the degree of*

BACHELOR OF TECHNOLOGY



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING TRIVANDRUM**

May 2019

CERTIFICATE



This is to certify that the thesis entitled **High Performance Optimizations and Dynamic Load Balancing for Computational Aerodynamics Solvers**, submitted by **Robin.R(LTVE15CS068)**, **Shilpa.S(TVE15CS055)**, **Blessy Bobby(TVE15CS020)**, **Gmon.K(TVE15CS024)**, to the APJ Abdul Kalam Technological University, for the award of the degree of **Bachelors of Technology**, is a bona fide record of the project presented by them under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.

Dr. Ajeesh Ramanujan
Project Guide
Assistant Professor
Department of CSE
CET-695016

Harichand M.V
Project Guide
Scientist Engineer
VSSC-695022

Vipin Vasu A V
Project Coordinator
Associate Professor
Department of CSE
CET-695016

Dr. Salim.A
Head of Department
Professor
Department of CSE
CET-695016

DECLARATION

We hereby declare that the report of the project **High Performance Optimizations and Dynamic Load Balancing for Computational Aerodynamics Solvers**, is submitted to the Department of Computer Science and Engineering at College of Engineering Trivandrum affiliated to APJ Abdul Kalam Technological University, Kerala in partial fulfillment of the award of Bachelor of Technology in Computer Science and Engineering, is a bona fide work done by our team under the guidance of **Dr.Ajeesh Ramanujan**, Assistant Professor, Department of Computer Science and Engineering and **Harichand M V**, Scientist Engineer, Vikram Sarabhai Space Centre. Also we declare that matter embedded in this project report has not been submitted for the award of any degree of any other Institution or University.

Place: Trivandrum
Date:

Group Members

ACKNOWLEDGEMENTS

Firstly, We would like to thank the Almighty for giving me the wisdom and grace for implementing our project.

With a profound sense of gratitude, We would like to express our heartfelt thanks to our project guides **Dr.Ajeesh Ramanujan**, Assistant Professor, Department of Computer Science and Engineering and **Harichand M V**, Scientist Engineer, Vikram Sarabhai Space Centre for their expert guidance, co-operation and immense encouragement in pursuing this project.

We are very much thankful to **Dr. Salim A**, Head of Department and **Prof. Vipin Vasu A V**, Associate Professor, our guide for providing necessary facilities and his sincere co-operation.

Our sincere thanks is extended to all the teachers of the department of Computer Science and Engineering and to all our friends for their help and support.

Group Members

ABSTRACT

KEYWORDS: Load Balancing, High Performance Computing, Graph Partitioning, Message Passing Interface(MPI) Communication, Centralized, Peer to Peer, Greedy

The challenge of the project is to design a distributed **MPI**[\[13\]](#) framework for solving computational problems. On designing the framework, I have to ensure load balancing in parallel computing environment, as most of the applications running in High Performance Computing use **MPI** based parallel computing. The data consists of mesh files in *VTK* format, which was converted into graphs and partitioned in order to distribute them. So an algorithm to convert mesh file to graph file is created. Now, the partitioned graph file is distributed such that, all the cores of the system is load balanced and the communication between them is efficient. Firstly, an algorithm with centralized approach is created, in which, one of the core is set as the master. The algorithm involves many communications, thus making it expensive and inefficient. A different algorithm is created in a peer to peer approach. In this, all the cores plays equal roles. When compared to the number of communication involved in centralized approach, the peer to peer approach involves far less communication overhead. But, for certain inputs, deadlock follows. So a greedy approach was proposed to solve the problem of deadlock. Also, when communication delay is considered as a parameter of measure, this is the best approach. Thus, an efficient distributed **MPI** framework is designed.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
LIST OF ALGORITHMS	viii
ABBREVIATIONS	ix
1 INTRODUCTION	1
1.1 OBJECTIVE	2
1.2 SCOPE OF THE PROJECT	2
2 BACKGROUND	3
2.1 CONCEPTS	3
2.1.1 HIGH PERFORMANCE COMPUTING(HPC)	3
2.1.2 COMPUTATIONAL FLUID DYNAMICS (CFD)	4
2.1.3 MESSAGE PASSING INTERFACE(MPI)	5
2.2 REQUIREMENTS	5
2.2.1 MESH FILE	5
2.2.2 METIS	8
2.2.3 MPI	12
3 PROPOSED SYSTEM	16
3.1 ALGORITHM TO CONVERT MESH TO GRAPH	18
3.1.1 ALGORITHM-1	18

3.1.2	ALGORITHM-2	20
3.2	COMMUNICATION	22
3.2.1	CENTRALIZED APPROACH	22
3.2.2	PEER-TO-PEER APPROACH	25
3.2.3	GREEDY APPROACH	27
3.3	THEORETICAL PROOF	30
3.3.1	CENTRALIZED APPROACH	30
3.3.2	PEER-TO-PEER AND GREEDY APPROACH	31
4	RESULTS AND DISCUSSION	33
5	CONCLUSION	36
A	HOW TO RUN IN A CLUSTER	38

LIST OF TABLES

2.1	Connectivity and number of nodes of different elements	8
4.1	Comparison between different communication approaches based on total amount of communication	34
4.2	Comparison between greedy and peer to peer based on delay . .	35

LIST OF FIGURES

2.1	Vertex order of different elements in mesh file(VTK format)	6
2.2	Input file to metis - 1	10
2.3	Input file to metis - 2	10
2.4	Input file to metis - 3	11
2.5	Input file to metis - 4	11
3.1	Different Phases of Project	16
3.2	Geometric Representation of Cube	16
3.3	Sample graph	22
3.4	Partitioned graph	22
3.5	Sending vertex to corresponding core and constructing sub-graph in centralized approach	24
3.6	Requesting and sending of ghost data in centralized approach	24
3.7	Requesting ghost data in peer-to-peer approach	26
3.8	Sending ghost data in peer-to-peer approach	26
3.9	Deadlock situation in peer-to-peer approach	26
3.10	Sending synchronization message	28
3.11	Process of taking maximum load from matrix and forming slots	28
3.12	Representation of slot-0	30
3.13	Representation of slot-1	30

