Georgia State University

ScholarWorks @ Georgia State University

Computer Science Theses

Department of Computer Science

8-2-2006

iC2mpi: A Platform for Parallel Execution of Graph-Structured Iterative Computations

Harnish Botadra

Follow this and additional works at: https://scholarworks.gsu.edu/cs_theses

Part of the Computer Sciences Commons

Recommended Citation

Botadra, Harnish, "iC2mpi: A Platform for Parallel Execution of Graph-Structured Iterative Computations." Thesis, Georgia State University, 2006.
doi: https://doi.org/10.57709/1059370

This Thesis is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Theses by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

IC2MPI: A PLATFORM FOR PARALLEL EXECUTION OF GRAPH-STRUCTURED ITERATIVE COMPUTATION

by

HARNISH A BOTADRA

Under the Direction of Sushil Prasad

ABSTRACT

Parallelization of sequential programs is often daunting because of the substantial development cost involved. Various solutions have been proposed to address this concern, including directive-based approaches and parallelization platforms. These solutions have not always been successful, in part because many try to address all types of applications. We propose a platform for parallelization of a class of applications that have similar computational structure, namely graph-structured iterative applications. iC2mpi is a unique proof-of-concept prototype platform that provides relatively easy parallelization of existing sequential programs and facilitates experimentation with static partitioning and dynamic load balancing schemes. We demonstrate with various generic application graph topologies and an existing application, namely a time-stepped battlefield management simulation, that our platform can produce good performance with very little effort.

INDEX WORDS: Parallelization, Load Balancing, Graph Partitioning, Battlefiled Management Simulation, Metis, PaGrid

IC2MPI: A PLATFORM FOR PARALLEL EXECUTION OF GRAPH-STRUCTURED ITERATIVE COMPUTATION

by

HARNISH A BOTADRA

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Science

Georgia State University

2006

Copyright by
Harnish A Botadra
2006

IC2MPI: A PLATFORM FOR PARALLEL EXECUTION OF GRAPH-STRUCTURED ITERATIVE COMPUTATION

by

HARNISH A BOTADRA

Major Professor:	Dr. Sushil Prasad
Committee:	Dr. Raj Sunderraman
	Dr. Alex Zelikovsky

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
August 2006

Acknowledgements

I would like to thank my advisor, Dr. Sushil K. Prasad, for his expert guidance. He was always receptive to new ideas and ready to explore options. He always helped me see the bigger picture thus helping me focus on the main issues in the system.

I would also like to thank Dr. Eric Aubanel and Dr. Virendra Bhavsar of University of New Brunswick, Canada for their review of the manuscript.

Thanks to Qiong Cheng for helping me, with a portion of the Experimental Results related to my research work.

Dr. Raj Suderraman & Dr. Alex Zelikovsky were kind enough to review the manuscript and provide me with fine pointers to meet the standards.

Dedicated to everyone who was a part of this
For all the support

<u>**Table of contents**</u>

List of Tables

List of Figures

1. Introduction

Many application domains, wherein distributed computation has been successfully employed, fall into the class of iterative computations – this class can be characterized by underlying mesh or graph structured application programs and iterative local computations over nodes, dependent only on the neighboring nodes. Examples of such application domains include many time-stepped simulations, such as battlefield management [DMP98], weather forecasting [K90], or fluid dynamics [HMP03], and mesh-structured computations, such as difference equations [Q04], finite element methods [SG04], and cellular automata [CCE01]. Despite sharing essentially similar program structures, there is no generic framework to help the programmers of these applications to easily transition from their sequential implementations to distributed machines or to grid platforms (see Section 6 for relevant literature). Our iC2mpi platform addresses this and related issues.

1.1 Current Limitations

Typically, scientists and engineers have proven sequential C/C++ codes and converting these to distributed versions in MPI entails challenges of explicit parallel programming, debugging, and revalidating. Most applications programmers, therefore, are limited to automatic loop-based parallelization, or to inserting compiler directives (as in OpenMP [OP] and HPF [Z02]). If distributed versions are manually developed (usually employing MPI), quite often these programs hard code a specific pattern of domain decomposition to statically partition the application program graph/mesh among the processors of a target machine topology, aiming to best balance computational load and minimize communication. Therefore, the resultant MPI code that programmers develop typically requires code changes to study various static partitioning techniques, as these are not developed in a framework-like fashion to easily enable such performance studies. When load can be unpredictable, for example in a battlefield simulation where combat zones form dynamically, they also need to employ dynamic load balancing, which requires even more versatile design of the distributed programs, further challenging an application programmer.

On the other hand, the algorithm designers of graph partitioning packages, such as Metis [KK98], Jostle [WC01], and PaGrid [WA04, HAB06], are also limited as they can only estimate the efficiency of their techniques analytically. There is currently no general-purpose test-bed available that allows them to easily plug-in their algorithms, and execute and verify the performances on various program graphs and processor architectures.

1.2 Specific Goals

This work describes our project to develop a suitable platform with the following goals.

1. Design an MPI-based platform with an open architecture for the class of iterative graph-structured application programs, possibly with dynamically varying computational loads, into which application programmers can *plug-in* the code and the data structures for their computational nodes, the graphs for their application programs and for the processor network, and the third-party algorithms for partitioning and load balancing.
2. Enable application programmers to

   a. easily execute their sequential code for iterative computations on distributed architectures without any code change in their node computations or in the basic node data structures, and without any MPI coding, and
   b. to compare the performance of different static graph partitioners, and the impact of various dynamic load balancing and repartitioning techniques, without any additional coding.
3. Enable designers of algorithms for graph partitioning and for dynamic load balancing to validate the efficiency of their techniques by actual execution over a variety of graph-structured iterative computations and load characteristics on different parallel and distributed architectures and heterogeneous grids instead of typical analytical estimation.
4. Enable carrying out of refinements and performance tuning for efficient computation and communication on the platform itself to impact the entire class of relevant iterative computations.

1.3 iC2mpi Solution

Our resultant platform, namely iC2mpi (for enabling transition from an *i*terative *c*omputation in *C* to an *MPI*-enabled execution), is our first prototype demonstrating a proof of concept. Its initial goal has been to architect a generic platform, fulfilling Goals 1, 2, and 3. Although efficiency enhancements of the platform have not been taken up yet (Goal 4), we do obtain reasonably good speedups over a variety of example codes. Section 2 describes the methodology entailed in deploying various graph topologies and Battlefield Management Simulation on our platform. Section 3 describes the overall architecture of iC2mpi, and Section 4 gives the details of its internal algorithms and data structures. Section 5 shows the results of various experiments with different iterative applications on iC2mpi platform. We employ Metis and PaGrid as examples of third-party plug-ins to study various static partitioning schemes. We have also developed our own simple repartitioning heuristic plug-in to experiment with dynamic task migrations.

We had earlier developed a large battlefield management simulator [DMP98] which employs hexagonal grid-based terrain. Each grid cell simulates all its battlefield units in each time step. It was efficiently parallelized earlier on hypercube machines using PVM employing mesh-to-hypercube gray-code-based embedding. But its efficient yet rigid array-based hex-cell data structure and associated messaging algorithms became serious impediment to incorporate various static (or dynamic) partitioning schemes, and needed major code changes. Using iC2mpi platform, we here demonstrate that we are able to convert the C code and study various partitioning schemes with relative ease. Section 6 briefly reviews related work, Section 7 talks about the possible design enhancements and Section 8 concludes by discussing ongoing and future work in extending the iC2mpi platform and improving its performance.

2. Methodology

In this section we talk about the methodology entailed in parallelization of the various graph topologies and the battlefield management simulation deployed on the iC2mpi platform. We discuss three user plug-in points in each of the cases namely the application program graph, node data structures and the node computation function.

2.1 Graph Topologies

Methodology for both the hexagonal grids and the random graphs would be the same, so we shall just discuss one over here.

Application Program Graph:

Different size application program graphs are generated (32- and 64- node in case of random graphs & 32-, 64- and 96- node in case of hexagonal grids). The application program graph is input to the static graph partitioner (example: Metis and PaGrid). The static graph partitioner gives the initial node to processor mapping. The initialization phase uses this mapping and initializes the required data structures in the local memory of the processors. The application program graph is supplied from the command line to the platform as: mpirun –np num_procs MPIFramework $program_graph. Here, ‘num_procs’ is the number of processors and ‘$program_graph’ is the application program graph.

Node Data Structures:

The struct type used for the node data structures is as shown in figure 1.

‘globalID’ is the global node identifier. ‘data’ stores the actual node data. Updated node data is stored in ‘most_recent_data’, since the old data might still be required for the computation purposes of the neighboring nodes. Finally, ‘*next_node’ is a pointer to the

```
struct node_data
{
int globalID:
int data;
int most_recent_data;
struct node_data * next_node;
};
```

Figure 1: Node Data Structure for Graph Topologies

next node in the list. These node data structures are initialized and fed to the ‘Initialization Phase’ of the platform. In the platform’s current version, there’s no clean interface for the user to input the node data structures. Providing a clean user interface for the user to easily feed the node data structures to the ‘Initialization Phase’ of the platform is one of the main design enhancements required in the current version.

Computation Function:

For these graphs, each node computes the average of the data maintained by all its neighbors. A dummy ‘for loop’ is used to inject the grain size. A size of 0.3 ms is used for the fine grain and 3 ms is used for the coarse grain. ‘Computation Phase’ of the platform maintains a pointer to this computation function. For updating of a node, a list is sent to this function with the current node’s data as the head followed by the data of its neighbors. So this computation function needs to compute over the node using this specification. Look at the appendix B with the user code for this computation function and how the platform’s ‘Computation Phase’ invokes this node computation function by maintaining a pointer to the same.

2.2 Battlefield Management Simulation

As mentioned, Battlefield management simulations are employed to evaluate doctrine and tactics, and to predict the battle outcome in various scenarios [DMP98]. These are amongst the most computationally intense and complex simulations in existence. Though, not tried with the dynamic load balancer on our platform, battlefield simulation remains an interesting topic for researching load balancing because its computation load dynamically changes with both time and space.

Application Program Graph:

32 x 32-hex battlefield (computational domain is divided into hexes and hex-to-processor assignment is used for initial partitioning) graphs are used and various static graph partitioners are used to obtain the initial partitioning.

Node Data Structures:

Figure 2 shows the node data structures used for deploying this application on the platform. Figure 3 shows the data structures used for maintaining node information (compare this with figure 7). Refer to [DMP98] for the initialization of these data structures.

Computation Function:

As in the previous case, this function is invoked from the platform’s computation phase which maintains a pointer to this function. Here, there’s one small customization in the platform flow of control. For the battlefield management simulation the computation and communication overlap with each other. Hence, the computation and communication function sequence is called more than once, rather than just once.

```
typedef struct nData{
          int globalID;
          int data;
          hex_node_data_struct *hexdata;
          int most_recent_data;
          struct nData *next_node;
}  node_data;

typedef struct {
          unsigned       neighbor[6];/* the six neighboring processor node */
          int            hex_displacement[6];
          hex_struct      my_units[no_of_hexes_per_processor];
                      /* my_units[i] contains the current units in `i'th hex */
          hex_struct      buffer[6][no_of_hexes_per_processor];
                      /* temp space to store the neighbor's units */
          target_type target[no_of_hexes_per_processor];
                      /* target units for each of my_units */
          unsigned
destroyed[no_of_hexes_per_processor][2][no_of_units_per_hex][7];
                      /* destroyed[hex][0/1][i][j] is the number of assets
                      that the red/blue unit i at `hex' has destroyed
                      in the direction j (0..5 neighbor, 6 own hex)*/
          hex_struct       old_status_of_my_units[no_of_hexes_per_processor];
                                          /* temp storage to save the status of my units
*/
          unsigned        my_unit_size, no_of_units, nodeid;
          unsigned        target_list_length, cumulative_target_list_length;

}hex_node_data_struct;
```

Figure 2: Node Data Structures used for Battlefield Management Simulation

```
typedef struct own
{
        int globalID;
        char internal_or_peripheral;
        int owning_proc;
        struct nData *data_location;
        int neighboring_nodes[BF_MAX_Neighbor_NUM ];
        int processor_neighbors[BF_MAX_Neighbor_NUM ];
        int shadow_for_procs[BF_MAX_Neighbor_NUM ];
        struct own *next_node;
} own_node;
```

Figure 3: Data Structure for maintaining node information for Battlefield Simulation

3. Platform Architecture

In this section we discuss the platform architecture, before we go in-depth with each of the platform components in the next section. Figure 4 shows the layered architecture view of the platform. The application program sits on top and provides the application graph, node data structures and the node computation function as user plug-ins to the platform. The platform uses a static graph partitioner for the initial partitioning. A dynamic load balancer is incorporated in the platform for load balancing of dynamic domain applications.

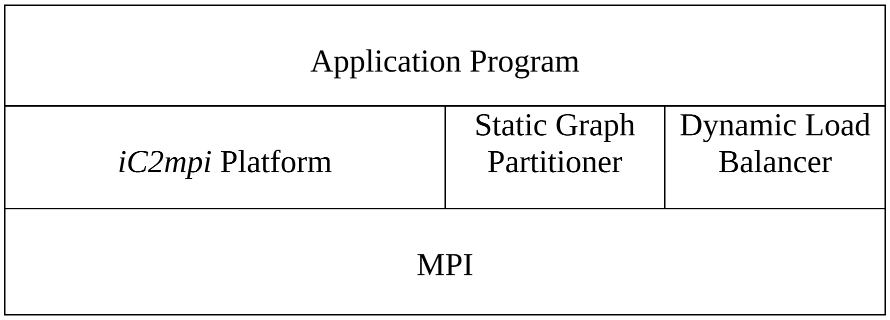

Figure 4: Layered Architecture View of the iC2mpi Platform

The platform uses an MPI approach for parallelization, one of the most widely used methods to achieve parallelism on today's clusters and multiprocessor supercomputers [SM02]. Figure 5 shows the detailed platform architecture, and explicitly indicates user plug-in points (application program graph, node data structures, and node computation function) and the interaction of the platform components with the data structures they use. The static graph partitioner and dynamic load balancer are the third-party components. Solid templates are the platform components. To begin, Metis, a package for partitioning irregular graphs [KK98] (or, PaGrid, another package designed specially for heterogeneous grids), is used to obtain the initial partitioning. The application problem graph is input to Metis/PaGrid, which gives the initial node-to-processor mapping. There are three major phases involved in the platform architecture: initialization, computation & communication, and load balancing & task migration. The

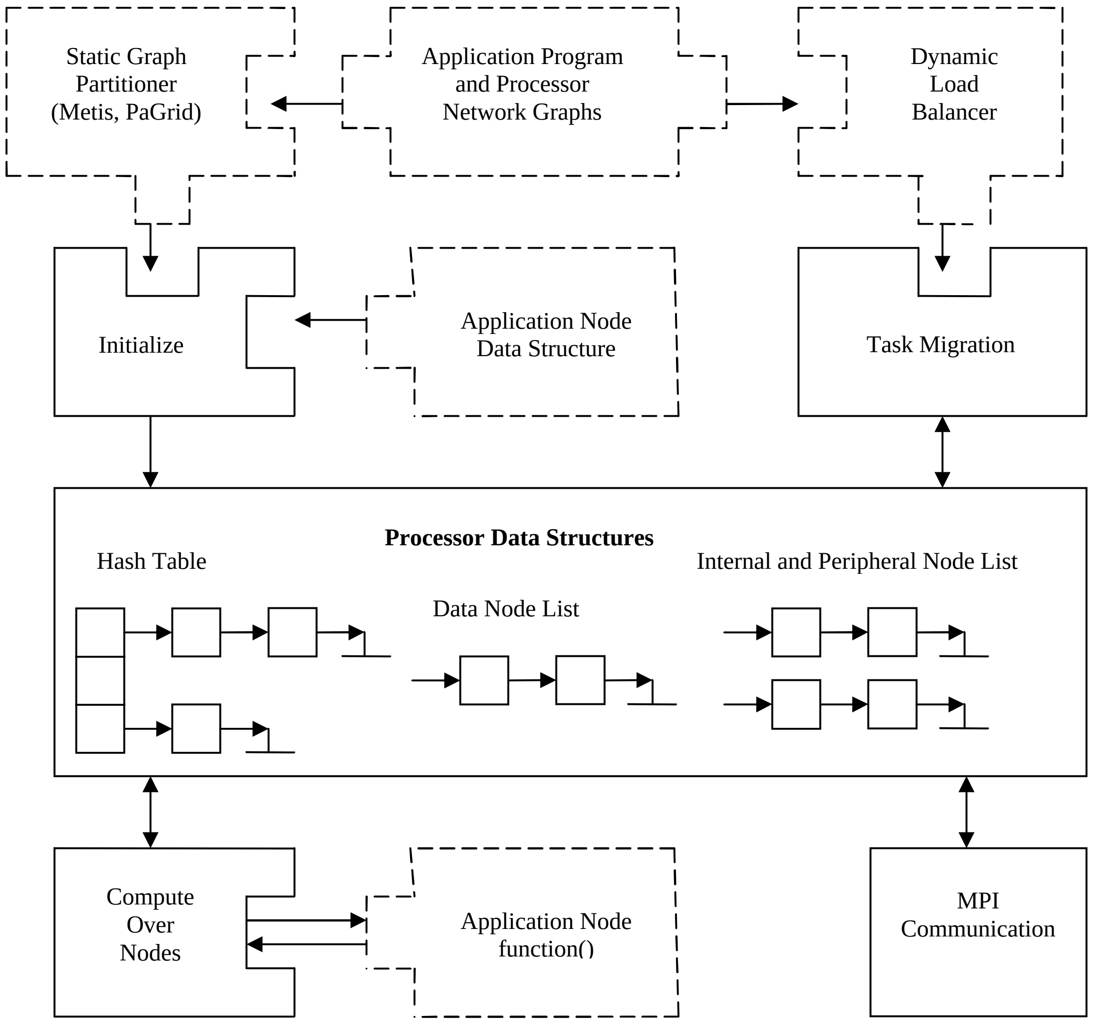
Static Graph Partitioner (Metis, PaGrid)
Application Program and Processor Network Graphs
Dynamic Load Balancer
Initialize
Application Node Data Structure
Task Migration
Processor Data Structures
Hash Table
Data Node List
Internal and Peripheral Node List
Compute Over Nodes
Application Node function()
MPI Communication

Figure 5: Detailed Architecture of iC2mpi Platform

initialization phase uses the node-to-processor mapping provided by the static graph partitioner to initialize the required data structures in the local memory of the processors. It takes the Application Node Data Structures as a user plug-in to initialize the Internal and Peripheral Node List, the Hash Table, and the Data Node List. Compute Over Nodes is responsible for the node computation. It invokes the Application Node function for the node computation and updates the data structures with the new node data. Updated peripheral node data is then communicated to appropriate neighboring processors by MPI Communication. Dynamic load balancer is periodically invoked to balance the work-load on the fly for dynamic applications with varying load requirements. We briefly talk about each of these components.