List of Algorithms

1	Algorithm to convert mesh to graph file-I	19
2	Algorithm to convert mesh to graph file-II	21
3	Algorithm for centralized approach	23
4	Algorithm for peer-to-peer approach	25
5	Algorithm for greedy approach	29

ABBREVIATIONS

VSSC	Vikram Sarabhai Space Centre
CET	College of Engineering Trivandrum
ISRO	Indian Space Research Organisation
HPC	High Performance Computing
CFD	Computational Fluid Dynamics
MPI	Message Passing Interface
DFS	Depth First Search
BFS	Breadth First Search

CHAPTER 1

INTRODUCTION

The motivation for doing this project was primarily an interest in undertaking a project at VSSC, ISRO. The opportunity to learn about a new area of computing not covered in lectures was appealing. Project is intended to develop a distributed **MPI** framework for solving computational problems efficiently. Challenge here is to design data layouts which can effectively utilize memory bandwidth. Multiple re-ordering schemes, along with data duplication can be attempted to improve the memory performance. While solving computational problems in a distributed **MPI** environment, it is important to ensure load balancing. But the static load balancing algorithms can fail if the computational loads vary over time. A distributed work queue may be attempted to solve this problem.

Load Balancing[1] is a challenging and interesting problem that appear in all High Performance Computing(HPC) [2] environments. Most of the applications running in HPC environments use **MPI** [14] 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 [3] (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.

1.1 OBJECTIVE

Two problems can be attempted as part of this Project

- Improve the dynamic load balancing capability in an **MPI** environment for solving CFD problems.
- Improve data locality of CFD problems.

1.2 SCOPE OF THE PROJECT

The scope of the project is to develop a frame work for to solving computational fluid dynamics Problems keeping in mind the following.

- Explore graph re-ordering, data layout modification for increased compute efficiency.
- Dynamic load balancing.
- Hybrid CPU-GPU computing. And finally, compare the results with default **MPI** method.

CHAPTER 2

BACKGROUND

This chapter includes different background concepts and requirements for the project.

2.1 CONCEPTS

2.1.1 HIGH PERFORMANCE COMPUTING(HPC)

High Performance Computing (HPC) [2] is the utilization of parallel handling for running propelled application programs effectively, dependably and rapidly. The term applies particularly to frameworks that work over a teraflop or 10¹² gliding point tasks for each second. The term HPC is every so often utilized as an equivalent word for supercomputing, albeit in fact a supercomputer is a framework that performs at or close to the at present most elevated operational rate for PCs. A few supercomputers work at more than a petaflop or 10¹⁵ coasting point tasks for each second.

The most widely recognized clients of HPC frameworks are logical scientists, architects and scholarly organizations. Some administration organizations, especially the military, likewise depend on HPC for complex applications. Superior frameworks frequently utilize hand crafted segments notwithstanding alleged product segments. As interest for handling force and speed develops, HPC will probably intrigue organizations everything being equal, especially for exchange

preparing and information stockrooms. An incidental techno-savages may utilize a HPC framework to fulfill an extraordinary want for trend setting innovation.

2.1.2 COMPUTATIONAL FLUID DYNAMICS (CFD)

Computational Fluid Dynamics (CFD) is a part of liquid mechanics that utilizes diverse calculations and numerical examination so as to dissect and take care of issues including liquid streams. The fundamental center is the utilization of PCs and information demonstrating so as to recreate and examine how a liquid streams regarding a surface. Certifiable applications incorporate examination of wind stream for a streamlined flying machine plan or investigation of the hydrodynamic properties of a vessel body, mechanical structure of oil and water funneling, and some more. CFD is connected to a wide scope of research and building issues in numerous fields of study and enterprises, including optimal design aerodynamics and aviation investigation.

The crucial premise of practically all CFD issues is the Navier Stokes conditions, which characterize many single-stage (gas or fluid, yet not both) liquid streams. These conditions can be streamlined by evacuating terms depicting thick activities to yield the Euler conditions. Further disentanglement, by expelling terms depicting vorticity yields the maximum capacity conditions. At long last, for little bothers in subsonic and supersonic streams (not transonic or hypersonic) these conditions can be linearized to yield the linearized potential conditions.

2.1.3 MESSAGE PASSING INTERFACE(MPI)

MPI[\[14\]](#) is a particular for the designers and clients of message passing libraries. Without anyone else, it's anything but a library - yet rather the detail of what such a library ought to be. **MPI** basically addresses the message-passing parallel programming model: Information is moved from the location space of one procedure to that of another procedure through helpful tasks on each procedure.

It is a framework that expects to give a versatile and proficient standard for message passing. It is broadly utilized for message passing projects, as it characterizes helpful punctuation for schedules and libraries in various PC programming dialects.

2.2 REQUIREMENTS

2.2.1 MESH FILE

A polygon mesh is an accumulation of vertices, edges and faces that characterizes the state of a polyhedral article in 3D PC designs and strong displaying. The countenances typically comprise of triangles (triangle work), quadrilaterals, or other straightforward curved polygons, since this disentangles rendering, however may likewise be made out of increasingly broad inward polygons, or polygons with gaps. Items made with polygon networks must store distinctive kinds of elements[\[4\]](#).

These incorporate vertices, edges, faces, polygons and surfaces. In numerous applications, just vertices, edges and either faces or polygons are put away. Polygon cross sections might be spoken to in an assortment of ways, utilizing

distinctive techniques to store the vertex, edge and face information. These incorporate Face-vertex networks, Winged-edge, Half-edge networks, Quad-edge networks, Corner-tables, Vertex-vertex networks. There exist a wide range of record groups for putting away polygon work information. Each arrangement is best when utilized for the reason planned by its maker. Here we utilize *VTK* format.

The Visualization Toolkit (*VTK*) is an open-source, unreservedly accessible programming framework for 3D PC illustrations, picture handling and perception. In this format, it can spare hybrid networks that contain several sorts of finite elements. A mesh can be characterized by various bits called pieces, every one containing its own directions, networks and fields. The worldwide ordering of nodes, edges, faces and components are characterized by integers and the vertex coordinates and field values are characterized by floats.