Initialization Phase: 'Initialize Data Structures' uses the node data structures provided by the user to set up the data structures for maintaining node information and node data in the local memories of the processors. In addition to the data of the nodes owned by the processor, it also maintains shadow node information locally. The data structures set up include internal and peripheral node list for maintaining node information, the data node list for maintaining the node data and hash table for easy access of the node data. Note that the data node list not only includes data for the nodes owned by the processor but also the shadow node data. This phase can be a sizeable overhead for large application program graphs. Good initial partitioning which more or less assigns the same amount of work to each of the processors and minimizes the edge-cut would keep this overhead to a minimum. Different static graph partitioners can be plugged-in not only to study the different partitioning of the graph partitioners but also to fine tune the performance of the iterative computation parallelized on the platform.

Computation & Communication Phase: 'Compute over Nodes' does the actual computation for each of the nodes owned by the processor using data of the neighboring nodes, also maintained locally. A list with the current node data as the head, followed by the data of the neighbors is passed to the Application Node function invoked using a pointer maintained by this routine. Application Node function provided by the user incorporates the actual code for the node computation. Updated peripheral node data from the application node function is packed into communication buffers. By the time ‘Compute Over Nodes’ returns, the communication buffers to communicate the shadow node information to the neighboring processors are already set up. Hence, ‘Communicate Shadows’ just sends the necessary shadow node information packed in the buffers to the neighboring processors and receives the updated shadow node information from its neighbors. Note, that there’s an overlap in the overheads of computation and communication so as to maximize the throughput.

Load Balancing & Task Migration Phase: The load balancing routine is periodically invoked to check for any load imbalance. We incorporate a centralized heuristic algorithm as the dynamic load balancer, which can also be a third party plug-in. It measures the relative work done by each processor, before deciding if the load-imbalance is substantial enough to invoke the task migration routine. Based on the load imbalance statistics gathered by the load balancing routine, the task migration routine actually balances the work load among the processors by sending the task from a busy processor to its idle neighbor at different locations in the computational domain where there’s substantial load-imbalance. Hence, this phase helps maintain the work load among the processors. In essence, it provides node to processor remapping without having to go through the initialization phase all over again. This phase is very important for dynamic applications with varying load requirements. There’s no way a static graph partitioner can capture varying load requirements. Dynamic load balancer balances the work-load on the fly. Invoking the initialization phase for re-partitioning from the scratch can be very costly. Hence, this phase is vital to realize the true potential of a parallel platform. Figure 6 shows the framework components in action in the system flow of control.

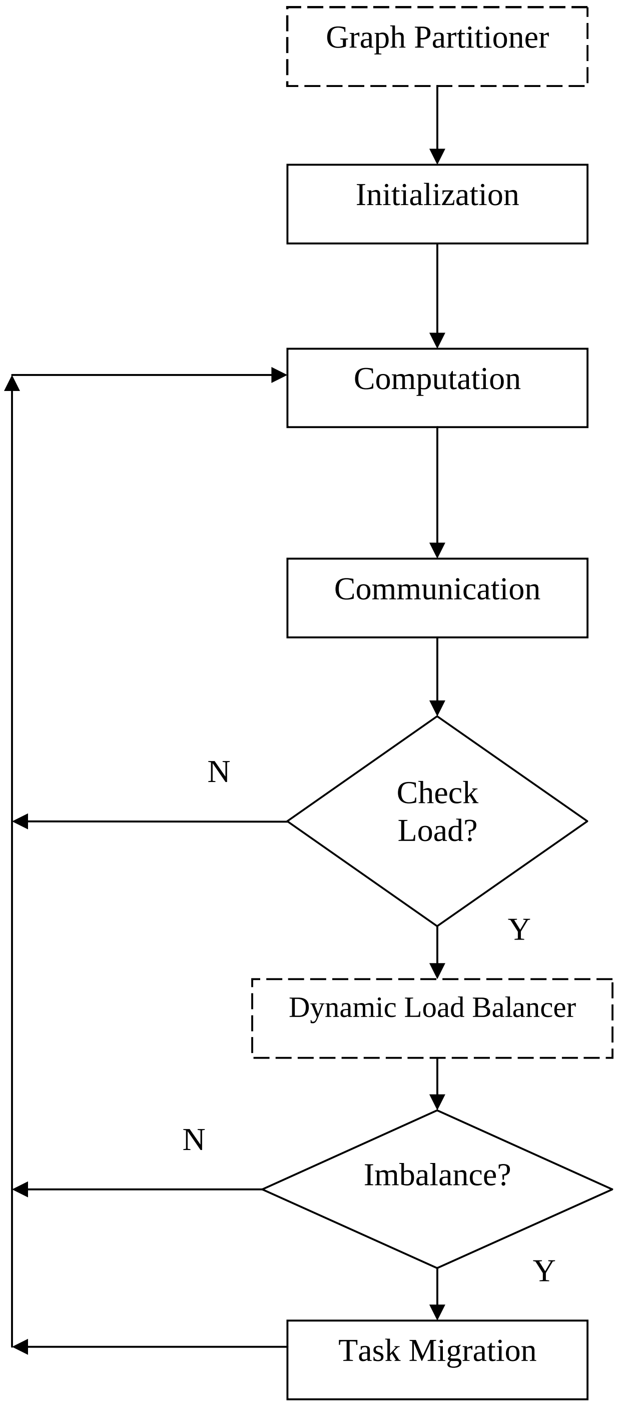

Figure 6: System Flow of Control

Different templates used for Graph Partitioner and Dynamic Load Balancer, emphasize that those can be third-party packages. Metis and PaGrid were employed as the static graph partitioners. Later section on Experimental Results compare the performance of these two partitioners on various graph topologies employed on the platform. For attaining dynamic load balancing a centralized heuristic algorithm was used in the platform. A third party dynamic load balancer can be used instead. Hence, the platform provides a test bed to study partitioning algorithms and static and dynamic load balancing strategies.

4. Detailed Data Structures & Algorithms

We now discuss the data structures and algorithms involved in each of the three phases of the platform in detail.

4.1 Initialization Phase

This phase is concerned with the data structures initialized in the local memory of each of the processors, which are required for efficient computation and communication. Selection of data structures is central to the efficient solving of irregular concurrent problems on distributed memory architectures [PSC95].

During this phase, all the processors initialize data structures in their local memories to maintain graph connectivity of the iterative problem, node information and node data. In addition to the data of the nodes owned by the processor (some are internal nodes and others are peripheral nodes, the ones which have at least one neighboring node on another processor), it also maintains shadow node information locally (the non-local nodes which are neighboring to its peripheral nodes). The data structures set up include internal and peripheral node list for maintaining node information, the data node list for maintaining the node data, and a hash table for fast access of the node data. As mentioned the data node list not only includes data for the nodes owned by the processor but also the shadow node data.

The initial input graph and the initial node-to-processor mapping yielded by graph partitioner is stored into appropriate data structures. Typically, since file access is slower than data structure access, the available information in file format is copied into data structures for easy access and retrieval whenever required.

Figure 7 shows the node information maintained by each processor. Maintaining node information is important apart from maintaining node data. Properties of the node determine the treatment that should be meted to the node. Here we deal with three types of nodes namely internal, peripheral and shadow nodes. Node types might change during the course of execution (due to repartitioning of the computational domain) and with that the information attached with the node needs to be changed as well.

Each node in the iterative application problem graph has a unique global ID signified by *global_ID*. A single character specifies the node type, internal or peripheral. The owning processor of the node is specified by *owning_proc*. A pointer to the user supplied node data structure is kept at **data_location*. The neighbors of the node are maintained in an array *neighboring_nodes[]*. Finally, the *shadow_for_procs[]* array specifies the processors for which the current node is a shadow, if at all. This array helps in setting up the communication buffers. By analyzing this array for each of its peripheral nodes, a processor exactly knows the neighboring processors it needs to communicate, and what to communicate. Note that a node is a shadow node for neighboring processors only if it is a peripheral node.

Separate lists are maintained for the internal and the peripheral nodes owned by processor. The advantage of maintaining the node information in separate lists is leveraged by the load balancing phase, where separate maintenance of these lists allows for easy updating of data structures (during task migration) related to node and thereby easy migration to neighboring processors. Here, it should be noted that no separate shadow node list is maintained by the processor. A processor is only concerned with the shadow node data which is easily accessible using the hash table which holds the location for the same.

```
struct own_node
{
        int global_ID;
         char internal_or_peripheral;
        int owning_proc;
        struct node_data *data_location;
        int neighboring_nodes[];
        int shadow_for_procs[];
        struct own_node *next_node;
};
```

Figure 7: Maintaining Node Information

Apart from node information, node data is also maintained in the form of a list. The global ID and the actual node data being the members of the structure used for the same. After computation, updated shadow node information is received by the processor, and this information is updated before proceeding to the next iteration of computation.

Besides these data structures, hash tables are also set up in the local memory of each processor. Hash tables are implemented as an array of pointers to sorted linked lists which contain the locations for node data. A modulo hash function is applied on the node global ID (key) to obtain the location for node data. The buckets (sorted linked lists) maintain pointers to the node data in the data node lists. This helps provide an amortized constant time access to the node data during computation without having to go through the entire data node list [PSC95]. Note that the Hash tables play a dual role. Firstly, it helps in accessing the node data during computation as mentioned, and secondly in the updating of the node data after communication. Hash tables help in easy access of the shadow node data from the data node list, which would be needed during the computation of the node.

4.2 Computation & Communication Phase

During this phase, the processor performs computations for each of its nodes using data of the neighboring nodes. After updating the node data, a processor sends the updated peripheral node data to appropriate neighboring processors and then receives the updated shadow node data from appropriate processors.

Figure 8 shows the computation and communication functions. For each node it owns, it forms a list comprising of the current node's data as the head followed by the data of the neighbors to be passed to the application node computation function. The platform maintains a pointer to the application node function supplied by the user. This allows for a clean and robust decoupling between the iC2mpi platform and the application program code.

Internal nodes are updated followed by the peripheral nodes. As the peripheral node data is updated, this updated data is packed into communication buffers even as the next peripheral node in the 'for loop' is ready to be updated. An array of pointers to an array of structures, one for each neighboring processor, is used for the communication buffers. This array is of the size equal to the number of neighboring processors with each element in the array pointing to the communication buffer (array of structures) to be sent to the processor corresponding to the index of that array element. From the initialization phase, a processor already knows the neighboring processors it needs to communicate the shadow node information to, and the number of such shadow nodes. Hence, the data structure chosen for the communication buffers gives optimum memory usage. By the time the computation routine returns, the communication buffers are all set up, and communication can proceed.

```
void ComputeOverNodes()
{
        for each internal node
        {
                form list of node and its neighbors;
                invoke application node function( );
        }

        for each peripheral node
        {
                form list of node and its neighbors;
                invoke application node function;
                pack updated peripheral node data into communication buffers;
        }
}

// Communication buffers set up at this stage.

void CommunicateShadows()
{
        for each neighboring processor
        {
                MPI_Isend(shadow nodes);
        }

        for each neighboring processor
        {
                MPI_Recv(shadow nodes);
        }
}
```

Figure 8: Computation & Communication Functions

For physical communication of these buffers, the structure type used to set up the buffers is committed to an MPI data type (using MPI_Type_commit), since it is not a standard MPI data type but a derived data type. All the processors send these buffers at one go. MPI_Isend(), a non-blocking call, is used for sending these buffers to appropriate processors. MPI_Recv() receives these buffers which are then used to update the locally maintained shadow node information.

A version of iC2mpi employs MPI_Irecv() to overlap the computations with communications, by processing the peripheral nodes first and dispatching the shadow nodes to neighboring processors, and while the communication takes place, proceeds with the processing of the internal nodes. The resulting computation and communication functions are shown in figure 8a. These and other performance enhancements are currently underway – the results reported in Section 4 employs the basic prototype of iC2mpi platform.

4.3 Load Balancing & Task Migration Phase

Dynamic applications require periodic load-balancing, since the computational workload of individual nodes may change throughout the course of computation. For dynamic applications, it is important to devote the available resources in sub-domains with higher computational complexity which varies with time. Hence, this phase is vital for the efficient parallelization of such applications. Repartitioning should not only balance the work load among the processors but also keep the edge-cut to a minimum, so as to minimize the inter-processor communication [KK98].

"Although, massively parallel computers can deliver impressive peak performances, their computational power is not sufficient to simulate physical problems with highly localized phenomena by using only brute force computations. Adaptive computations offer the potential to provide large increases in performance for problems

```
void ComputeOverNodes()
{
        for each peripheral node
        {
                form list of node and its neighbors;
                invoke application node function;
                pack updated peripheral node data into communication buffers;
        }
}

// Communication buffers set up at this stage.

void CommunicateShadows()
{
        for each neighboring processor
        {
                MPI_Isend(shadow nodes); // non-blocking call
        }

        for each neighboring processor
        {
                MPI_Irecv(shadow nodes); // non-blocking call
        }
}

void ComputeOverNodes()
{
        // Do the remainder of computation even as communication continues
        for each internal node
        {
                form list of node and its neighbors;
                invoke application node function;
        }
}

void CommunicateShadows()
{
        // Wait for MPI_Irecv() to finish
        MPI_Wait();
        Unpack the buffers;
}
```

Figure 8a: Computation & Communication Functions

with dissimilar physical scales by focusing the available computing power on the regions where the solution changes rapidly. [CS96]" This only reiterates the importance of this phase. Dynamic load balancing utilities incorporated in the framework make it robust even in face of domain applications with varying load requirements.

We now briefly describe a centralized heuristic algorithm used in the platform in order to attain dynamic load balancing. As mentioned a third party plug-in is possible here.

1) A weighted processor network graph is set up. The execution time of the processors for a specific number of iterations represents the weight on the nodes and the weight of the edge connecting two processors is the amount of communication between the two estimated by the length of the communication buffers.

2) A designated processor (chosen as the one with id = 0 in this case) examines the processor graph so as to measure the relative workload on the processors spread across the computational domain. The processor that does 25% more work (obtained from the node weights) than all its neighbors is considered to be the 'busy' processor. The one among its neighbors doing the least amount of work is labeled as an 'idle' one. The central processor obtains all such 'busy-idle' pairs from the processor graph.

3) As a final step, we just need to decide upon the task that should be migrated from the 'busy' processor to the 'idle' processor. The task which keeps the edge-cut to a minimum becomes the candidate for 'task migration.' For example, in Figure 9, out of the two nodes A and B which could be migrated from processor 0 to processor 1, we choose the migration of node B. Migrating node A would cause three edges to cross the

boundary of processor 0, which would increase the overall edge cut by 2, and so we choose B in this case.

The statistics obtained by the dynamic load balancer namely a set of ‘busy-idle’ pairs is then fed to the task migration routine, which then balances the work load among the processors. Now, we talk about the task migration routine which deals with the migration of a task from the ‘busy’ processor to an ‘idle’ one. We discuss, how exactly a task is migrated and the changes involved in the data structures of the concerned processors.

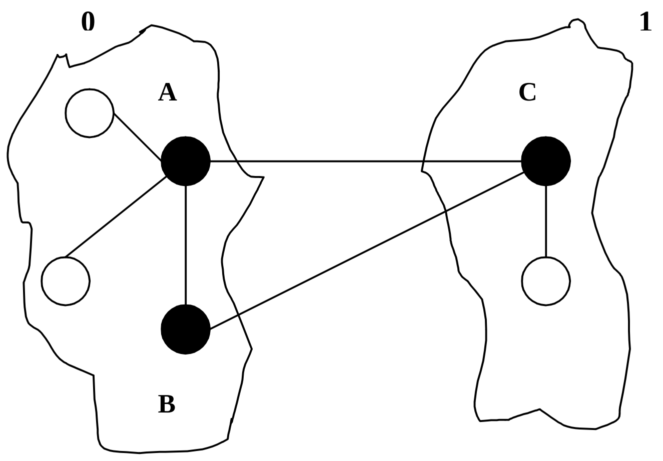

Figure 9: Choosing B over A for task migration, keeping edge-cut to a minimum

Task Migration: This phase repartitions the computational domain balancing the work load among the processors. A single task migration involves the ‘busy’ processor which sends the task, an ‘idle’ processor which receives the task, and a set of processors which hold shadows for the migrating node. Here, we talk about the changes in data structures & the send/receive operations at each of these 3 types of processors upon execution of task migration.

There are changes in the node information maintained by the ‘busy’ processor. The internal nodes with an edge to the migrating node would become peripheral nodes. Hence, such nodes need to be removed from the internal node list and added to the peripheral node list. The migrating node needs to be removed from the peripheral node list. Here, it should be noted that, the entry of the migrating node isn’t be removed from the data node list and the hash table, since, the migrating node now becomes a shadow node for the ‘busy’ processor, and it still needs to maintain its data for the computation of the other nodes owned by the same. Apart from these changes in the data structures, the ‘busy’ processor also sends the data of the neighbors of the migrating node to the ‘idle’ processor. This is needed since the neighbors of the migrating node now become shadow nodes for the ‘idle’ processor and it needs the same for the computation of the migrating node, by the time the next iteration of the computation occurs.

As far as the ‘idle’ processor is concerned, it receives the data of the neighbors of the migrating node sent from the ‘busy’ processor. Hash tables and data node lists would need to be updated to maintain the data just received. Apart from that, in this case, there is a possibility of the peripheral nodes becoming internal. So the internal and peripheral node lists are updated accordingly. Besides, the node information of the migrating node is added in the peripheral node list.

Finally, for the set of processors which hold the neighbors of the migrating node other than the ‘busy’ and ‘idle’ processors needs to update the maintenance of shadow node information. Now they receive the updated shadow node information of the migrating node from the ‘idle’ processor instead of the ‘busy’ processor due to the change in ownership of the migrating node.

We have considered migration of a single task. In our framework such migrations occur in parallel across the computational domain. For dynamic applications, the data redistribution cost for load balancing can be comparable to the actual time for computation [SKK00]. Further, it is only apt to leverage the maximum out of the overhead incurred by the load balancing routine to gather the load statistics across the computational domain. Figure 10 which show employment of task migration in parallel helps achieve just this.

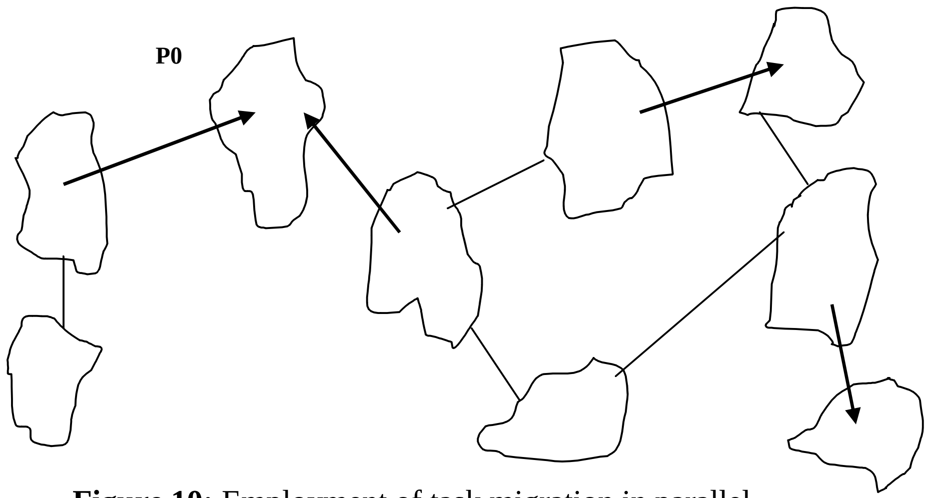

Figure 10: Employment of task migration in parallel

Here, it should be mentioned that there’s no limiting condition for the parallel execution of task migration. It is possible for a processor to be a part of more than one migration. Yet the migrations can take place in parallel. To understand this consider table 1. A processor with respect to a single migration can be either a ‘busy’ processor, or an ‘idle’ processor or belongs to the set of processors which hold the shadow for the migrating node.

Processor	‘Busy’	‘Idle’	Holding Shadow for the Migrating Node
Busy	Y	N	N
Idle	N	Y	Y
Holding Shadow for the Migrating Node	N	Y	Y

Table 1: Possible state of the processors during the parallel execution of task migration.

When a processor for a particular migration is a ‘busy’ processor, it cannot be either ‘idle’ or holding shadow for the migrating node of any other migration. From, the centralized heuristic algorithm we know that a processor is designated as ‘busy’ when it does 25% more work than all its neighbors. When, that’s the case, there’s no question of it receiving any task from any of its neighbors. Further, since the neighbors of the ‘busy’ processor do not migrate any task for the current run (since, they do less work than the ‘busy’ processor), the ‘busy’ processor also cannot in the third category of processors, namely the set of processors holding shadows for the migrating node. On the basis of similar reasoning, we obtain the values for the second and third rows of table 1. So, the two cases (corresponding to the 2nd and 3rd row of table 1) when a processor is a part of two migrations, it handles those sequentially. Further, it is possible, that a processor might be receiving two tasks from two different processors during the same run (P0 in figure 10). The processor would then handle such situations sequentially (receive tasks from the neighboring processors one after the other).

5. Experimental Results

The experimental results were conducted on Silicon Graphics Origin-2000 computer with 24 CPUs. It has CRAY link high bandwidth interconnects, and is based on hypercube cc-NUMA architecture. Two generic applications with hexagonal grid and random graph topologies, and one existing application, namely battlefield management simulation [DMP98], are parallelized using the iC2mpi platform. In each case, the application node function and data structure was prepared and plugged into iC2mpi platform with relative ease.

5.1 Hexagonal Grids

We tested the performance of the platform on hexagonal grids. Different size hexagonal grids are used to deploy on the platform using different numbers of processors.

Each node computes the average of the data maintained by all its neighbors. A dummy 'for loop' is used to inject a grain size of about 0.3 ms for fine grain and 3 ms for coarse grain on the nodes.

Tables 2, 3 and 4 show the overall runtimes in seconds for parallelizing these grids on the platform using Metis for the initial node-to-processor mapping. Most data corresponds to average over five different fine grained graphs, unless specified otherwise. Figure 11 shows the speedup plots for 20 iterations on 32, 64 and 96-node hexagonal grids deployed on the iC2mpi platform. In figure 12, we compare the performance of Metis [KK98] and PaGrid [WA04, HAB06]. We obtain the speedup plots using fine and coarse grain computation for the nodes. This figure shows the speedup plots for 20 iterations for 64-node hexagonal grid using Metis and PaGrid as the static graph partitioners. As expected, we see that coarse grain node computation results in considerably better speedups as compared to the fine grain node computation.

Iterations	Processors				
	1	2	4	8	16
10	.111	.0580	.0315	.0191	.028
15	.165	.085	.0462	.027	.035
20	.209	.113	.0605	.0435	.0434

Table 2: Execution Time (in seconds) on 32-node Hexagonal Grids

Iterations	Processors				
	1	2	4	8	16
10	.218	.113	.0708	.0348	.039
15	.344	.178	.092	.0585	.056
20	.458	.236	.136	.0829	.0638

Table 3: Execution Time (in seconds) on 64-node Hexagonal Grids

Iterations	Processors				
	1	2	4	8	16
10	.3528	.177	.0912	.0603	.052
15	.527	.254	.135	.0809	.071
20	.7016	.352	.180	.106	.085

Table 4: Execution Time (in seconds) on 96-node Hexagonal Grids

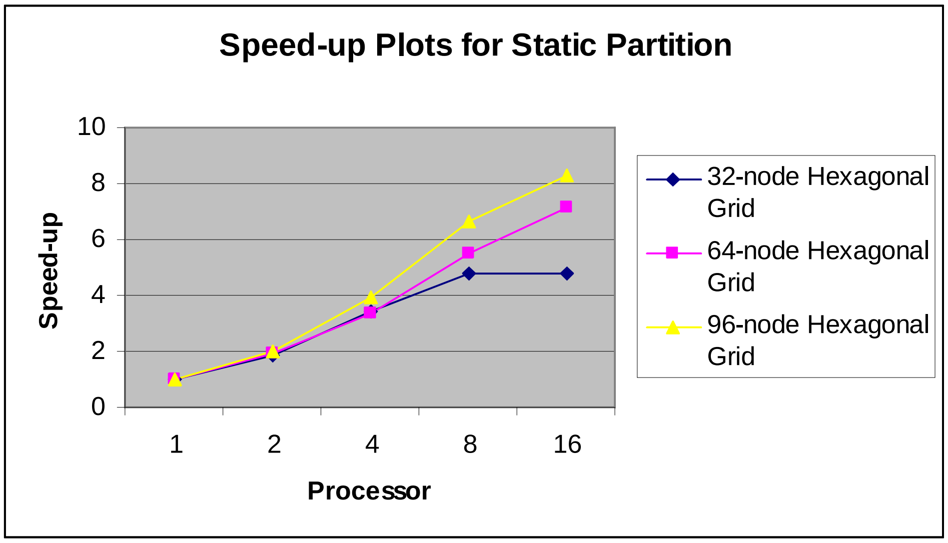

Figure 11: Speedup for Hexagonal Grids using Metis

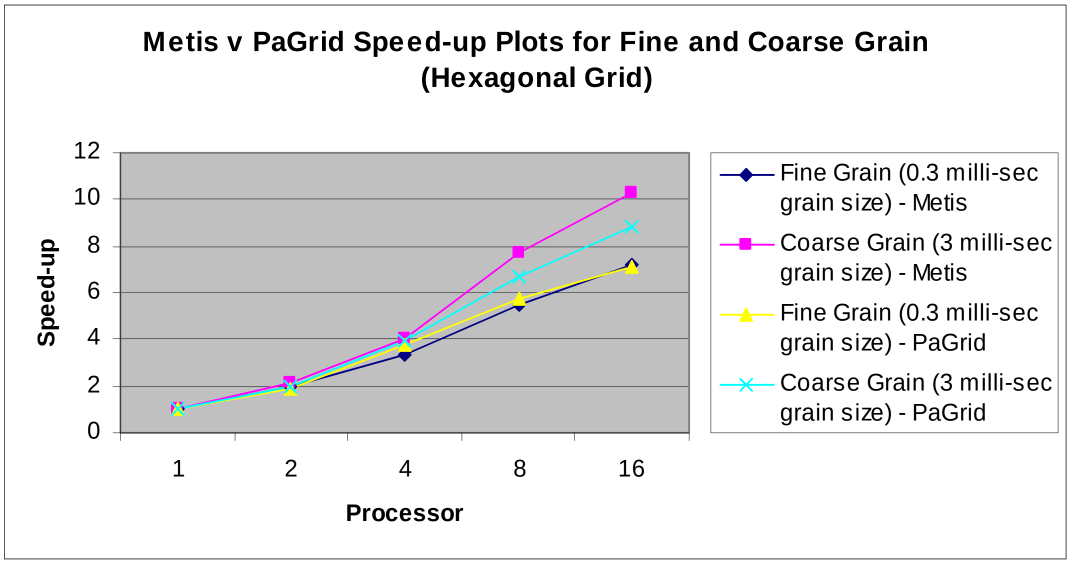

Figure 12: Metis vs PaGrid for Fine & Coarse Grained 64-node Hexagonal Grids

We employed Chaco format [KK98] for the application program graph as input to the partitioners employed, namely Metis and PaGrid. In addition, the processor network graph used for PaGrid (hypercube) used the grid format specified in [WA04, HAB06]. Of particular interest is the "Rref" parameter for PaGrid, defined as the ratio of communication time to the computation time per node in the application graph. For the graph topologies discussed here, we used Rref = 0.45. For Metis, we set the parameter fmt=0, indicating uniform weighted program graph; Metis does not use processor network graph [KK98].

Next, we compare the performance of the platform with and without dynamic load balancing utility. Figures 13, 14 and 15 show the comparison plots for the static and dynamic load balancing utilities of the platform. Dynamic load imbalance is created by varying the grain size for the node throughout the course of computation which cannot be captured by the static partition. Speedups are measured for 25 iterations and load balancing routine is invoked every 10 time steps. Dynamic load balancing is better, even for finer grained grids.

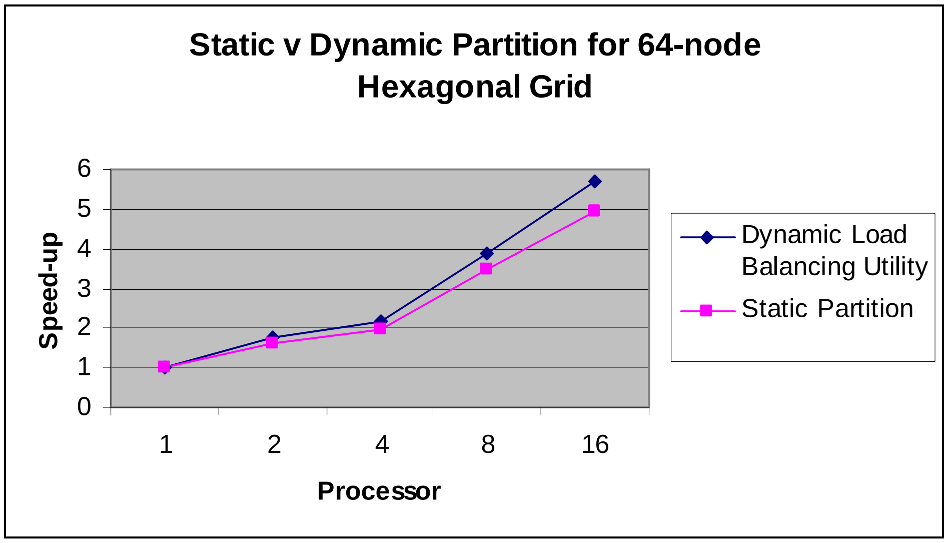

Figure 13: Static vs. Dynamic Partitioning on 64-node Hexagonal Grids

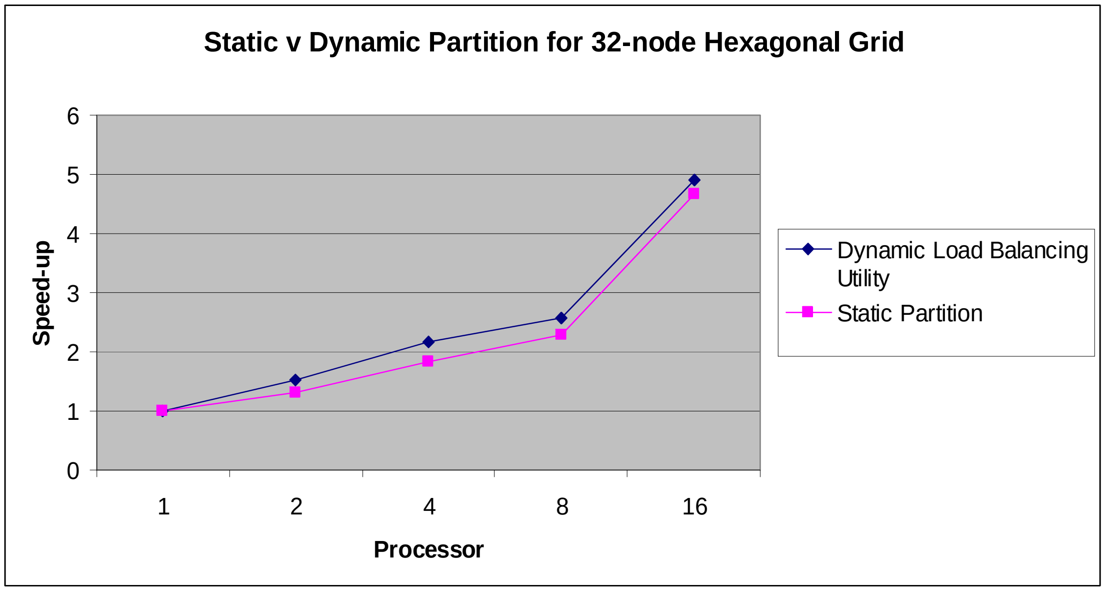

Figure 14: Static vs. Dynamic Partitioning on 32-node Hexagonal Grids

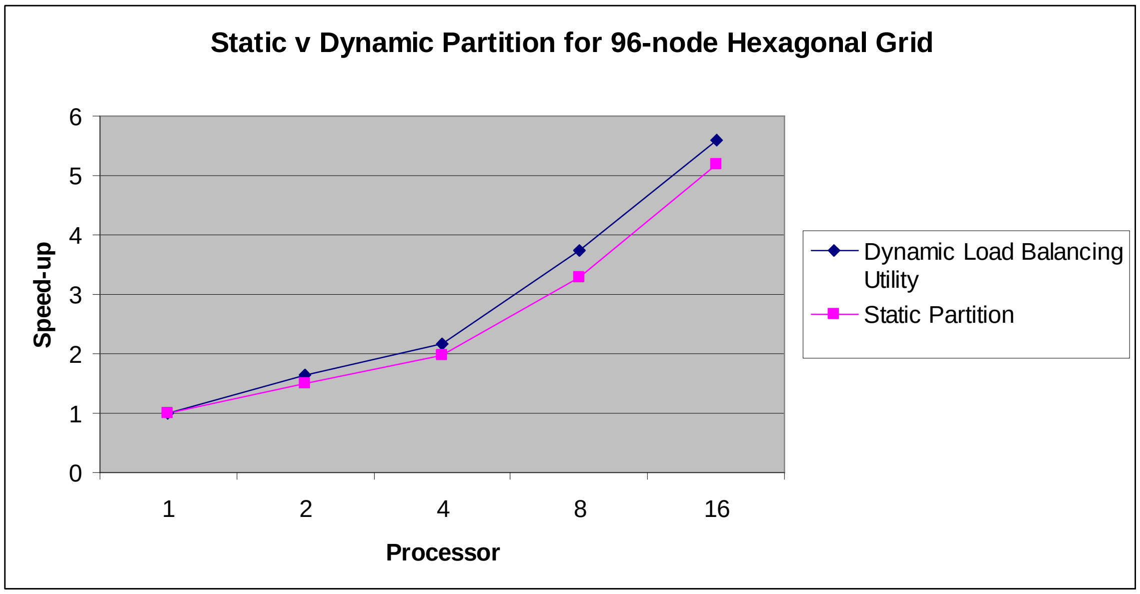

Figure 15: Static vs. Dynamic Partitioning on 96-node Hexagonal Grids

5.2 Random Graphs

We also experimented with random graphs on the iC2mpi platform. As in case of Hexagonal grids, even here each node computes the average of the data maintained by all its neighbors. A dummy 'for loop' is used to inject a grain size of about 0.3 ms for fine grain and 3 ms for coarse grain on the nodes.

Tables 5 and 6 give the overall runtimes in seconds with static partition on 32 and 64-node random graphs using different number of iterations and processors. Metis is used as the initial static graph partitioner for both the case. The readings are an average over 5 runs.

Iterations	Processors				
	1	2	4	8	16
10	.108	.056	.030	.020	.035
15	.161	.082	.045	.037	.044
20	.215	.109	.059	.046	.049

Table 5: Execution Time (in seconds) on 32-node Random Graphs

Iterations	Processors				
	1	2	4	8	16
10	.218	.111	.064	.050	.051
15	.325	.167	.095	.059	.067
20	.434	.221	.126	.073	.083

Table 6: Execution Time (in seconds) on 64-node Random Graphs

Figure 16 shows the speedup plots for 20 iterations for 32 and 64-node random graphs. Note, that the speed-up dips slightly when the number of processors increases from 8 to 16. Fine grain size used for the node computation in this case is partially responsible for that. Using coarse grain size for node computation, we obtain better speed-ups as expected.