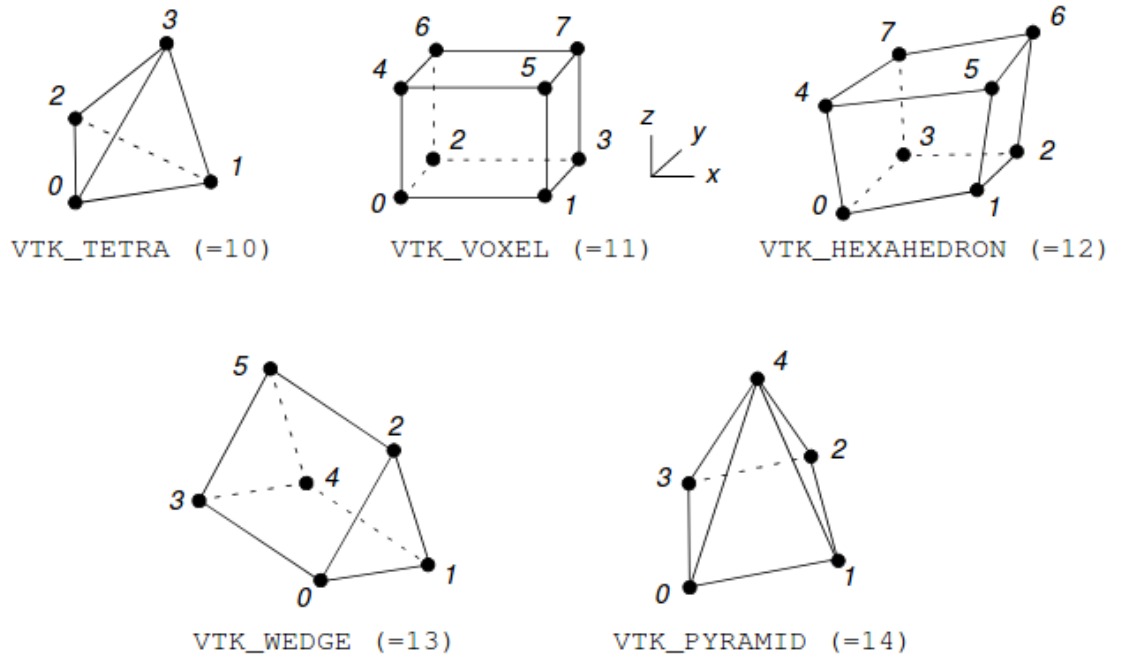


Figure 2.1: Vertex order of different elements in mesh file(*VTK* format)

In Figure 2.1, VTK_VOXEL, represent the vertex order of different elements in cube like mesh file. In this mesh, vertices 0-1-3-2 in the order forms a face. Also 0-4-6-2 forms another face. But the vertices in the order 0-1-2-3 cannot represent a face. Similarly, 0-2-4-6 cannot form a face. Thus the order in which faces are represented is 0-1-3-2, 4-5-7-6, 0-4-6-2, 1-5-7-3, 0-1-5-4, 2-3-7-6. This is the ordering used in VTK format.

INPUT FORMAT OF MESH

The SU2 mesh format carries an extension of .su2, and the files are in a readable ASCII format. As an unstructured code, SU2 requires information about both the node locations as well as their connectivity. The connectivity description provides information about the types of elements (triangle, rectangle, tetrahedron, hexahedral, etc.) that make up the volumes in the mesh and also which nodes make up each of those elements[5].

The first line of the .su2 mesh declares the dimensionality of the problem (*NDIME*). The next part of the file describes the interior element connectivity, which begins with the *NELEM*(Number of Elements in the mesh) keyword. Remaining part contains the information about Element in the format.

Connectivity node1 node2 etc...

Where *connectivity* and number of nodes depends on the type of element.

Table 2.1 shows connectivity and number of nodes of different elements[5] used in VTK format and it is derived from Figure 2.1.

Element	Connectivity	Number of nodes
Triangle	5	3
Quadrilateral	9	4
Tetrahedral	10	4
Hexahedral	12	8
Prism	13	6
Pyramid	14	5

Table 2.1: Connectivity and number of nodes of different elements

2.2.2 METIS

The **METIS** [6, 15] is a product bundle for parceling extensive unpredictable graphs and networks. Its source code can be downloaded straightforwardly from <http://www.cs.umn.edu/metis>. **METIS** provides a set of stand-alone command-line programs for computing partitionings as well as an application programming interface (API) that can be utilized to summon its different calculations from C/C++ or Fortran programs.

APPLICATION

PARTITIONING OF GRAPH

METIS can segment an unstructured chart into a client determined number k of parts utilizing either the staggered recursive division or the staggered k -way parceling ideal models. **METIS** independent program for parceling a diagram is **gpmetis** also, the usefulness that it gives is accomplished by the **METIS Part-GraphRecursive** and **METIS PartGraphKway** API routines.

PARTITIONING OF MESH

METIS gives the **mpmetis** program to dividing networks emerging in limited component or limited volume techniques. This program takes as input, the element-node array of the mesh and registers a k -path apportioning for the two components and its nodes. This program first converts the mesh into either a dual graph (i.e., every component turns into a graph vertex) or on the other hand a nodal graph and after that utilizes the diagram partitioning API schedules to segment this diagram. The functionality provided by **mpmetis** is achieved by the **METIS PartMeshNodal** and **METIS PartMeshDual** API routines.

CONVERTING A MESH TO GRAPH

METIS provides the **m2gmetis** program for converting a mesh into the graph format used by **METIS**. This program can generate either the nodal or dual graph of the mesh. The corresponding API routines are **METIS MeshToNodal** and **METIS MeshToDual**.

INPUT FILE FORMAT

The primary input of the mesh partitioning programs in **METIS** is the graph to be partitioned. This mesh is stored in a file in the form of the element node array. A mesh with n elements is stored in a plain text file that contains $n + 1$ lines. The first line contains 2 parameters that is the number of elements, n in the mesh and number of weights, w associated with each element(optional). Figure [2.2](#), [2.3](#), [2.4](#), [2.4](#) and [2.5](#) shows the different input to **METIS**.

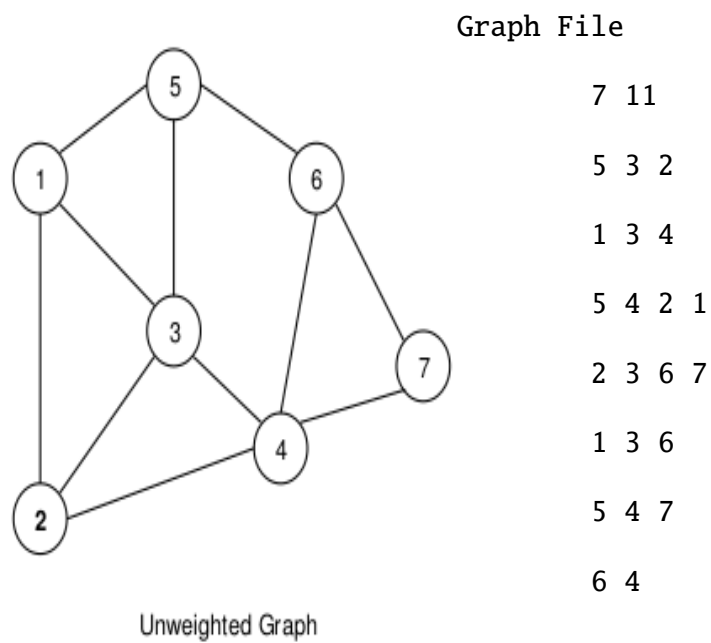


Figure 2.2: Input file to metis - 1

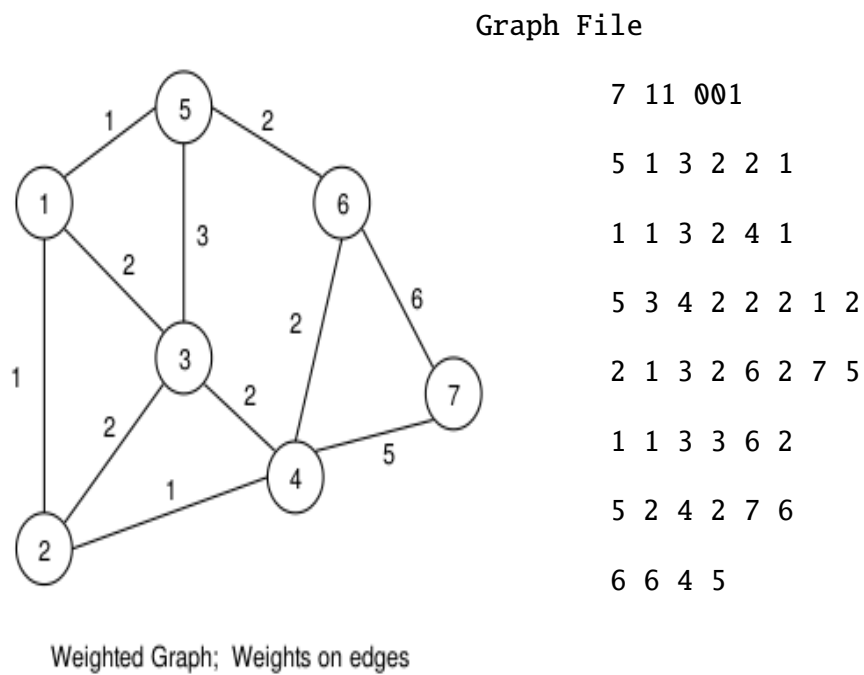


Figure 2.3: Input file to metis - 2

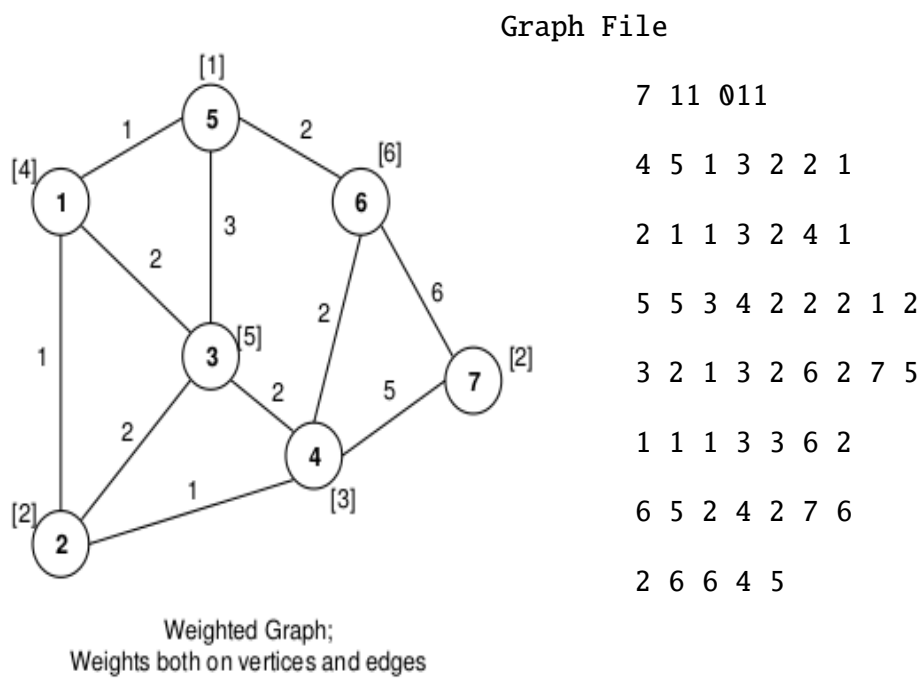


Figure 2.4: Input file to metis - 3

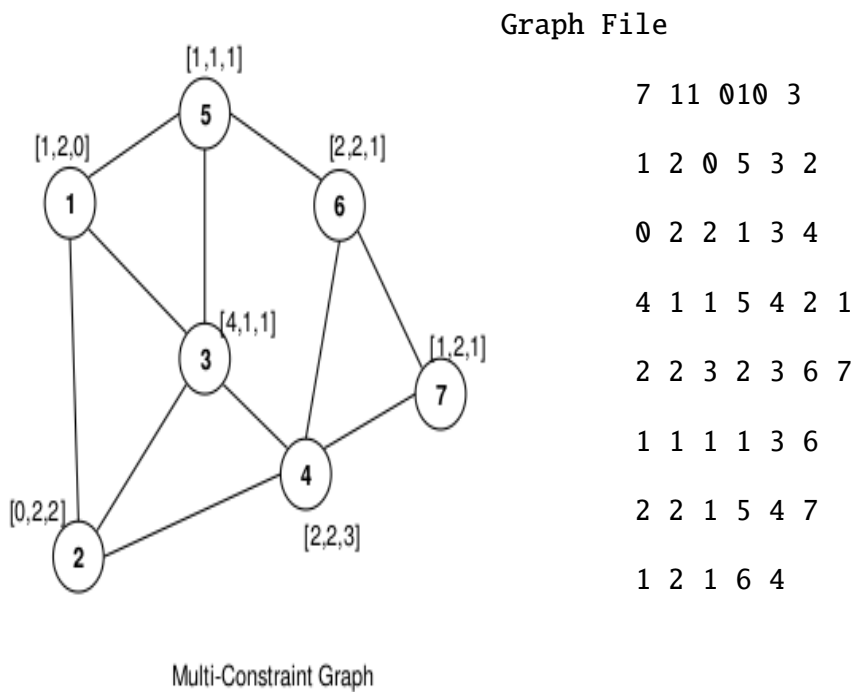


Figure 2.5: Input file to metis - 4

OUTPUT FILE FORMAT

The output of **METIS** is a partition file. The partition file of a graph with n vertices consists of n lines with a single number per line. The i^{th} line of the file contains the partition number that the i^{th} vertex belongs to. Partition numbers start from 0 up to the number of partitions minus one.

METIS is always a better option than DFS/BFS. DFS/BFS technique will not reduce the communication volume (represented as edgecuts in metis). So if we need to replace metis with any other tool, then the proper alternative is space filling curves.

2.2.3 MPI

The **Message Passing Interface (MPI)** [14] is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory. It can be implemented using Open **MPI** which can be download from it's official page: <http://www.open-mpi.org/>

MPI Send and receive

Sending and receiving are the two foundational concepts of **MPI**. Almost every single function in **MPI** can be implemented with basic send and receive calls.

```
MPI_Send (                                int destination ,
        void* data ,                      int tag ,
        int count ,                      MPIComm communicator
        MPI_Datatype datatype ,          )
```

```

MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPIComm communicator,
    MPI_Status* status)

```

- The first argument is the data buffer.
- The second and third arguments describe the *count* and *type* of elements that reside in the buffer.
- **MPI_Send** sends the exact count of elements, and **MPI_Recv** will receive at most the count of elements.
- The fourth and fifth arguments specify the *rank* of the sending or receiving process and the *tag* of the message.
- The sixth argument specifies the communicator and the last argument (for MPI_Recv only) provides information about the received message.

MPI Allgather

Gathers data from all tasks and distribute the combined data to all tasks [\[7\]](#).

```

int MPI_Allgather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    MPI_Comm comm)

```

- The first argument, **sendbuf** defines starting address of send buffer (choice). It is an input parameter.
- The second argument, **sendcount** defines number of elements in send buffer (integer). It is an input parameter.
- The third argument, **sendtype** defines data type of send buffer elements (handle). It is an input parameter.

- The fourth argument, **recvbuf** defines address of receive buffer (choice). It is an output parameter.
- The fifth argument, **recvcount** defines number of elements received from any process (integer). It is an input parameter.
- The sixth argument, **recvtype** defines data type of receive buffer elements (handle). It is an input parameter.