In Figure 17, we compare the performance of Metis and PaGrid for 64-node random graphs. We obtain the speedup plots using fine and coarse grain size for these graphs for 20 iterations. Note that unlike hexagonal grids, PaGrid outperforms Metis for random graphs, both for fine and coarse grains. Even the fine grain speed-ups for PaGrid are better than the coarse grain speed-ups for Metis.

Figures 18, 19 show the performance comparison of random graphs with and without dynamic load balancing utility. Speedups are calculated for 25 iterations, and load balancing routine is invoked every 10 time steps for the dynamic load balancing approach. The former approach beats the latter even with fine grain used for node computation.

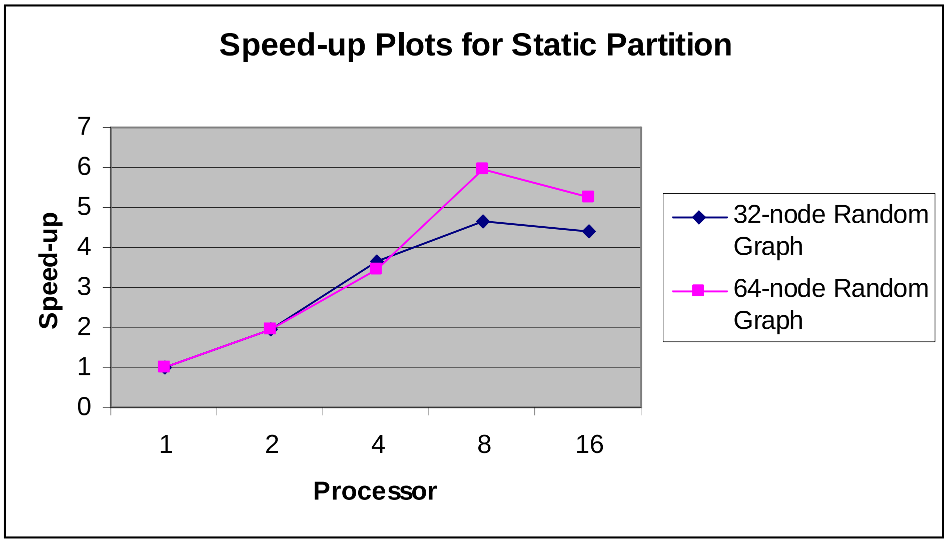

Figure 16: Speedup for Random Graphs with Static Partition (Metis)

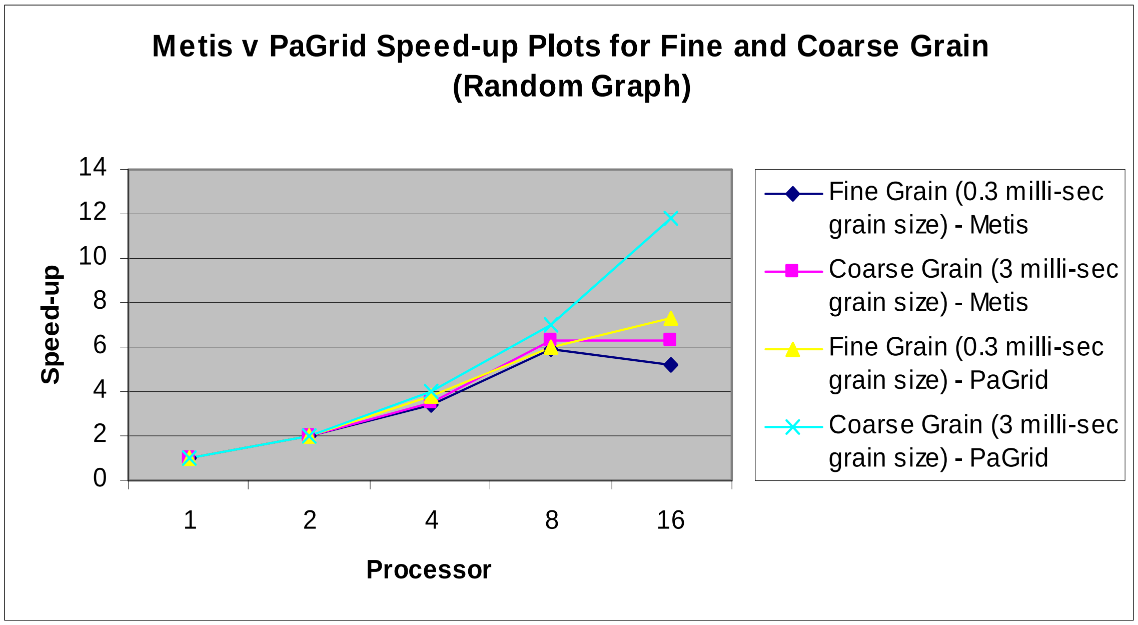

Figure 17: Metis vs. PaGrid on Fine & Coarse Grained 64-node Random Graphs

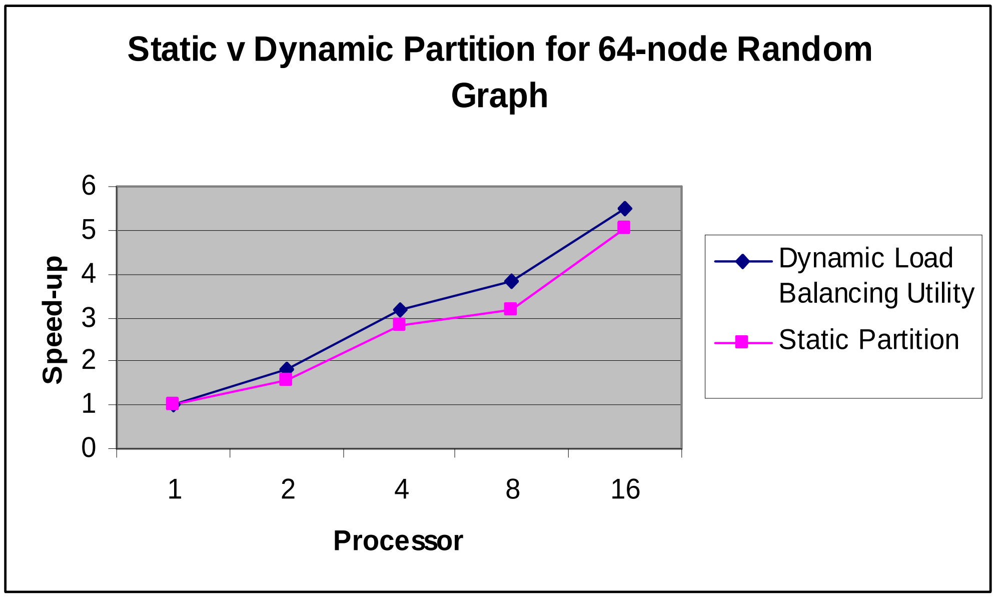

Figure 18: Performance of Dynamic Partitioning on 64-node Random Graphs

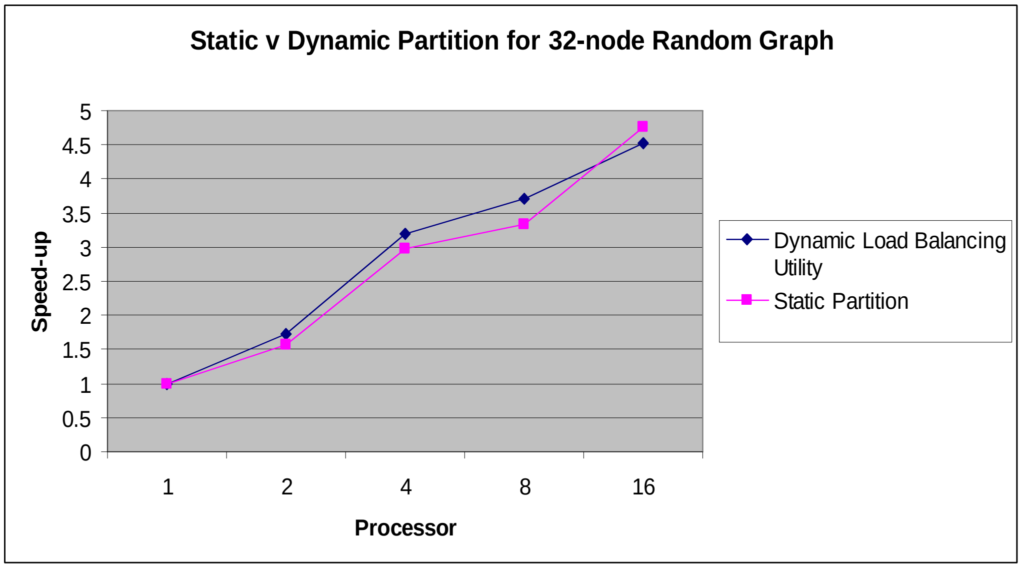

Figure 19: Performance of Dynamic Partitioning on 32-node Random Graphs

5.3 Battlefield Management Simulations

Battlefield management simulations are employed to evaluate doctrine and tactics, and to predict the battle outcome in various scenarios [DMP98]. These are amongst the most computationally intense and complex simulations in existence. We show the test results using different patterns of initial partitioning. In essence, the platform proves to be an apt test-bed for the study of different static partitioning schemes.

The experiments were conducted on a 32 x 32-hex battlefield (computational domain is divided into hexes and hex-to-processor assignment is used for initial partitioning), varying the number of iterations and the number of processors using different static partitioning schemes.

The different algorithms employed for the initial static partitioning include (i) Metis [KK98], (ii) gray-code-based mesh-to-hypercube fine-grained embedding [DMP98] wherein a hex and its six neighbors are allocated to different processors, (iii) row band, (iv) column band, and (v) rectangular band partitionings. Tables 7, 8, 9, 10 and 11 show the overall runtimes in seconds for parallelizing the battlefield simulation deployed on iC2mpi platform. Figure 20 shows the speedup plots. From the figure it is evident that Metis easily outperforms the rest of the static partitioning algorithms.

Simulation Steps	Processors				
	1	2	4	8	16
5	.684	.654	.537	.461	.390
15	1.463	1.447	1.109	.869	.623
25	2.248	2.245	1.666	1.265	.847

Table 7: Execution Time (in seconds) of Battlefield Simulator using Metis

Simulation Steps	Processors				
	1	2	4	8	16
5	.681	1.360	.926	.645	.454
15	1.410	3.578	2.279	1.413	.814
25	2.255	5.752	3.627	2.166	1.164

Table 8: Execution Time (in seconds) of Battlefield Simulator using Fine-Grained Mesh-to-Hypercube Embedding (BF Partition)

Simulation Steps	Processors				
	1	2	4	8	16
5	.680	.756	.606	.507	.467
15	1.456	1.780	1.347	1.006	.854
25	2.226	2.781	2.057	1.502	1.229

Table 9: Execution Time (in seconds) of Battlefield Simulator using Row Band Partition

Simulation Steps	Processors				
	1	2	4	8	16
5	.679	.666	.543	.465	.453
15	1.463	1.463	1.112	.887	.820
25	2.242	2.245	1.689	1.286	1.168

Table 10: Execution Time (in seconds) of Battlefield Simulator using Column Band Partition

Simulation Steps	Processors				
	1	2	4	8	16
5	.682	.663	.591	.503	.404
15	1.456	1.465	1.260	.981	.679
25	2.243	2.247	1.932	1.464	.950

Table 11: Execution Time (in seconds) of Battlefield Simulator using Rectangular Partition

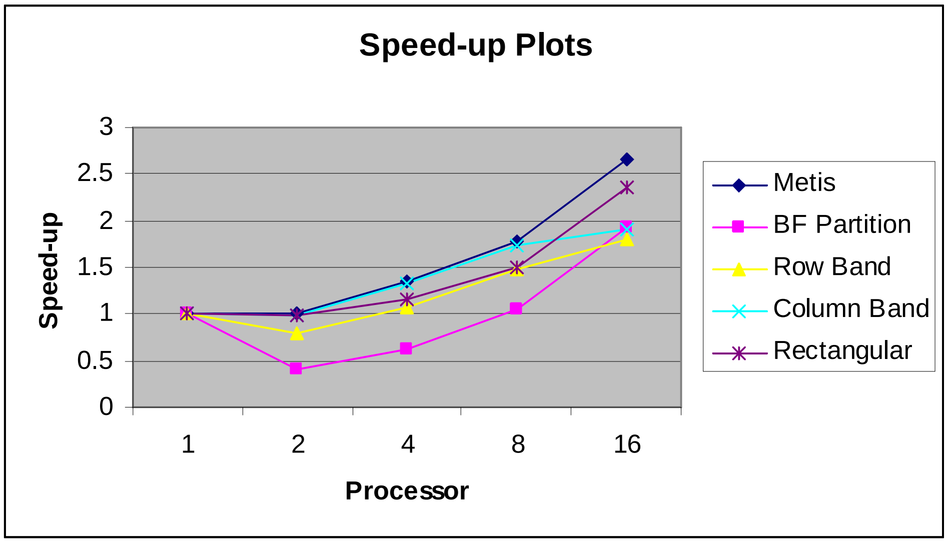

Figure 20: Performance of Battlefield Management Simulation for different Static Partitioning Algorithms

5.4 Measurement of Overheads

We measured the current overheads of the iC2mpi platform's various phases. We plot the overheads for fine-grained 64-node hexagonal grids and 64-node random graphs varying the number of processors for 35 iterations (dynamic load balancer invoked every 10 time steps) in figures 21 and 22. Here, the phases and overheads comprise:

(i) Initialization: Includes setting up the data structures for graph connectivity, node-to-processor mapping, internal and peripheral node lists, data node lists and hash tables.

(ii) Computation Overhead: Setting up the list of the current node and its neighbors to be passed on to the application node function for node computation, and updating of the data node lists after computation.

(iii) Compute: Actual node computation.

(iv) Communication Overhead: Packing buffers for communication, unpacking the received buffers and updating the data node lists. This is clearly the most significant source of overhead.

(v) Communicate: Communication of the shadow node information to appropriate processors and receiving the required information from neighbors.

(vi) Load Balancing & Task Migration: Gathering information about load imbalance across the computational domain and balancing the work load among the processors.

Note, that the compute and computation overhead comes down with the number of processors as it should for both the cases.

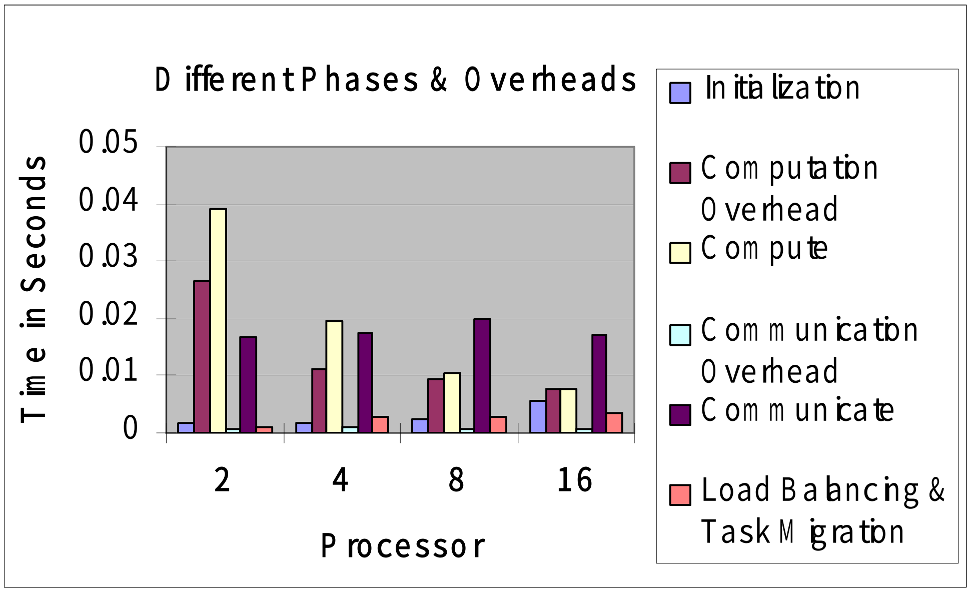

Figure 21: Overheads in iC2mpi Platform for fined grained (0.3ms) 64-node Hexagonal Grids

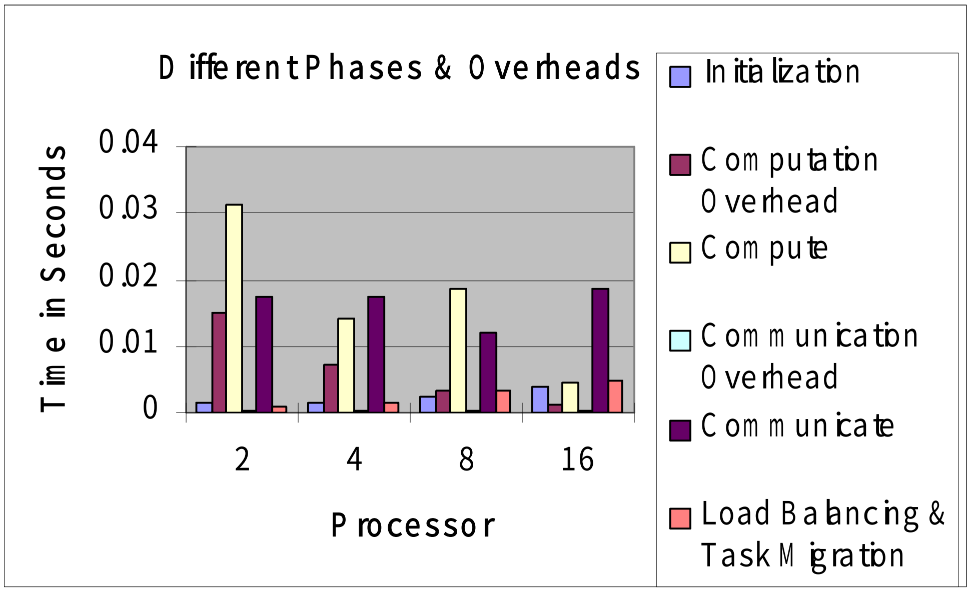

Figure 22: Overheads in iC2mpi Platform for fined grained (0.3ms) 64-node Random Graphs

5.5 Creation of Dynamic Load Imbalance

In this section we talk about the creation of dynamic load imbalance, which was used for testing the static v dynamic partitioning performances. We created dynamic load imbalance, which couldn't have been captured by static partitioner in any way, and that's where the dynamic load balancer scores over it. Here, we talk about the creation of dynamic load imbalance on a 64-node graph. Figure 23 shows how the grain size for the node was varied. Grain size was varied for nodes across the computational domain throughout the course of execution.

```
if (iteration_num <= 10)
{
        if (node globalID in the first 50%)
                do more work; // coarse grain size
        else
                do less work;   // fine grain size
}
else if (iteration_num <= 20)
{
        if (node globalID between 25% to 75%)
                do more work; // coarse grain size
        else
                do less work;   // fine grain size
}
else if (iteration_num <= 30)
{
        if (node globalID beween 50% and 100%)
                do more work; //coarse grain size
        else
                do less work;   // fine grain size
}
```

Figure 23: Varying the grain size of the node for creating dynamic load imbalance

Note, that the dynamic load balancer is invoked every 10 time steps, and the total number of iterations amount to 35. Each time the dynamic load balancer is invoked, we try and create an inertial load imbalance across the computational domain. This imbalance is captured by the dynamic partitioner, while there's no way a static graph partitioner could capture this phenomena.