- The seventh argument, **comm** defines communicator (handle). It is an input parameter.

MPI Broadcast

In **MPI**, broadcasting can be accomplished by using **MPI_Bcast** . The function prototype looks like this[Z]:

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator
)
```

Although the root process and receiver processes do different jobs, they all call the same **MPI_Bcast** function. When the root process calls **MPI_Bcast** , the data variable will be sent to all other processes. When all of the receiver processes call **MPI_Bcast** , the data variable will be filled in with the data from the root process.

After the mesh file is partitioned, each node is allotted into different partitions. These partitions are distributed into different systems. Some times the neighbouring node of a particular node in the mesh might not belong to the same partition.

So we create ghost nodes which simulate the neighbouring nodes in the partition where the particular node belongs.

Whenever we are in need of the data from the neighbouring node, we find out the original neighbour node's partition and retrieve the data from the node and then copy it to the ghost node in our partition. Here there is a need to communicate between the different partitions so that the values of ghost nodes can be updated as per the need. This is where **MPI** plays its role.

This chapter discussed about concepts (HPC, CFD and MPI) and requirements (MESH file, METIS, MPI) of the project.

CHAPTER 3

PROPOSED SYSTEM

We are provided with mesh file represented in *VTK* format as initial input. In order to use **MPI** communication, we need to convert this mesh file into graph file which is then partitioned using METIS. And this partitioned graph file is further applied to centralized, peer-to-peer and greedy approach and its performance comparison is noted. Figure 3.1 explains different phases of the project.

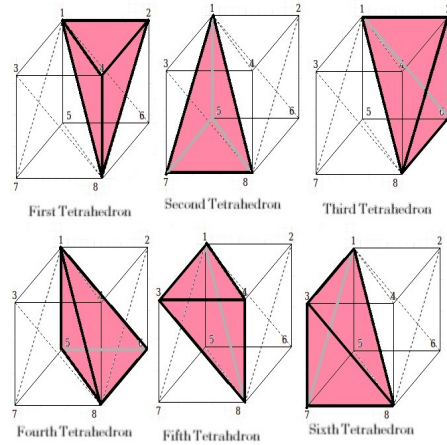
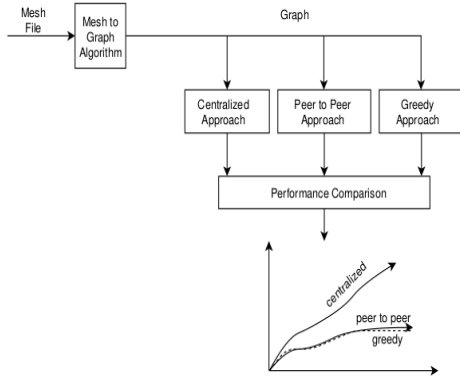


Figure 3.1: Different Phases of Project Figure 3.2: Geometric Representation of Cube

MESH FILE IN *VTK* FORMAT FOR THE SAMPLE CUBE

Cube in Figure 3.2 is represented in *VTK* format using properties of mesh file described in 2.2.1 and it is shown below.

NDIME= 3

NELEM= 6

10 1 2 4 8

10 1 5 7 8

10 1 2 6 8

10 1 5 6 8

10 1 3 4 8

10 1 3 7 8

GRAPH FILE

Graph file can be manually verified using Figure [3.2](#). The first line of the graph file describes the number of vertices and number of edges which are 6 and 6 respectively according to the Figure [3.2](#). In that Figure, 5th tetrahedron and 3rd tetrahedron are neighbours to 1st tetrahedron because they share common face 148 and 128 respectively with 1st tetrahedron. This information is represented by line 2 in the graph file shown below i.e, line 2 indicates 3rd and 5th tetrahedron are neighbours to 1st tetrahedron.

6 6

3 5

6 4

1 4

2 3

1 6

2 5

GRAPH PARTITION FILE

Input to metis is the graph file which is the output of Algorithm [1](#) or Algorithm [2](#)

Output from metis is graph partition file.