6. Related Work

Existing alternatives to time-consuming manual parallelization with MPI for application programmers consist of the use of compiler directives to assist the compiler in parallelizing sections of a sequential program. This approach has mainly been implemented in High Performance Fortran [Z02] for multicomputers and OpenMP [OP] for multiprocessors. The former has had limited success due to its limited expressivity for a broad range of scientific applications. While OpenMP has been relatively successful, it is mainly limited to expensive multiprocessor platforms, can produce non-deterministic programs that are hard to debug, and requires significant work (sometimes comparable to MPI) to achieve highly scalable programs. By restricting itself to iterative graph-based applications, our platform can achieve good performance for a minor time investment on the user's part.

Other platforms to assist parallelization do exist. Here, we discuss two in particular, namely Zoltan [DBH02] and CO_2P_3S [MSS00], "the correct object-oriented pattern-based parallel programming system," which come close to our goals. Some of the other related frameworks include the Armada framework [OK02], which is an I/O framework that allows data intensive applications to efficiently access geographically distributed data sets, and TOP-C [PSC95], a package with its own MPI subset to parallelize sequential applications.

6.1 Zoltan

Zoltan is a library of data management services for parallel, unstructured and dynamic applications. Zoltan basically simplifies load-balancing, data movement, unstructured communication and memory usage difficulties that arise in dynamic applications such as adaptive finite-element methods.

Zoltan provides utilities for the efficient functioning of an already parallel application, while our platform does not need the iterative application to be already parallelized. Zoltan uses call-back functions to interface with the application. These functions query the application for the needed data, and the application in turn must provide functions to answer those queries. Our iC2mpi platform, on the other hand, does not require any coding on the part of the user, nor should the user be concerned about the system architecture. But the user needs to model the iterative application as per the platform specification. Zoltan provides utilities like distributed data directories for locating off-processor data and a communication library to incorporate changing communication patterns for dynamic applications. In our platform, these utilities are built-in, so the user need not worry about them. All in all, Zoltan provides for efficient management of an already parallelized application, while iC2mpi platform aims at easy parallelization of the iterative applications and their efficient runtime management.

6.2 CO_2P_3S

It is a parallel programming system that combines three abstraction techniques, namely, object-oriented programming, design patterns, and frameworks, using a layered programming model which supports fast development of parallel programs and fine tuning of the resulting programs for performance. CO_2P_3S uses design patterns to ease the effort required to write parallel programs [MSS00].

While CO_2P_3S emphasizes easy parallelization of a range of applications based on user selection of pattern templates and fine-tuning of the parallel programs for performance, there are no dynamic load balancing capabilities incorporated in the mesh framework. While the approach would be a reasonable solution for domain applications with more or less the same load across the computational domain throughout the course of computation, it may not be effective for unstructured dynamic applications with varying load requirements. CO_2P_3S uses a design pattern approach to generating a framework which is a set of classes that implements the basic structure of the mesh computation. Instead of directly adding code to the framework, the user creates subclasses of the framework classes to provide the application-specific details. While CO_2P_3S does well to separate the application-independent framework structure from application-dependent code, it does not ensure that the mesh framework employed by the same is a black box from the point of view of the user, whereas in our iC2mpi platform, the user needs to model the iterative application as per the platform specification without worrying about its architecture.

7. Design Enhancements

In this section we talk about the possible design enhancements in the current platform design.

To begin, in order to realize easy parallelization of domain applications, one of the goals of this project, the user plug-in points need to be made crisper. Currently, there's no user interface for the user to plug-in data structures and node computation function. Platform maintains a pointer to the latter, more detailing needs to be done on the former. Currently, the platform design is good enough to incorporate varying domain data structures, but more work needs to be done on that. Platform should be able make certain template data structures independent of the domain data structures. This would make the platform flexible enough for various domain applications. Currently, this is possible but not without the customization of the data structures used by the platform.

Change in design of the computation and communication phase is desirable. While, the platform does attempt to minimize the overheads for these phases, there's clearly no overlap between these phases. A different version using MPI_Irecv() (currently underway, see figure 8a) could result in significant performance improvement for applications with unstructured communication and possibly coarse grain size for the node. Communication overhead and the actual communication can be a sizeable proportion of the total runtime. This makes it that much more important to deal with it efficiently.

Platform employed a centralized heuristic algorithm as a dynamic load balancer. Interface for the same needs to be made cleaner so as to make third party plug-ins for the dynamic load balancer much easier so as to study the performance of various dynamic load balancers. Improvements are also possible in the implementation of task migration. Though, the task migration executes in parallel, it doesn't send more than one task from a 'busy' processor to an 'idle' one. A more rigorous algorithm can be used here, which would specify the number of tasks that should be migrated from a 'busy' processor to an 'idle' one.

All in all, the basic prototype of the platform is set up. Further, design enhancements and performance fine-tuning could make it more versatile. Now, we talk about the possible future extensions for the platform.

7.1 Future Extensions:

While, the Battlefield Management Simulation was parallelized using static graph partitioner, it would be interesting to see the performance of the platform while parallelizing the same with the dynamic load balancer utilities. More domain applications need to be tested on the platform so as to realize one of its main goals, which is easy parallelization of domain applications which would help the user avoid all hassles related to MPI programming for parallelization of iterative sequential applications.

There's a possibility of incorporating library routines to deal with load balancing, unstructured communication etc which could be invoked as per the user requirements. Distributed data directory could be built which would help the processor locate off-processor data. Currently, the processor is able to get all the required shadow node information, but by the use of distributed directories, it might have a possible access to the data of far off processors (which are not neighbors of the current processor). This concept might be useful for certain types of domain applications.

Finally, various static graph partitioners were employed to study the performance of those algorithms. Similarly, different dynamic load balancers need to be tested for a more comprehensive study on the dynamic partitioning on domain applications and comparing the performance of static and dynamic load balancing utilities.

8. Conclusions and Future Work

We have presented a unique proof-of-concept prototype platform for parallelization of iterative graph-structured applications. It provides a relatively easy transition from sequential programs to their distributed executions, and facilitates experimentation with static partitioning and dynamic load balancing schemes. We demonstrated with two generic iterative applications with underlying hexagonal and random graph structures, and a time-stepped battlefield simulation, that our platform can produce good performance with very little effort. The iC2mpi platform has good potential for further improvements and extensions. Future work will include improving the performance of our platform by reducing its overheads, and extending it to adaptive mesh-based applications. We will also explore extending it to applications that use the BSP model [HMS98], as this model essentially divides the computation from communication phases as iC2mpi does. Finally, we will employ the platform to perform comprehensive evaluation of static and dynamic partitioners.

9. Bibliography

[C96] G. Cooperman, “TOP-C: A Task-Oriented Parallel C Interface”, *International Symposium on High Performance Distributed Computing* (HPDC-5), 1996, IEEE Press, pp. 141-150.

[CCE01] R. Cappuccio , G. Cattaneo , G. Erbacci , U. Jocher, A parallel implementation of a cellular automata based model for coffee percolation, *Parallel Computing,* v.27 n.5, p.685-717, April 2001

[CS96] Jose G. Castanos and John E. Savage, *“The Dynamic Adaptation of Parallel Mesh-Based Computation”*, Technical Report: CS-96-31, 1996.

[DBH02] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson and Courtenay Vaughan, “Zoltan Data Management Services for Parallel Dynamic Applications”, *Computing in Science & Engineering,* Vol. 4, No. 2, March/April 2002, pp. 90-97.

[DMP98] Narsingh Deo, Muralidhar Medidi and Sushil K Prasad, “Load balancing in parallel battlefield management simulation on local- and shared-memory architectures”, *J. Computer Systems: Science & Engineering,* Special Issue on ‘Simulation in parallel and Distributed Computing Environments’, Guest Editor: A. Zomaya, Vol. 13, No. 1, pp. 55-65, 1998.

[HAB06] S. Huang, E. Aubanel, and V.C. Bhavsar, "PaGrid: A Mesh Partitioner for Computational Grids", *Journal of Grid Computing,* Vol. 4 No. 1, pp. 71 - 88, 2006.

[HMS98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. *BSPlib: The BSP Programming Library*. Parallel Computing, 1998, pp. 1947-80.

[HMP03] R. Henderson, D. Meiron, M. Parashar, R, Samtaney, "Parallel Computing in Computational Fluid Dynamics". In J. Dongarra et al., editors, *"Sourcebook of Parallel Computing",* Chapter 5, Morgan Kaufmann, 2003.

[K90] Kauranne, T., 1990: An introduction to parallel processing in meteorology. *The Dawn of Massively Parallel Processing in Meteorology,* G. R. Hoffman and D. K. Maretis, Eds., Springer-Verlag, 3–20.

[KK98] George Karypis and Vipin Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs", *J. Parallel & Distributed Computing*, 48(1): 96-129, 1998.

[MSS00] Steve MacDonald, Duane Szafron, Jonathan Schaeffer and Steven Bromling, "Generating Parallel Program Frameworks from Parallel Design Patterns", in proceedings of *6th International Euro-Par Conference* (Euro-Par 2000), Munich, Germany, August 2000, Lecture Notes in Computer Science 1900, Springer-Verlag, pages 95-104.

[OK02] Ron Oldfield and David Kotz, "Armada: A Parallel I/O Framework for Computational grids", *Future Generation Computing Systems* (FGCS), 18(4):501-523, March 2002.

[OP] www.openmp.org

[PSC95] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang and Geoffrey Fox, "Runtime Support & Compilation Methods for User-Specified Irregular Data Distributions", *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[Q04] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[SG04] I.M. Smith and D.V. Griffiths : *Programming the finite element method,* (John Wiley & Sons, 2004, 4th edn.)

[SKK00] Kirk Schloegel, George Karypis and Vipin Kumar, "A Unified Algorithm for Load-balancing Adaptive Scientific Simulations", *Supercomputing, 2000.*

[SM02] Dale Shires and Ram Mohan, "An Evaluation of HPF & MPI Approaches and Performance in Unstructured Finite Element Simulations", *Journal of Mathematical Modeling and Algorithms* 1:153-167, 2002.

[WA04] R. Wanschoor and E. Aubanel, “Partitioning and Mapping of Mesh-Based Applications onto Computational Grids”, *5th IEEE/ACM International Workshop on Grid Computing (Grid 2004),* Nov. 8, 2004, Pittsburgh, PA, USA, IEEE Computer Society, pp. 156-162.

[WC01] C. Walshaw and M. Cross, "Multilevel Mesh Partitioning for Heterogeneous Communication Networks", *Future Generation Computer Systems,* 17(5):601-623, 2001.

[Z02] H. P. Zima, “High Performance Fortran - History, Status and Future”, *Proceedings of the 4th international Symposium on High Performance Computing* (May 15 - 17, 2002), Lecture Notes In Computer Science, vol. 2327. Springer-Verlag, London, 490, 2002.

Appendix A

This appendix holds the procedures for the platform's Initialization Phase.

```
/* Below listed procedures initialize data structures in the local memory of the processors to maintain node data and
node information*/

void InitializeGraph(int *num_of_vertices, int *num_of_edges, int *fmt)
{
        FILE *f_input_file;
        char line[MAX_STR_LENGTH],ch[10];
        int graph_parameters[3],k=0,i=0,j=0,flag=0;

        f_input_file = fopen("64_r_in.txt","r");

        if (f_input_file == NULL)
        {
                printf("Cannot open input file!");
                exit(1);
        }

        /* Getting the number of vertices, number of edges and the fmt parameters for the input graph */

        fgets(line,MAX_STR_LENGTH,f_input_file);

        while (line[i] != '\0')
        {
                while (line[i]==' ')
                {
                        i++;
                }

                while (line[i] != ' ' && line[i] != '\0')
                {
                        ch[j++] = line[i++];
                }

                ch[j]='\0';
                graph_parameters[k++]=atoi(ch);
                j=0;
        }

        *num_of_vertices = graph_parameters[0];
        *num_of_edges = graph_parameters[1];
        *fmt = graph_parameters[2];

        fclose(f_input_file);
}

void InitializeInputArray(int **input_arr, int num_of_vertices, int fmt, int *vertex_weight)
{
        FILE *f_input_file;
        char line[MAX_STR_LENGTH],ch[10];
        int i,j,k=0,l,cnt=0,m=0;

        for (i=0;i<num_of_vertices;i++)
        {
                for (j=0;j<COLUMN_WIDTH_OF_INPUT_ARRAY;j++)
                {
                        input_arr[i][j]=0;
                }
        }

        for (i=0;i<num_of_vertices;i++)
        {
                vertex_weight[i]=0;
        }

        f_input_file = fopen("64_r_in.txt","r");

        if (f_input_file == NULL)
        {
                printf("Cannot open input file!");
                exit(1);
        }

        fgets(line,MAX_STR_LENGTH,f_input_file); /* Bypass the first line with number of vertices, edges...etc */

  /* When fmt = 0, we're dealing with an unweighted graph,
                        fmt = 1, we're dealing with a weighted graph with weights on edges,
                        fmt = 10, we're dealing with a weighted graph with 'single' weight on vertices
                        fmt = 11, we're dealing with a weighted graph with weights on both vertices and
edges...*/

        while (fgets(line,MAX_STR_LENGTH,f_input_file))
        {
                i=0,j=0,l=0,cnt=0;
                while (line[i]!='\0')
                {
                        while (line[i] == ' ')
                        {
                                i++;
                        }

                        while (line[i] != ' ' && line[i] != '\0')
                        {
                                ch[j++] = line[i++];
                        }

                        cnt++;
                        ch[j] = '\0';
                        j=0;

                        if (fmt == 0)
                        {
                                input_arr[k][l++] = atoi(ch);
                        }

                        if ((fmt==1)&&((cnt%2)!=0))
                        {
                                input_arr[k][l++] = atoi(ch);
                        }

                        if ((fmt==11)&&((cnt%2)==0))
                        {
                                input_arr[k][l++] = atoi(ch);
                        }

                        if ((fmt==11)&&(cnt==1))
                        {
```

```
                        vertex_weight[m++] = atoi(ch);
                    }

                    if ((fmt==10)&&(cnt!=1))
                    {
                        input_arr[k][l++] = atoi(ch);
                    }

                    if ((fmt==10)&&(cnt==1))
                    {
                        vertex_weight[m++] = atoi(ch);
                    }
                }

                k++;
        }

        fclose(f_input_file);
}

void InitializeOutputArray(int *output_arr, int num_of_vertices)
{
        FILE *f_output_file;
        int i,j,k=0;
        char ch[3],line[MAX_STR_LENGTH];

        for (i=0;i<num_of_vertices;i++)
        {
                output_arr[i]=0;
        }

        f_output_file = fopen("64_r_out_16p.txt","r");

        if (f_output_file == NULL)
        {
                printf("Cannot open input file!");
                exit(1);
        }

        while (fgets(line,MAX_STR_LENGTH,f_output_file))
        {
                i=0,j=0;
                while (line[i]!='\0')
                {
                        ch[j++] = line[i++];
                }

                ch[j]='\0';
                output_arr[k++] = atoi(ch);
        }

        fclose(f_output_file);
}

void InitializeGlobalDataList(struct node_data **global_data_head, int num_of_vertices)
{
        int i=1,j=0;
        struct node_data *temp, *r;

        temp = *global_data_head;

        if (*global_data_head == NULL)
        {
                temp = malloc(sizeof(struct node_data));
                temp -> globalID = 1;
                temp -> data = 1;
                temp -> next_node = NULL;

                *global_data_head = temp;
        }

        while (j<num_of_vertices-1)
        {
                r = malloc(sizeof(struct node_data));
                r->globalID = i+1;
                r->data = i+1;
                r->next_node = NULL;

                temp->next_node = r;
                temp = r;
                i++;
                j++;
        }
}


/* This function peforms the initialization of node lists, namely the internal and peripheral node lists...*/

void InitializeNodeLists(struct own_node **internal_head, struct own_node **peripheral_head, struct node_data
**global_data_head, struct hash_node *hash_table[], int **input_arr, int *output_arr, int num_of_vertices, int fmt, int
*buffer_size_for_communication, MPI_Comm comm)
{
        int myid,i,j,k,flag_node_check,insert_shadow_flag,hash_map;
        struct own_node *internal_temp, *peripheral_temp, *r_internal, *r_peripheral;
        struct node_data *temp_data_node;
        struct hash_node *hash_temp, *r_hash;

        internal_temp = *internal_head;
        peripheral_temp = *peripheral_head;
        temp_data_node = *global_data_head;

        MPI_Comm_rank(comm,&myid);

        for (i=0;i<num_of_vertices;i++)
        {
                flag_node_check=0;
                if (output_arr[i] == myid)
                {
                        for (j=0;input_arr[i][j]!=0;j++)
                        {
                                if (output_arr[input_arr[i][j]-1] != myid)
                                        flag_node_check=1;
                        }

                        if (flag_node_check==1) /* we're dealing with a peripheral node */
                        {
                                if (*peripheral_head == NULL)
                                {
                                        peripheral_temp = malloc(sizeof(struct own_node));
                                        peripheral_temp -> globalID = i+1;
                                        peripheral_temp -> internal_or_peripheral = 'p';
                                        peripheral_temp -> owning_proc = myid;

                                        temp_data_node = *global_data_head;
```

```
                    for (;temp_data_node!=NULL;temp_data_node = temp_data_node ->
next_node)
                    {
                        if (temp_data_node -> globalID == i+1)
                        {
                            peripheral_temp -> data_location =
temp_data_node;

                            /* Calculate Hash function...*/
                            hash_map = (int)(pow(3,i+1)) %
HASH_TABLE_LENGTH;

                            if (hash_table[hash_map]==NULL)
                            {
                                hash_temp = malloc(sizeof(struct
hash_node));
                                hash_temp->globalID = i+1;
                                hash_temp->data_location =
temp_data_node;
                                hash_temp->next_node = NULL;
                                hash_table[hash_map] = hash_temp;
                            }
                            else
                            {
                                hash_temp = hash_table[hash_map];

                                while (hash_temp-
>next_node!=NULL)
                                    hash_temp=hash_temp-
>next_node;

                                r_hash = malloc(sizeof(struct
hash_node));
                                r_hash->globalID = i+1;
                                r_hash->data_location =
temp_data_node;
                                r_hash->next_node=NULL;

                                hash_temp->next_node=r_hash;
                            }
                            break;
                        }
                    }

                    for (k=0;k<10;k++)
                        peripheral_temp -> neighboring_nodes[k]=-1;

                    for (j=0,k=0;input_arr[i][j]!=0;j++,k++) /* Populating
neighboring_nodes[] arr...*/
                        peripheral_temp -> neighboring_nodes[k] = input_arr[i][j];

                    for (k=0;k<10;k++)
                        peripheral_temp -> shadow_for_procs[k] = -1;

                    for (k=0;peripheral_temp -> neighboring_nodes[k]!=-1;k++) /*
Populating shadow_for_procs[] arr...*/
                    {
                        insert_shadow_flag=0;
                        if (output_arr[peripheral_temp -> neighboring_nodes[k] -
1] != myid)
                        {
```

```
                                                for (j=0;peripheral_temp ->
shadow_for_procs[j]!=-1;j++)
                                                {
                                                        if (output_arr[peripheral_temp ->
neighboring_nodes[k] -1]==peripheral_temp->shadow_for_procs[j])
                                                                insert_shadow_flag=1;
                                                }

                                                if (insert_shadow_flag==0)
                                                        peripheral_temp ->
shadow_for_procs[j] = output_arr[peripheral_temp -> neighboring_nodes[k] - 1];
                                        }
                                }

                                for (k=0;peripheral_temp->shadow_for_procs[k]!=-1;k++)
                                        buffer_size_for_communication[peripheral_temp-
>shadow_for_procs[k]]++;

                                peripheral_temp -> next_node = NULL;

                                *peripheral_head = peripheral_temp;
                        }
                        else
                        {
                                r_peripheral = malloc(sizeof(struct own_node));
                                r_peripheral -> globalID = i+1;
                                r_peripheral -> internal_or_peripheral = 'p';
                                r_peripheral -> owning_proc = myid;

                                temp_data_node = *global_data_head;
                                for (;temp_data_node!=NULL;temp_data_node = temp_data_node ->
next_node)
                                {
                                        if (temp_data_node -> globalID == i+1)
                                        {
                                                r_peripheral -> data_location = temp_data_node;

                                                /* Calculate Hash function...*/
                                                hash_map = (int)(pow(3,i+1)) %
HASH_TABLE_LENGTH;

                                                if (hash_table[hash_map]==NULL)
                                                {
                                                        hash_temp = malloc(sizeof(struct
hash_node));
                                                        hash_temp->globalID = i+1;
                                                        hash_temp->data_location =
temp_data_node;
                                                        hash_temp->next_node = NULL;
                                                        hash_table[hash_map] = hash_temp;
                                                }
                                                else
                                                {
                                                        hash_temp = hash_table[hash_map];

                                                        while (hash_temp-
>next_node!=NULL)
                                                                hash_temp=hash_temp-
>next_node;

                                                        r_hash = malloc(sizeof(struct
hash_node));
```

```
                                r_hash->globalID = i+1;
                                r_hash->data_location =
temp_data_node;

                                r_hash->next_node=NULL;

                                hash_temp->next_node=r_hash;
                            }
                            break;
                        }
                    }

                    for (k=0;k<10;k++)
                        r_peripheral -> neighboring_nodes[k] = -1;

                    for (j=0,k=0;input_arr[i][j]!=0;j++,k++) /* Populating
neighboring_nodes[] arr...*/
                        r_peripheral -> neighboring_nodes[k] = input_arr[i][j];

                    for (k=0;k<10;k++)
                        r_peripheral -> shadow_for_procs[k] = -1;

                    for (k=0;r_peripheral -> neighboring_nodes[k]!=-1;k++) /*
Populating shadow_for_procs[] arr...*/
                    {
                        insert_shadow_flag=0;
                        if (output_arr[r_peripheral -> neighboring_nodes[k] - 1] !=
myid)
                        {
                            for (j=0;r_peripheral -> shadow_for_procs[j]!=-
1;j++)
                            {
                                if (output_arr[r_peripheral ->
neighboring_nodes[k] -1]==r_peripheral->shadow_for_procs[j])
                                    insert_shadow_flag=1;
                            }

                            if (insert_shadow_flag==0)
                                r_peripheral -> shadow_for_procs[j] =
output_arr[r_peripheral -> neighboring_nodes[k] - 1];
                        }
                    }

                    for (k=0;r_peripheral->shadow_for_procs[k]!=-1;k++)
                        buffer_size_for_communication[r_peripheral-
>shadow_for_procs[k]]++;

                    r_peripheral -> next_node = NULL;

                    peripheral_temp -> next_node = r_peripheral;
                    peripheral_temp = r_peripheral;
                }
            }
            else /* we're dealing with an internal node */
            {
                if (*internal_head == NULL)
                {
                    internal_temp = malloc(sizeof(struct own_node));
                    internal_temp -> globalID = i+1;
                    internal_temp -> internal_or_peripheral = 'i';
                    internal_temp -> owning_proc = myid;

                    temp_data_node = *global_data_head;
```

```
                                        for (;temp_data_node!=NULL;temp_data_node = temp_data_node ->
next_node)
                                        {
                                                if (temp_data_node -> globalID == i+1)
                                                {
                                                        internal_temp      ->      data_location      =
temp_data_node;

                                                        /* Calculate Hash function...*/
                                                        hash_map      =      (int)(pow(3,i+1))      %
HASH_TABLE_LENGTH;

                                                        if (hash_table[hash_map]==NULL)
                                                        {
                                                                hash_temp    =    malloc(sizeof(struct
hash_node));
                                                                hash_temp->globalID = i+1;
                                                                hash_temp->data_location            =
temp_data_node;
                                                                hash_temp->next_node = NULL;
                                                                hash_table[hash_map] = hash_temp;
                                                        }
                                                        else
                                                        {
                                                                hash_temp = hash_table[hash_map];

                                                                while                     (hash_temp-
>next_node!=NULL)
                                                                        hash_temp=hash_temp-
>next_node;

                                                                r_hash      =      malloc(sizeof(struct
hash_node));
                                                                r_hash->globalID = i+1;
                                                                r_hash->data_location               =
temp_data_node;
                                                                r_hash->next_node=NULL;

                                                                hash_temp->next_node=r_hash;
                                                        }
                                                        break;
                                                }
                                        }

                                        for (k=0;k<10;k++)
                                                internal_temp -> neighboring_nodes[k] = -1;

                                        for    (j=0,k=0;input_arr[i][j]!=0;j++,k++)       /*    Populating
neighboring_nodes[] arr...*/
                                                internal_temp -> neighboring_nodes[k] = input_arr[i][j];

                                        /* No shadows for internal nodes! */
                                        for (k=0;k<10;k++)
                                                internal_temp -> shadow_for_procs[k] = -1;

                                        internal_temp -> next_node = NULL;

                                        *internal_head = internal_temp;
                                }
                                else
                                {
                                        r_internal = malloc(sizeof(struct own_node));
r_internal -> globalID = i+1;
r_internal -> internal_or_peripheral = 'i';
r_internal -> owning_proc = myid;

temp_data_node = *global_data_head;
for (;temp_data_node!=NULL;temp_data_node = temp_data_node -> next_node)

{
        if (temp_data_node -> globalID == i+1)
        {
                r_internal -> data_location = temp_data_node;

                /* Calculate Hash function...*/
                hash_map = (int)(pow(3,i+1)) % HASH_TABLE_LENGTH;


                if (hash_table[hash_map]==NULL)
                {
                        hash_temp = malloc(sizeof(struct hash_node));

                        hash_temp->globalID = i+1;
                        hash_temp->data_location = temp_data_node;

                        hash_temp->next_node = NULL;
                        hash_table[hash_map] = hash_temp;
                }
                else
                {
                        hash_temp = hash_table[hash_map];

                        while (hash_temp->next_node!=NULL)

                                hash_temp=hash_temp->next_node;


                        r_hash = malloc(sizeof(struct hash_node));

                        r_hash->globalID = i+1;
                        r_hash->data_location = temp_data_node;

                        r_hash->next_node=NULL;

                        hash_temp->next_node=r_hash;
                }
                break;
        }
}

for (k=0;k<10;k++)
        r_internal -> neighboring_nodes[k] = -1;

for (j=0,k=0;input_arr[i][j]!=0;j++,k++) /* Populating neighboring_nodes[] arr...*/

        r_internal -> neighboring_nodes[k] = input_arr[i][j];

/* No shadows for internal nodes! */
for (k=0;k<10;k++)
        r_internal -> shadow_for_procs[k] = -1;

r_internal -> next_node = NULL;

internal_temp -> next_node = r_internal;
```

```
                                        internal_temp = r_internal;
                                }
                        }
                }
        } /* End of for loop */
}
void InsertShadowsIntoHashTable(struct own_node **peripheral_head, struct node_data **global_data_head, int
*output_arr, struct hash_node *hash_table[], MPI_Comm comm)
{
        int myid,i,hash_map,insert_flag=0;
        struct own_node *peripheral_temp;
        struct node_data *temp_data_node;
        struct hash_node *r_hash,*hash_temp;

        MPI_Comm_rank(comm,&myid);

        temp_data_node = *global_data_head;
        peripheral_temp = *peripheral_head;

        for (;peripheral_temp!=NULL;peripheral_temp=peripheral_temp->next_node)
        {
                for (i=0;peripheral_temp->neighboring_nodes[i]!=-1;i++)
                {
                        if (output_arr[peripheral_temp->neighboring_nodes[i]-1]!=myid) /* viz..we're dealing
with a shadow! */
                        {
                                temp_data_node=*global_data_head;
                                for (;temp_data_node!=NULL;temp_data_node=temp_data_node->next_node)
                                {
                                        if (temp_data_node->globalID==peripheral_temp-
>neighboring_nodes[i])
                                        {
                                                hash_map=(int)(pow(3,peripheral_temp-
>neighboring_nodes[i]))%HASH_TABLE_LENGTH;

                                                if (hash_table[hash_map]==NULL)
                                                {
                                                        hash_temp = malloc(sizeof(struct hash_node));

                                                        hash_temp->globalID=peripheral_temp-
>neighboring_nodes[i];
                                                        hash_temp->data_location=temp_data_node;
                                                        hash_temp->next_node=NULL;
                                                        hash_table[hash_map]=hash_temp;
                                                }
                                                else
                                                {
                                                        hash_temp=hash_table[hash_map];
                                                        insert_flag=0;
                                                        while (hash_temp->next_node!=NULL)
                                                        {

                                                                /* Checkin if the shadow is already
inserted for some other node...*/
                                                                if (hash_temp-
>globalID==peripheral_temp->neighboring_nodes[i])
                                                                {
                                                                        insert_flag=1;
                                                                        break;
                                                                }
                                                                hash_temp=hash_temp->next_node;
                                                        }
```

```
                                                            /* Checkin for the current node...(the last insert)
*/
                                                            if        (hash_temp->globalID==peripheral_temp-
>neighboring_nodes[i])
                                                                    insert_flag=1;

                                                            if (insert_flag==0)
                                                            {
                                                                    r_hash        =        malloc(sizeof(struct
hash_node));
                                                                    r_hash->globalID=peripheral_temp-
>neighboring_nodes[i];
                                                                    r_hash-
>data_location=temp_data_node;
                                                                    r_hash->next_node=NULL;

                                                                    hash_temp->next_node=r_hash;
                                                            }
                                                    }
                                            }
                                    }
                            }
                    }
            }
}
```

Appendix B

This appendix holds the procedures which comprise the platform's Computation & Communication Phase.

```
void ComputeOverNodes(struct own_node **internal_head, struct own_node **peripheral_head, struct node_data
**global_data_head, struct hash_node *hash_table[], struct buffer_data_node *buffer_arr[], int num_procs, int
*buffer_size_for_communication, void (*simulator_ptr)(), MPI_Comm comm, int index, int num_of_vertices)
{
        struct own_node *internal_temp,*peripheral_temp;
        struct node_data *head_for_simulator_function; // Node list consisting of node and neighbors sent to the
simulator code...
        struct node_data *temp_data_node1,*temp_data_node2,*r_for_simulator;
        struct hash_node *temp_hash_node;
        int i,j,hash_index,myid,cnt;

        internal_temp = *internal_head;
        peripheral_temp = *peripheral_head;

        MPI_Comm_rank(comm,&myid);

        // Scan through the internal nodes...
        for (;internal_temp!=NULL;internal_temp=internal_temp->next_node)
        {
                head_for_simulator_function=NULL;

                // Add the current internal node as head of the list to be sent to the simulator function...

                temp_data_node1=malloc(sizeof(struct node_data));
                temp_data_node1->globalID = internal_temp->globalID;
                temp_data_node1->data = internal_temp->data_location->data;

                temp_data_node1->next_node=NULL;
                head_for_simulator_function=temp_data_node1;

                for (i=0;internal_temp->neighboring_nodes[i]!=-1;i++)
                {
                        // Obtain the location of the neighbor...
                        hash_index                    =                    ((int)pow(3,internal_temp-
>neighboring_nodes[i]))%HASH_TABLE_LENGTH;

                        for
(temp_hash_node=hash_table[hash_index];temp_hash_node!=NULL;temp_hash_node=temp_hash_node->next_node)
                        {
                                if  (temp_hash_node->globalID  ==  internal_temp->neighboring_nodes[i])  //
Neighbor obtained...
                                        break;
                        }

                        temp_data_node1=head_for_simulator_function;
                        while (temp_data_node1->next_node!=NULL)
                                temp_data_node1=temp_data_node1->next_node;

                        r_for_simulator=malloc(sizeof(struct node_data));
                        r_for_simulator->globalID = internal_temp->neighboring_nodes[i];
                        r_for_simulator->data = temp_hash_node->data_location->data;
                        r_for_simulator->next_node=NULL;

                        temp_data_node1->next_node=r_for_simulator;
                }


        (*simulator_ptr)(&head_for_simulator_function,&(*global_data_head),myid,num_of_vertices,index);
                free(head_for_simulator_function);
        }

        // Allocate memory for buffer arrays...
        for (i=0;i<num_procs;i++)
        {
                if (buffer_size_for_communication[i]!=0)
                        buffer_arr[i]                    =                    malloc(sizeof(struct
buffer_data_node)*buffer_size_for_communication[i]);
                else
                        buffer_arr[i]=NULL;
        }

        // Initialize buffer arrays...
        for (i=0;i<num_procs;i++)
        {
                if (buffer_size_for_communication[i]!=0)
                {
                        for (j=0;j<buffer_size_for_communication[i];j++)
                        {
                                buffer_arr[i][j].globalID=-1;
                                buffer_arr[i][j].data=-1;
                        }
                }
        }

        // Scan through peripheral nodes...
        for (;peripheral_temp!=NULL;peripheral_temp=peripheral_temp->next_node)
        {
                head_for_simulator_function=NULL;

                // Add the current peripheral node as head of the list to be sent to the simulator function...

                temp_data_node1=malloc(sizeof(struct node_data));

                temp_data_node1->globalID = peripheral_temp->globalID;
                temp_data_node1->data = peripheral_temp->data_location->data;

                temp_data_node1->next_node=NULL;
                head_for_simulator_function=temp_data_node1;

                for (i=0;peripheral_temp->neighboring_nodes[i]!=-1;i++)
                {
                        // Obtain the location of the neighbor...
                        hash_index                  =                  ((int)pow(3,peripheral_temp-
>neighboring_nodes[i]))%HASH_TABLE_LENGTH;

                        for
(temp_hash_node=hash_table[hash_index];temp_hash_node!=NULL;temp_hash_node=temp_hash_node->next_node)
                        {
                                if  (temp_hash_node->globalID  ==  peripheral_temp->neighboring_nodes[i])  //
Neighbor obtained...
                                        break;
                        }

                        temp_data_node1=head_for_simulator_function;
                        while (temp_data_node1->next_node!=NULL)
                                temp_data_node1=temp_data_node1->next_node;
                        r_for_simulator=malloc(sizeof(struct node_data));
                        r_for_simulator->globalID = peripheral_temp->neighboring_nodes[i];
                        r_for_simulator->data = temp_hash_node->data_location->data;
                        r_for_simulator->next_node=NULL;

                        temp_data_node1->next_node=r_for_simulator;
                }


        (*simulator_ptr)(&head_for_simulator_function,&(*global_data_head),myid,num_of_vertices,index);

                // Upto date data for the current node returns at this point...hence add the same,
                // onto the buffer arr for communication...

                for (i=0;peripheral_temp->shadow_for_procs[i]!=-1;i++)
                {
                        // Obtain right position in buffer arr to add the peripheral node...
                        for (j=0;buffer_arr[peripheral_temp->shadow_for_procs[i]][j].globalID!=-1;j++);

                        buffer_arr[peripheral_temp->shadow_for_procs[i]][j].globalID      =      peripheral_temp-
>globalID;
                        buffer_arr[peripheral_temp->shadow_for_procs[i]][j].data         =         peripheral_temp-
>data_location->most_recent_data;
                }

                free(head_for_simulator_function);
        }

        // Update data to most recent data before the next iteration of computation...
        internal_temp = *internal_head;
        peripheral_temp = *peripheral_head;

        for (;internal_temp!=NULL;internal_temp=internal_temp->next_node)
        {
                internal_temp->data_location->data = internal_temp->data_location->most_recent_data;
        }

        for (;peripheral_temp!=NULL;peripheral_temp=peripheral_temp->next_node)
        {
                peripheral_temp->data_location->data = peripheral_temp->data_location->most_recent_data;
        }
}

void  SimulatorFunction(struct  node_data  **head_for_simulator_function,  struct  node_data  **global_data_head,  int
myid, int num_of_vertices, int index)
{
        struct node_data *temp_for_simulator, *temp_data_node;
        int current_globalID, i, j;

        temp_for_simulator = *head_for_simulator_function;
        temp_data_node = *global_data_head;

        current_globalID = temp_for_simulator->globalID;

        for (;temp_data_node!=NULL;temp_data_node=temp_data_node->next_node)
        {
                if (temp_data_node->globalID == current_globalID)
                        break;
        }
```

```
		temp_for_simulator = temp_for_simulator->next_node; /* Move to the next node, coz., 1st node is the
current node to be simulated...*/

		/* Dummy calculation...*/
		temp_data_node->most_recent_data = temp_data_node->data;
		for (i=0;temp_for_simulator!=NULL;temp_for_simulator=temp_for_simulator->next_node,i++)
		{
//			if (i%2 == 0)
//			{
				temp_data_node->most_recent_data += temp_for_simulator->data;
//			}
//			else
//			{
//				temp_data_node->most_recent_data -= temp_for_simulator->data;
//			}
		}

		// Dummy loop to get the measure of the load...
//		for (j=0;j<100000;j++);

		// Creating dynamic imbalance across the computational domain by varying the grain size...
		if (index <= 10)
		{
			if (current_globalID <= (int)((50*num_of_vertices)/100))
			{
				for (j=0;j<100000;j++);
			}
			else
			{
				for (j=0;j<1000;j++);
			}
		}
		else if (index <=20)
		{
			if ((int)((25*num_of_vertices)/100) <= current_globalID <= (int)((75*num_of_vertices)/100))
			{
				for (j=0;j<100000;j++);
			}
			else
			{
				for (j=0;j<1000;j++);
			}
		}
		else if (index <=30)
		{
			if ((int)((50*num_of_vertices)/100) <= current_globalID <= (int)((100*num_of_vertices)/100))
			{
				for (j=0;j<100000;j++);
			}
			else
			{
				for (j=0;j<1000;j++);
			}
		}
}

void CommunicateShadows(struct buffer_data_node *buffer_arr[], struct buffer_data_node *recvbuffer_arr[], int
*buffer_size_for_communication, int num_procs, struct node_data **global_data_head, MPI_Comm comm)
{
		int myid,i,j;
		int blockcounts[1];
		struct node_data *temp_data_node=NULL;
```

```
	MPI_Datatype buffer_datatype, oldtypes[1];
	MPI_Aint offsets[1];
	MPI_Status status;
	MPI_Request pending;

	/* Set up description of 2 MPI_INT fields: globalID and data of the struct type buffer_data_node...*/
	offsets[0]=0;
	oldtypes[0]=MPI_INT;
	blockcounts[0]=2;

	/* Define structured type & commit it...*/
	MPI_Type_struct(1,blockcounts,offsets,oldtypes,&buffer_datatype);
	MPI_Type_commit(&buffer_datatype);

	MPI_Comm_rank(comm,&myid);

	/* Allocate memory for recvbuffers...*/
	for (i=0;i<num_procs;i++)
	{
		if (buffer_size_for_communication[i]!=0)
			recvbuffer_arr[i]		=		malloc(sizeof(struct
buffer_data_node)*MAX_SIZE_FOR_RECVBUFFER);
		else
			recvbuffer_arr[i]=NULL;
	}

	/* Initialize recvbuffers...*/
	for (i=0;i<num_procs;i++)
	{
		if (buffer_size_for_communication[i]!=0)
		{
			for (j=0;j<MAX_SIZE_FOR_RECVBUFFER;j++)
			{
				recvbuffer_arr[i][j].globalID = -1;
				recvbuffer_arr[i][j].data = -1;
			}
		}
	}

	// Send buffers...
	for (i=0;i<num_procs;i++)
	{
		if (buffer_size_for_communication[i]!=0)
		{

	MPI_Isend(buffer_arr[i],buffer_size_for_communication[i],buffer_datatype,i,1,comm,&pending);
		}
	}

	// Recv buffers...
	for (i=0;i<num_procs;i++)
	{
		if (buffer_size_for_communication[i]!=0)
		{

	MPI_Recv(recvbuffer_arr[i],MAX_SIZE_FOR_RECVBUFFER,buffer_datatype,i,1,comm,&status);
		}
	}

	MPI_Barrier(comm);

	// Unpacking the recv buffers...
```

```
        for (i=0;i<num_procs;i++)
        {
                if (buffer_size_for_communication[i]!=0)
                {
                        for (j=0;recvbuffer_arr[i][j].globalID!=-1;j++)
                        {
                                temp_data_node = *global_data_head;
                                for (;temp_data_node!=NULL;temp_data_node=temp_data_node->next_node)
                                {
                                        if (temp_data_node->globalID == recvbuffer_arr[i][j].globalID)
                                        {
                                                temp_data_node->data = recvbuffer_arr[i][j].data;
                                                break;
                                        }
                                }
                        }
                }
        }
}
```