```
1
1
2
2
0
0
```

In graph partition file, each line indicates the core in which that vertex belong. In this example, first line indicates that first vertex belongs to core 1, second line indicates second vertex belongs to core 1, third line indicates third vertex belongs to core 2 and so on. Thus 1st and 2nd vertex belongs to core 1, 3rd and 4th vertex belongs to core 2 and 5th and 6st vertex belongs to core 0.

3.1 ALGORITHM TO CONVERT MESH TO GRAPH

3.1.1 ALGORITHM-1

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.

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.
-

In Algorithm [1](#) 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 [1](#) proceeding in the 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 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 \rightarrow 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 \rightarrow

tetrahedron1,tetrahedron2, 1 \rightarrow tetrahedron1,tetrahedron2, 2 \rightarrow tetrahedron1, 3 \rightarrow tetrahedron1,tetrahedron2, 4 \rightarrow 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 greater 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. 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.

3.1.2 ALGORITHM-2

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 Algorithm 2 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.

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.
-

Here, 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 Algorithm 2 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.

3.2 COMMUNICATION

A sample graph and its partition is shown in the Figure 3.3 and Figure 3.4 respectively. Figure 3.3 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 Figure 3.4, 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 subgraph of a core but it's ownership doesn't belong to that core. In Figure 3.4, vertices 10 and 5 are the ghost of core 0 and contributes in its subgraph but vertex 10's ownership belongs to core 2 and vertex 5's belongs to core 1.

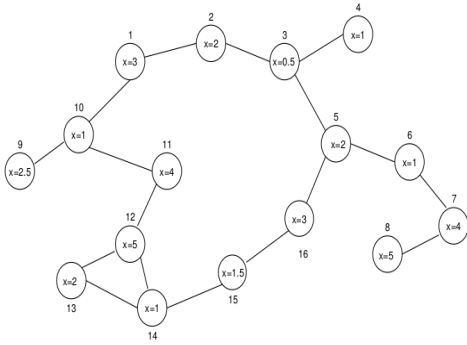


Figure 3.3: Sample graph

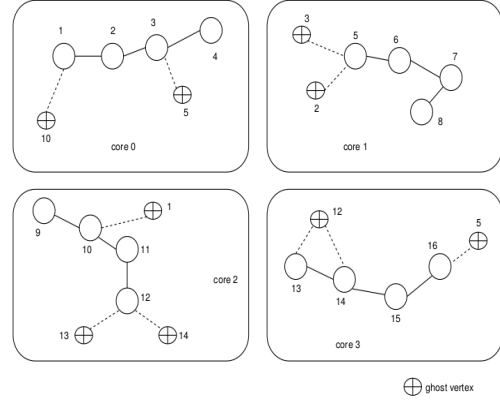


Figure 3.4: Partitioned graph

3.2.1 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 the 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

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
-

In Algorithm 3, 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 and it is depicted in Figure 3.5. Vertex data is available to both core 0 and the core in

which it belongs. After constructing subgraph, cores check 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 and this is shown by Figure 3.6.

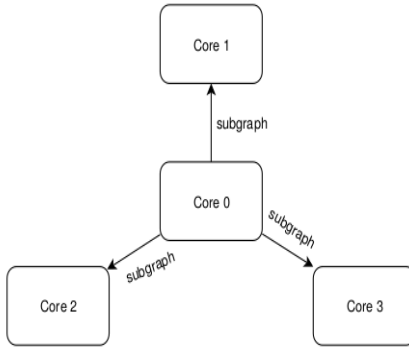


Figure 3.5: Sending vertex to corresponding core and constructing sub-graph in centralized approach

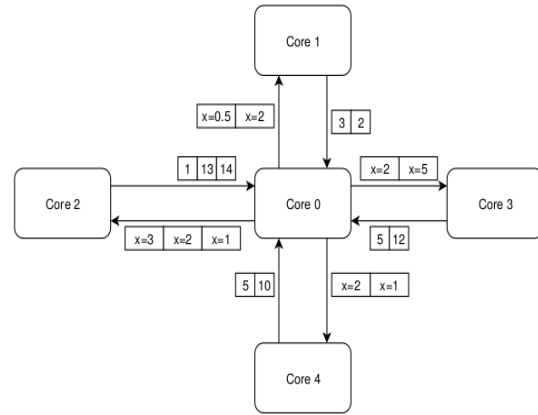


Figure 3.6: Requesting and sending of ghost data in centralized approach

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.

3.2.2 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 parallelly 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

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.
-

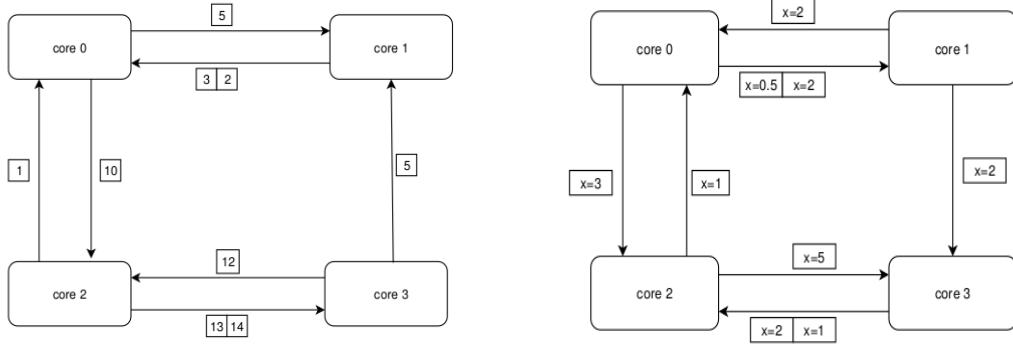


Figure 3.7: Requesting ghost data in peer-to-peer approach Figure 3.8: Sending ghost data in peer-to-peer approach

Algorithm 4 and Figure 3.7 and 3.8 explains the peer-to-peer concept.

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.
- Figure 3.9 shows the deadlock situation in peer-to-peer approach 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.

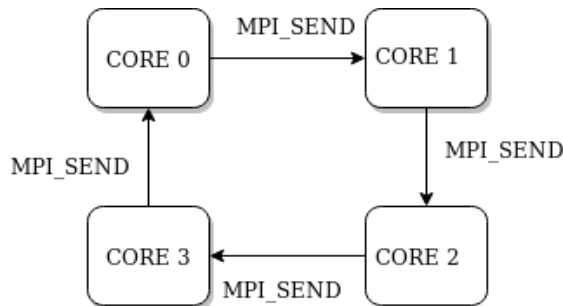


Figure 3.9: Deadlock situation in peer-to-peer approach

For example, consider a situation where core 0 sends a MPI.Send to core 1, core 1 sends a MPI.Send to core 2, core 2 sends a MPI.Send to core 3 and

core 3 sends a MPI_Send to core 0. Here communication between core 0 and core 1 is completed only when core 1 receives data that was sent by core 0 however core 1 can initiate MPI_Recv only when core 2 had received data that was sent by core 1. hence this will cause deadlock.