Appendix C

This appendix holds the ‘main’ function, and the load balancing and task migration phase of the platform.

```
int main(int argc, char *argv[])
{
	int num_of_vertices, num_of_edges, fmt, *output_arr, **input_arr, i, j, k, myid, num_procs,
*buffer_size_for_communication, *vertex_weight;
	struct node_data *global_data_head = NULL, *temp;
	struct own_node *internal_head = NULL, *peripheral_head = NULL, *internal_temp, *peripheral_temp;
	struct hash_node *hash_table[HASH_TABLE_LENGTH],*hash_temp;
	struct buffer_data_node **buffer_arr, **recvbuffer_arr;
	int
load_imbalance=1,migrating_node,to_proc,from_proc,**task_migration_pairs,*migrating_node_arr,**proc_holding_s
hadows,f_proc,t_proc;
	int **proc_node_graph_edges, **temp_buff, *isFiltered_arr, *isMigrated_arr, isFiltered, *to_proc_reserved,
migration_needed, bcast_flag;
	double time_elapsed, proc_execution_time=0, proc_execution_time_for_one_iteration=0,
*proc_execute_timings;
	MPI_Status status;

	MPI_Init(&argc,&argv);
	MPI_Comm_rank(MPI_COMM_WORLD,&myid);
	MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

	MPI_Barrier(MPI_COMM_WORLD);
	time_elapsed = -MPI_Wtime();

	/* Initialization Phase...*/
	InitializeGraph(&num_of_vertices, &num_of_edges, &fmt);

	input_arr = (int *)malloc(sizeof(int)*num_of_vertices);

	for (i=0;i<num_of_vertices;i++)
	{
		input_arr[i] = (int *)malloc(sizeof(int)*COLUMN_WIDTH_OF_INPUT_ARRAY);
	}

	vertex_weight = (int *)malloc(sizeof(int)*num_of_vertices);
	InitializeInputArray(input_arr,num_of_vertices,fmt,vertex_weight);

	output_arr = (int *)malloc(sizeof(int)*num_of_vertices);
	InitializeOutputArray(output_arr,num_of_vertices);

	InitializeGlobalDataList(&global_data_head,num_of_vertices);

	for (i=0;i<HASH_TABLE_LENGTH;i++)
		hash_table[i]=NULL;

	buffer_size_for_communication = (int *)malloc(sizeof(int)*num_procs);

	for (i=0;i<num_procs;i++)
		buffer_size_for_communication[i]=0;


	InitializeNodeLists(&internal_head,&peripheral_head,&global_data_head,hash_table,input_arr,output_arr,nu
m_of_vertices,fmt,buffer_size_for_communication,MPI_COMM_WORLD);
```

```
        InsertShadowsIntoHashTable(&peripheral_head,&global_data_head,output_arr,hash_table,MPI_COMM_W
ORLD);

        buffer_arr = malloc(sizeof(struct buffer_data_node)*num_procs); // Array of array of structures as buffer at
each proc for communication...
        recvbuffer_arr = malloc(sizeof(struct buffer_data_node)*num_procs);

        if (myid == 0) // Initialization of data structures for load-rebalancing strategies...
        {
                proc_execute_timings = (double *)malloc(sizeof(double)*num_procs);

                proc_node_graph_edges = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        proc_node_graph_edges[j] = (int *)malloc(sizeof(int)*num_procs);
                }

                temp_buff = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        temp_buff[j] = (int *)malloc(sizeof(int)*num_procs);
                }

                for (i=0;i<num_procs;i++)
                {
                        for (j=0;j<num_procs;j++)
                        {
                                proc_node_graph_edges[i][j]=0;
                                temp_buff[i][j]=0;
                        }
                }

                // Data structure which would hold the possible task migration pairs...
                task_migration_pairs = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        task_migration_pairs[j] = (int *)malloc(sizeof(int)*2);
                }

                for (i=0;i<num_procs;i++)  //  Note tht for the worst case scenario: for an application with n procs,
n-1 is the max number of task_migration pairs...
                {
                        for (j=0;j<2;j++)
                        {
                                task_migration_pairs[i][j]=-1;
                        }
                }

                migrating_node_arr = (int *)malloc(sizeof(int)*num_procs); //arr which would hold the migrating
nodes for the from_proc/to_proc pairs...
                for (j=0;j<num_procs;j++)
                {
                        migrating_node_arr[j] = -1;
                }

                // DS which would be required to filter out from_proc/to_proc pairs...
                proc_holding_shadows = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        proc_holding_shadows[j] = (int *)malloc(sizeof(int)*10);
                }
                for (i=0;i<num_procs;i++)
                {
                        for (j=0;j<10;j++)
                        {
                                proc_holding_shadows[i][j]=-1;
                        }
                }

                // array which would specify whether the kth pair is filtered or not...
                isFiltered_arr = (int *)malloc(sizeof(int)*num_procs);
                for (i=0;i<num_procs;i++)
                        isFiltered_arr[i] = -1;

                isMigrated_arr = (int *)malloc(sizeof(int)*num_procs);
                for (i=0;i<num_procs;i++)
                        isMigrated_arr[i] = -1;

                to_proc_reserved = (int *)malloc(sizeof(int)*num_procs);
                for (i=0;i<num_procs;i++)
                        to_proc_reserved[i] =-1;
        }
        else
        {
                // Note tht for myid !=0, only the task migration pairs are needed to participate in task migration...
                // Data structure which would hold the possible task migration pairs...
                task_migration_pairs = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        task_migration_pairs[j] = (int *)malloc(sizeof(int)*2);
                }

                for (i=0;i<num_procs;i++)  // Note tht for the worst case scenario: for an application with n procs,
n-1 is the max number of task_migration pairs...
                {
                        for (j=0;j<2;j++)
                        {
                                task_migration_pairs[i][j]=-1;
                        }
                }

                migrating_node_arr = (int *)malloc(sizeof(int)*num_procs); // arr which would hold the migrating
nodes for the from_proc/to_proc pairs...
                for (j=0;j<num_procs;j++)
                {
                        migrating_node_arr[j] = -1;
                }

                // DS which would be required to filter out from_proc/to_proc pairs...
                proc_holding_shadows = (int **)malloc(sizeof(int)*num_procs);
                for (j=0;j<num_procs;j++)
                {
                        proc_holding_shadows[j] = (int *)malloc(sizeof(int)*10);
                }

                for (i=0;i<num_procs;i++)
                {
                        for (j=0;j<10;j++)
                        {
                                proc_holding_shadows[i][j]=-1;
                        }
                }
```

```
		// array which would specify whether the kth pair is filtered or not...
		isFiltered_arr = (int *)malloc(sizeof(int)*num_procs);
		for (i=0;i<num_procs;i++)
			isFiltered_arr[i] = -1;

		isMigrated_arr = (int *)malloc(sizeof(int)*num_procs);
		for (i=0;i<num_procs;i++)
			isMigrated_arr[i] = -1;

		to_proc_reserved = (int *)malloc(sizeof(int)*num_procs);
		for (i=0;i<num_procs;i++)
			to_proc_reserved[i] =-1;
	}

	for (i=1;i<=20;i++)
	{
		proc_execution_time_for_one_iteration = -MPI_Wtime();

		// Computation Phase...

	ComputeOverNodes(&internal_head,&peripheral_head,&global_data_head,hash_table,buffer_arr,num_procs
,buffer_size_for_communication,SimulatorFunction,MPI_COMM_WORLD,i,num_of_vertices);

		proc_execution_time_for_one_iteration += MPI_Wtime();
		proc_execution_time += proc_execution_time_for_one_iteration;

		// Communication Phase...

	CommunicateShadows(buffer_arr,recvbuffer_arr,buffer_size_for_communication,num_procs,&global_data_
head,MPI_COMM_WORLD);

		if (i % 10 == 0) // Periodically invoking load-rebalancing strategies...
		{
			// Building the proc-node graph...(nodes)
			if (myid != 0)
			{

	MPI_Send(&proc_execution_time,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD);
			}
			else
			{
				proc_execute_timings[0] = proc_execution_time;
				for (j=1;j<num_procs;j++)
				{
					proc_execute_timings[j]=0;
				}

				for (j=1;j<num_procs;j++)
				{

	MPI_Recv(&proc_execute_timings[j],1,MPI_DOUBLE,j,j,MPI_COMM_WORLD,&status);
				}
			}
			MPI_Barrier(MPI_COMM_WORLD);

			// Building the proc-node graph...(edges)
			if (myid != 0)
			{

	MPI_Send(buffer_size_for_communication,num_procs,MPI_INT,0,myid,MPI_COMM_WORLD);
			}
			else
```

```
                {
                        for (j=0;j<num_procs;j++)
                                temp_buff[0][j] = buffer_size_for_communication[j];

                        for (j=1;j<num_procs;j++)
                        {
        MPI_Recv(temp_buff[j],num_procs,MPI_INT,j,j,MPI_COMM_WORLD,&status);
                        }

                        for (j=0;j<num_procs;j++)
                        {
                                for (k=0;k<num_procs;k++)
                                {
                                        if (j != k)
                                        {
                                                proc_node_graph_edges[j][k]  =  temp_buff[j][k]
+ temp_buff[k][j];
                                        }
                                }
                        }
                }

                // Get Rebalancing parameters if susbtantial load-imbalance as per the load-rebalance
algorithm...
                if (myid == 0)
                {
                        load_imbalance                                                              =
GetLoadRebalancingParameters(proc_execute_timings,proc_node_graph_edges,task_migration_pairs,num_procs,MPI
_COMM_WORLD);
                }
                MPI_Barrier(MPI_COMM_WORLD);

                // At this point proc 0 knows whtz the load_imbalance parameter
                MPI_Bcast(&load_imbalance,1,MPI_INT,0,MPI_COMM_WORLD);

                if (load_imbalance == 1)
                {
                        // At this point all the task migration pairs are ready but with myid=0, which
wud broadcast to rest of the procs...
                //
        MPI_Bcast(&task_migration_pairs[0][0],num_procs*2,MPI_INT,0,MPI_COMM_WORLD);

                        for (k=0;k<num_procs;k++)
                        {
                                if (myid == 0)
                                {
                                        f_proc = task_migration_pairs[k][0];
                                }
                                MPI_Bcast(&f_proc,1,MPI_INT,0,MPI_COMM_WORLD);
                                if (myid != 0)
                                {
                                        task_migration_pairs[k][0] = f_proc;
                                }

                                if (myid == 0)
                                {
                                        t_proc = task_migration_pairs[k][1];
                                }
                                MPI_Bcast(&t_proc,1,MPI_INT,0,MPI_COMM_WORLD);
                                if (myid != 0)
                                {
                                        task_migration_pairs[k][1] = t_proc;
                                }
                        }

                        for (k=0;task_migration_pairs[k][0]!=-1;k++)
                        {
                                from_proc = task_migration_pairs[k][0];
                                to_proc = task_migration_pairs[k][1];

                                // from_proc would get the migrating_node tht 'mite' be sent to
to_proc...
                                if (myid == from_proc)
                                {
                                        migrating_node                                          =
GetMigratingNode(&peripheral_head,to_proc,from_proc,input_arr,output_arr);
                                        migrating_node_arr[k] = migrating_node;
                                }
                                MPI_Barrier(MPI_COMM_WORLD);

                                // At this point migrating node is decided and with from_proc which
wud broadcast to the rest of the procs...

        MPI_Bcast(&migrating_node,1,MPI_INT,from_proc,MPI_COMM_WORLD);
                                if (myid != from_proc)
                                {
                                        migrating_node_arr[k] = migrating_node;
                                }

                                // At this point all the load-rebalancing parameters are decided for the
kth pair, check is they get filtered for task migration...
                                if (k==0)
                                {
                                        if (myid == from_proc)
                                        {

        UpdateMigratingNodeInfo(proc_holding_shadows,k,&peripheral_head,migrating_node_arr,task_migration_p
airs);
                                        }

        MPI_Bcast(&proc_holding_shadows[k][0],10,MPI_INT,from_proc,MPI_COMM_WORLD);

                                        isFiltered_arr[k] = 1; // first task migration filtered...(no
condition check required, which is obvious)...
                        //
        task_migrate(migrating_node,from_proc,to_proc,output_arr,&internal_head,&peripheral_head,hash_table,&
global_data_head,input_arr,num_procs,buffer_size_for_communication,MPI_COMM_WORLD);
                                }
                                else
                                {
                                        // logic for filtering the pairs...
                                        if (myid == from_proc)
                                        {
                                                isFiltered                                      =
UpdateMigratingNodeInfo(proc_holding_shadows,k,&peripheral_head,migrating_node_arr,task_migration_pairs);
                                                isFiltered_arr[k] = isFiltered;
                                        }

        MPI_Bcast(&proc_holding_shadows[k][0],10,MPI_INT,from_proc,MPI_COMM_WORLD);


        MPI_Bcast(&isFiltered,1,MPI_INT,from_proc,MPI_COMM_WORLD);
```

```
                        if (myid != from_proc)
                        {
                            isFiltered_arr[k] = isFiltered;
                        }
                    }
                }

                // Update the output_arr of all the procs coz. of the change of the owning_proc
of the migrating node...
                for (k=0;k<num_procs;k++)
                {
                    if (isFiltered_arr[k] == 1)
                    {
                        output_arr[migrating_node_arr[k]        -        1]        =
task_migration_pairs[k][1];
                    }
                }

                // Update isMigrated_arr as per isFiltered_arr...
                for (k=0;k<num_procs;k++)
                {
                    if (isFiltered_arr[k] == -1)
                        isMigrated_arr[k] = 1; // coz., there's no such kth pair to be
migrated...
                }

                while (1) // Note, tht now, even if a proc is receiving 2 tasks from different
neighboring procs, the migrations wud go ahead, but sequentially...(obvious)...
                {
                    from_proc = -1;
                    to_proc = -1;
                    migrating_node = -1;
                    migration_needed = 0;

                    for (k=0;k<num_procs;k++)
                        to_proc_reserved[k] = -1;

                    for (k=0;k<num_procs;k++)
                    {
                        bcast_flag = -1;
                        if (isMigrated_arr[k] == -1)
                        {
                            migration_needed = 1;

                            if (myid == task_migration_pairs[k][0])
                            {
                                // check if to_proc reserved for this
migration, if yes, this migration not possible during the current iteration...
                                if
(to_proc_reserved[task_migration_pairs[k][1]] == 1)
                                {
                                    bcast_flag = -1;
                                }
                                else
                                {

    to_proc_reserved[task_migration_pairs[k][1]] = 1;
                                    isMigrated_arr[k] = 1;
                                    bcast_flag = 1;

                                    from_proc = myid;
                                    to_proc                  =
task_migration_pairs[k][1];
                                    migrating_node           =
migrating_node_arr[k];
                                }
                            }
                            else if (myid == task_migration_pairs[k][1])
                            {
                                if
(to_proc_reserved[task_migration_pairs[k][1]] == 1)
                                {}
                                else
                                {
                                    to_proc = myid;
                                    from_proc                =
task_migration_pairs[k][0];
                                    migrating_node           =
migrating_node_arr[k];