- 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.

3.2.3 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 Algorithm 5, initially a synchronization message is send between each core. Figure 3.4 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 Figure 3.10, core 0 is

sending 0110 to every other core indicating that it contains vertex that core 1 and core 2 are in needed.

Also, here before sending ghost data, communication load (number of ghost vertex send between each core) is arranged in matrix (Figure 3.11) 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 i.e, first pair in slot 0 is communicated first.

Figure 3.12 and 3.13 represent arrangement of slot 0 and slot 1 respectively based on Figure 3.11.

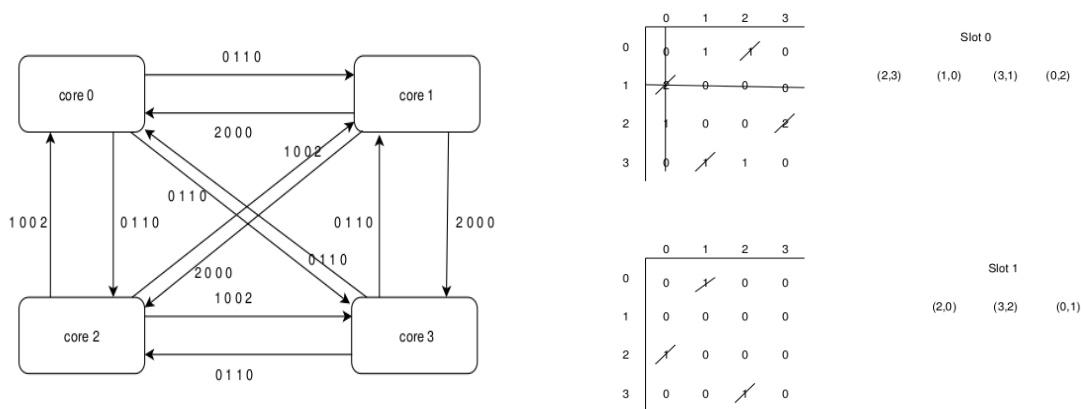


Figure 3.10: Sending message synchronization Figure 3.11: Process of taking maximum load from matrix and forming slots

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 'i' 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.
-

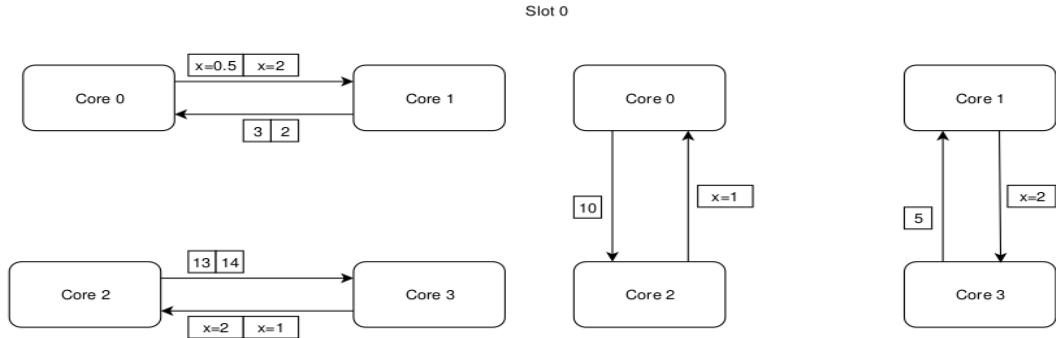


Figure 3.12: Representation of slot-0

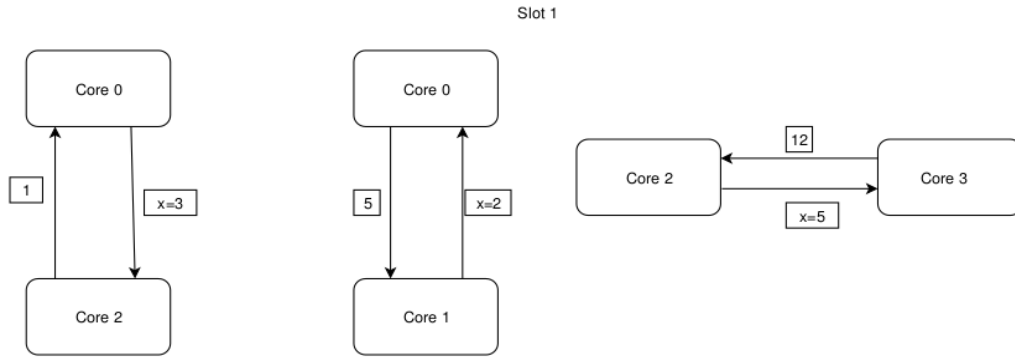


Figure 3.13: Representation of slot-1

3.3 THEORETICAL PROOF

3.3.1 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**.

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_Recv for receiving ghost details from 0th core. In total there are $m - 1$ cores, therefore amount of communication will be,

$$2n + 2(m - 1) * (n - 1)$$

General case:

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

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

where k_i is the number of ghost in i^{th} core

3.3.2 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.

Worst case:

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

$$2(m - 1) * m$$

This chapter discuss about the proposed system of the project. It explains algorithm for converting mesh to graph and different communication approaches (centralized, peer-to-peer and greedy) and theoretically proved peer-to-peer and greedy approach is more efficient than centralized approach based on total amount of communication. It also proved greedy approach is more efficient than peer-to-peer based on delay.

CHAPTER 4

RESULTS AND DISCUSSION

Table 4.1 compares different communication approaches based on total amount of communication. Total amount of communication is calculated both theoretically using equation given in Section 3.3 and experimentally using algorithms given in Section 3.2 and it also showed that the total amount of communication for centralized approach is much higher than both peer-to-peer and greedy approach and amount of communication for both peer-to-peer and greedy approaches are same.

Since theoretical value indicates worst case (each core communicates with every other core) and occurrence of worst case is not common and in most cases cores communicate with the core in which it's ghost reside, experimental and theoretical value of peer-to-peer and greedy varies largely at higher ranges. That is in Table 4.1, experimental and theoretical value of peer-to-peer and greedy for 100 cores and 7434 vertices are 820 and 19800 respectively.

Since experimental and theoretical value of greedy and peer-to-peer for different inputs are same, peer-to-peer and greedy approach is again compared based on delay to obtain efficient approach among them and it is depicted by Table 4.2.

	Total amount of communication				
Approach	No. of cores	No. of vertices	No. of ghost vertices	Experimental value	Theoretical value
Centralized	3	6770	146	13555	13832
	5	6770	248	13865	14036
	8	7434	119	14908	15106
	15	6770	658	14458	14856
	100	7434	3712	21984	22292
Peer to peer	3	6770	146	8	12
	5	6770	248	20	40
	8	7434	119	48	112
	15	6770	658	100	420
	100	7434	3712	820	19800
Greedy	3	6770	146	8	12
	5	6770	248	20	40
	8	7434	119	48	112
	15	6770	658	100	420
	100	7434	3712	820	19800

Table 4.1: Comparison between different communication approaches based on total amount of communication

Approach	No. of cores	No. of vertices	No. of ghost vertices	Delay
Peer to peer	3	6770	146	0.0484563
	5	6770	248	0.110254
	8	7434	119	0.167259
	15	6770	658	0.220573
	100	7434	3712	1.68878
Greedy	3	6770	146	0.000153747
	5	6770	248	0.0183151
	8	7434	119	0.0800315
	15	6770	658	0.148088
	100	7434	3712	0.958311

Table 4.2: Comparison between greedy and peer to peer based on delay

From Table 4.2, it is straight forward that peer to peer incur more delay compared to greedy. it is 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.

From Table 4.1 and Table 4.2, it is derived that greedy approach is more efficient approach compared to both centralized and peer-to-peer approach.

CHAPTER 5

CONCLUSION

In this report, we developed a distributed **MPI** framework for solving computational problems efficiently and designed data layouts which can effectively utilise memory bandwidth. Multiple re-ordering schemes, along with data duplication is attempted to improve the memory performance. And also proved, greedy approach is the best communication approach which is free from deadlock and starvation.

For partitioning graph file, we are using METIS software instead of DFS/BFS technique. It is because, DFS/BFS will not reduce communication volume (represented as edgecuts in metis). But, still there is a proper alternative for METIS which is space filling curves [13]. 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 [11, 12].

MPI is not only the standard for communication. There are many other communication libraries like ARMCI [8], Portals [9] etc. Which are used for special cases. But here we work on **MPI**.

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 **MPI C++**.

In future, instead of **MPI**, **Charm++** [10] can be used. Like Java RMI, or RPC, **Charm++** is the programming model which is an asynchronous message driven paradigm. but it is intended for high performance parallel machines coupled closely. Unlike **MPI's** "single program, multiple data" (SPMD) programming and execution model, Charm++ programs do not proceed in lockstep. The flow of control is determined by the order in which remote method invocations occur.

Also instead of greedy approach, an efficient dynamic approach can be developed by identifying sub problems in the synchronization matrix.

APPENDIX A

HOW TO RUN IN A CLUSTER

1. Establish physical connection between the systems.
2. Make sure that same **version** of MPI is installed on **all** systems.
3. Create a common user account **for all** systems.
4. Make sure that files are on same **path** on **all** the systems.
5. Enable password-less login in **all** systems.
6. Create a hostfile listing the ip address of **all** other systems.
7. Execute the mpi program passing the hostfile as argument.

REFERENCES

- [1] (). URL <https://www.nginx.com/resources/glossary/load-balancing/>.
- [2] (). URL <https://searchdatacenter.techtarget.com/definition/high-performance-computing-HPC>.
- [3] (). URL <https://www.simscale.com/docs/content/simwiki/cfd/whatis CFD.html>.
- [4] (). URL <https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf>.
- [5] (). URL <https://su2code.github.io/docs/Mesh-File/>.
- [6] (). URL <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>.
- [7] (). URL <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>.
- [8] (). URL https://wiki.mpich.org/armci-mpi/index.php/Main_Page.
- [9] (). URL <http://www.nwchem-sw.org/index.php/Release66:ARMCI>.
- [10] (). URL <http://charm.cs.illinois.edu/research/charm>.
- [11] (). The basic equations of fluid dynamics. URL <https://community.dur.ac.uk/suzanne.fielding/teaching/BLT/sec1.pdf>.
- [12] (). Fluid flow: Conservation of momentum, mass, and energy. URL <https://www.comsol.com/multiphysics/fluid-flow-conservation-of-momentum-mass-and-energy>.
- [13] (2006). What are alternatives to mpi for communication in parallel computing? URL <https://insidehpc.com/2006/06/what-are-alternatives-to-mpi-for-communication-in-parallel-computing/>.
- [14] **Gropp, W.**, *Tutorial on MPI: The Message-Passing Interface..*
- [15] **Karypis, G.**, *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0.* 1998.