                                }
                            }


    MPI_Bcast(&bcast_flag,1,MPI_INT,task_migration_pairs[k][0],MPI_COMM_WORLD);

                            if (bcast_flag == 1)
                            {

    to_proc_reserved[task_migration_pairs[k][1]] = bcast_flag;
                                isMigrated_arr[k] = bcast_flag;
                            }
                        }
                    }

                    MPI_Barrier(MPI_COMM_WORLD);

                    if (migration_needed == 0)
                        break;

                    // task migrate in parallel...

    task_migrate(migrating_node,from_proc,to_proc,output_arr,&internal_head,&peripheral_head,hash_table,&
global_data_head,input_arr,num_procs,buffer_size_for_communication,MPI_COMM_WORLD);
                }
            }

            // Re - initialize the data structures for load-rebalancing strategies...
            if (myid == 0)
            {
                for (k=0;k<num_procs;k++)
                {
                    for (j=0;j<num_procs;j++)
                    {
                        proc_node_graph_edges[k][j]=0;
                        temp_buff[k][j]=0;
                    }
                }

                for (k=0;k<num_procs;k++)
                {
                    for (j=0;j<2;j++)
                    {
                        task_migration_pairs[k][j]=-1;
                    }
                }

                for (j=0;j<num_procs;j++)
                {
                    migrating_node_arr[j] = -1;
                }

                for (k=0;k<num_procs;k++)
                {
                    for (j=0;j<10;j++)
                    {
                        proc_holding_shadows[k][j]=-1;
                    }
                }

                for (k=0;k<num_procs;k++)
                    isFiltered_arr[k] = -1;

                for (k=0;k<num_procs;k++)
                    isMigrated_arr[k] = -1;

                for (k=0;k<num_procs;k++)
                    to_proc_reserved[k] = -1;
            }
            else
            {
                for (k=0;k<num_procs;k++)
                {
                    for (j=0;j<2;j++)
                    {
                        task_migration_pairs[k][j]=-1;
                    }
                }

                for (j=0;j<num_procs;j++)
                {
                    migrating_node_arr[j] = -1;
                }

                for (k=0;k<num_procs;k++)
                {
                    for (j=0;j<10;j++)
                    {
                        proc_holding_shadows[k][j]=-1;
                    }
                }

                for (k=0;k<num_procs;k++)
                    isFiltered_arr[k] = -1;

                for (k=0;k<num_procs;k++)
                    isMigrated_arr[k] = -1;

                for (k=0;k<num_procs;k++)
                    to_proc_reserved[k] = -1;
            }

            proc_execution_time=0;
            proc_execution_time_for_one_iteration=0;
        }
```

```
	//		MPI_Barrier(MPI_COMM_WORLD);
	}

	MPI_Barrier(MPI_COMM_WORLD);
	time_elapsed += MPI_Wtime();

	if (myid == 0)
	{
		printf("\nTime Elapsed = %f\n",time_elapsed);

	/*	for (j=0;j<num_procs;j++)
		{
			printf(":*:%f:*:",proc_execute_timings[j]);
		}*/

		for (i=0;i<num_procs;i++)
		{
			printf("\n");
			for (j=0;j<num_procs;j++)
				printf("*%d*",proc_node_graph_edges[i][j]);
		}
	}

	// Displaying the initialized buffers for communication at the local memories of each proc...

/*  printf("\nfor proc %d...",myid);
	for (i=0;i<num_procs;i++)
	{
		printf("\n");
		if (buffer_size_for_communication[i]!=0)
		{
			printf("%d:",i);
			for (j=0;j<buffer_size_for_communication[i];j++)
			{
				printf("%d, %d ",buffer_arr[i][j].globalID,buffer_arr[i][j].data);
			}
		}
	}*/

	// Displaying the recv buffers after communication...

/*	for (i=0;i<num_procs;i++)
	{
		printf("\n");
		if (buffer_size_for_communication[i]!=0)
		{
			printf("%d:",i);
			for (j=0;j<MAX_SIZE_FOR_RECVBUFFER;j++)
			{
				printf("%d, %d ",recvbuffer_arr[i][j].globalID,recvbuffer_arr[i][j].data);
			}
		}
	}*/

	// Displaying Hash Table for internal and peripheral nodes & shadows!...
	for (i=0;i<HASH_TABLE_LENGTH;i++)
	{
		for (hash_temp=hash_table[i];hash_temp!=NULL;hash_temp=hash_temp->next_node)
			printf("\nProc   %d:   %d   ->   %d   ->   %d   ",myid,i,hash_temp->globalID,hash_temp-
>data_location->globalID);
	}
```

```
    // Displaying number of vertices, edges to be dealt with + input array (graph connectivity) +
        // output array (node to proc assignment)...

/*      printf("%d %d %d\n",num_of_vertices,num_of_edges,fmt);

        for (i=0;i<num_of_vertices;i++)
        {
                printf("%d ",output_arr[i]);
        }

        printf("\n");

        for (i=0;i<num_of_vertices;i++)
        {
                for (j=0;j<COLUMN_WIDTH_OF_INPUT_ARRAY;j++)
                {
                        printf("%d ",input_arr[i][j]);
                }
                printf("\n");
        }*/

        // Displaying data lists...

        temp=global_data_head;
        printf("\n");
        for (;temp!=NULL;temp=temp->next_node)
        {
                printf("%d->",temp->data);
        }

        // Displaying internal lists...

        printf("\nfor proc %d...",myid);
        printf("\ninternal lists...");

        internal_temp = internal_head;
        for (;internal_temp!=NULL;internal_temp=internal_temp->next_node)
        {
                printf("%d data location ->",internal_temp->globalID);
                printf("%d ",internal_temp->data_location->globalID);
        }

        // Displaying peripheral lists...

        printf("\n\nperipheral lists...");

        peripheral_temp = peripheral_head;
        for (;peripheral_temp!=NULL;peripheral_temp=peripheral_temp->next_node)
        {
                printf("%d(%d) data location ->",peripheral_temp->globalID,peripheral_temp->owning_proc);
                printf("%d ",peripheral_temp->data_location->globalID);
        }

        free(input_arr);
        free(output_arr);
        free(global_data_head);
        free(internal_head);
        free(peripheral_head);
        free(hash_table);
        free(buffer_arr);
        free(recvbuffer_arr);
        free(vertex_weight);
```

```
		free(task_migration_pairs);
		free(migrating_node_arr);
		free(proc_holding_shadows);
		free(isFiltered_arr);
		free(isMigrated_arr);
		free(to_proc_reserved);

		MPI_Finalize();
		return 0;
}

int UpdateMigratingNodeInfo(int **proc_holding_shadows, int k, struct own_node **peripheral_head, int
*migrating_node_arr, int **task_migration_pairs)
{
		struct own_node *peripheral_temp;
		int j,i,l;

		peripheral_temp = *peripheral_head;

		for (;peripheral_temp!=NULL;peripheral_temp=peripheral_temp->next_node)
		{
				if (peripheral_temp->globalID == migrating_node_arr[k])
						break;
		}

		// Updating the proc_holding_shadows[][] DS...
		for (j=0;peripheral_temp->shadow_for_procs[j]!=-1;j++)
		{
				proc_holding_shadows[k][j] = peripheral_temp->shadow_for_procs[j];
		}

		// All filtering conditions removed..., all the task migrations wud be xecuted!
		return 1;

/*		if (k==0)
		{
				return 1; // first migration wud take place no matter wht...
		}
		else
		{
				for (j=0;j<k;j++)
				{
						// if 'to_proc' for any two pairs clash, task migration cannot be xecuted in parallel...
						if (task_migration_pairs[j][1] == task_migration_pairs[k][1])
								return 0;
				}
				return 1;
		} */

/*		if (k==0)
		{
				return 1; // first migration wud take place no matter wht...
		}
		else
		{
				// logic for filtering...
				for (i=0;proc_holding_shadows[k][i]!=-1;i++)
				{
						for (j=0;j<k;j++)
						{
								for (l=0;proc_holding_shadows[j][l]!=-1;l++)
								{
```

```
                                                        // Check if the intersection set of the procs holding shadows for the
migrating node for any
                                                        // two pairs clashes...if they do, then the current task cannot be
migrated!
                                                        if (proc_holding_shadows[j][l] == proc_holding_shadows[k][i])
                                                                return 0;
                                                }
                                        }
                                }
                                return 1;
                        }*/
}

int GetMigratingNode(struct own_node **peripheral_head, int to_proc, int from_proc, int **input_arr, int *output_arr)
{
        struct own_node *temp;
        int shadow_for_to_proc[10],i,k=0,num_shadow_for_to_proc=0,*node_edge_cut,min_index=0;

        temp = *peripheral_head;

        for (i=0;i<10;i++)
        {
                shadow_for_to_proc[i] = -1;
        }

        for (;temp!=NULL;temp=temp->next_node)
        {
                for (i=0;temp->shadow_for_procs[i]!=-1;i++)
                {
                        if (temp->shadow_for_procs[i] == to_proc)
                        {
                                shadow_for_to_proc[k++] = temp->globalID;
                                break;
                        }
                }
        }

        for (k=0;shadow_for_to_proc[k]!=-1;k++)
        {
//              printf("\n%d#",shadow_for_to_proc[k]);
                num_shadow_for_to_proc ++;
        }

        node_edge_cut = (int *)malloc(sizeof(int)*num_shadow_for_to_proc);

        for (k=0;k<num_shadow_for_to_proc;k++)
        {
                node_edge_cut[k]=0;
        }

        for (i=0;shadow_for_to_proc[i]!=-1;i++)
        {
                for (k=0;input_arr[shadow_for_to_proc[i] - 1][k]!=0;k++)
                {
                        if (output_arr[input_arr[shadow_for_to_proc[i] - 1][k] - 1] == from_proc)
                        {
                                node_edge_cut[i]++;
                        }
                        else if (output_arr[input_arr[shadow_for_to_proc[i] - 1][k] - 1] == to_proc)
                        {
                                node_edge_cut[i]--;
                        }
                }
        }

/*      for (i=0;i<num_shadow_for_to_proc;i++)
        {
                printf("##%d##",node_edge_cut[i]);
        } */

        for (i=1;i<num_shadow_for_to_proc;i++)
        {
                if (node_edge_cut[i] < node_edge_cut[min_index])
                {
                        min_index=i;
                }
        }

        return shadow_for_to_proc[min_index];
}

int GetLoadRebalancingParameters(double *proc_execute_timings, int **proc_node_graph_edges, int
**task_migration_pairs, int num_procs, MPI_Comm comm)
{
        int myid,i,j,k,imbalance_flag=0,proc_imbalance_flag;
        int **relative_proc_load;

        MPI_Comm_rank(comm,&myid);

/*      for (j=0;j<num_procs;j++)
        {
                printf(":*:%f:*:",proc_execute_timings[j]);
        }

        for (i=0;i<num_procs;i++)
        {
                printf("\n");
                for (j=0;j<num_procs;j++)
                        printf("*%d*",proc_node_graph_edges[i][j]);
        } */

        relative_proc_load = (int **)malloc(sizeof(int)*num_procs);
        for (i=0;i<num_procs;i++)
        {
                relative_proc_load[i] = (int *)malloc(sizeof(int)*num_procs);
        }

        for (i=0;i<num_procs;i++)
        {
                for (j=0;j<num_procs;j++)
                {
                        relative_proc_load[i][j] = 0;
                }
        }

        // Set up the relative processor work load to check for imbalance...
        for (i=0;i<num_procs;i++)
        {
                for (j=0;j<num_procs;j++)
                {
                        if ((i != j) && (proc_node_graph_edges[i][j] > 0)) // check if edge exists between procs i
and j...
                        {
                                if (proc_execute_timings[i] > proc_execute_timings[j])
```

```
                                        {
                                                relative_proc_load[i][j]      =      (int)(((proc_execute_timings[i]      -
proc_execute_timings[j])/proc_execute_timings[j])*100);
                                        }
                                }
                        }
                }

                for (i=0;i<num_procs;i++)
                {
                        printf("\n");
                        for (j=0;j<num_procs;j++)
                                printf("(*%d*)",relative_proc_load[i][j]);
                }

                k=0;
                for (i=0;i<num_procs;i++)
                {
                        proc_imbalance_flag=1;
                        for (j=0;j<num_procs;j++)
                        {
                                if ((i != j) && (proc_node_graph_edges[i][j] > 0)) // check if edge exists between procs i
and j...
                                {
                                        // Check if the imbalance among neighbors is within feasible threshold (25%)
here...
                                        if (relative_proc_load[i][j] < 25)
                                        {
                                                proc_imbalance_flag=0;
                                                break;
                                        }
                                }
                        }

                        // At this point we obtain from_proc/to_proc pair, if proc_imabalance_flag =1...
                        // from_proc wud be i, while to_proc wud be the proc w.r.t. which i does max work...(viz. to_proc
is the idle most w.r.t. from_proc)
                        if (proc_imbalance_flag == 1)
                        {
                                task_migration_pairs[k][0] = i;

                                task_migration_pairs[k][1] = 0;
                                for (j=1;j<num_procs;j++)
                                {
                                        if (relative_proc_load[i][j] > relative_proc_load[i][task_migration_pairs[k][1]])
                                        {
                                                task_migration_pairs[k][1] = j;
                                        }
                                }

                                k++;
                                imbalance_flag=1;
                        }
                }

                for (i=0;i<num_procs;i++)
                {
                        for (j=0;j<2;j++)
                        {
                                printf("*%d*",task_migration_pairs[i][j]);
                        }
                        printf("\n");
```

```
        }

        free(relative_proc_load);
        return imbalance_flag;
}

void task_migrate(int migrating_node, int from_proc, int to_proc, int *output_arr, struct own_node **internal_head,
struct own_node **peripheral_head, struct hash_node *hash_table[], struct node_data **global_data_head, int
**input_arr, int num_procs, int *buffer_size_for_communication, MPI_Comm comm)
{
        int
myid,i,flag,k,j,insert_shadow_flag,neighbors_of_migrating_node[10],num_neighbors_of_migrating_node=0;
        int blockcounts[1],hash_map,hash_flag=0,data_flag=0,p_flag=0;
        struct own_node *temp,*prev,*temp1=NULL,*save_node=NULL,*temp_p,*r,*temp_i;
        struct buffer_data_node send_buff[MIGRATING_NEIGHBORS],recv_buff[MIGRATING_NEIGHBORS];
        struct hash_node *hash_temp,*r_hash;
        struct node_data *temp_data, *p;
        MPI_Datatype buffer_datatype, oldtypes[1];
        MPI_Aint offsets[1];
        MPI_Status status;

        MPI_Comm_rank(comm,&myid);

        /* Set up description of 2 MPI_INT fields: globalID and data of the struct type buffer_data_node...*/
        offsets[0]=0;
        oldtypes[0]=MPI_INT;
        blockcounts[0]=2;

        /* Define structured type & commit it...*/
        MPI_Type_struct(1,blockcounts,offsets,oldtypes,&buffer_datatype);
        MPI_Type_commit(&buffer_datatype);

        for (k=0;k<MIGRATING_NEIGHBORS;k++)
        {
                send_buff[k].globalID = -1;
                send_buff[k].data = -1;
        }

        if (myid == from_proc)
        {
                temp = *internal_head;
                prev = *internal_head;

        //      output_arr[migrating_node - 1] = to_proc;

                for (;temp!=NULL;temp=temp->next_node)
                {
                        i=0;
                        flag=0;

                        for(;temp->neighboring_nodes[i]!=-1;i++)
                        {
                                if (output_arr[temp->neighboring_nodes[i]-1]!=myid)
                                {
                                        flag=1;
                                        break;
                                }
                        }

                        if (flag==1)
                        {
                                save_node = malloc(sizeof(struct own_node));
```

```
if (temp == *internal_head)
{
        temp1 = temp->next_node;
        save_node = temp;
        *internal_head=temp1;
}
else
{
        prev->next_node = temp->next_node;
        save_node = temp;
}

// Append the deleted node to the peripheral node list...
temp_p = *peripheral_head;

if (*peripheral_head != NULL)
{
        while (temp_p->next_node != NULL)
                temp_p=temp_p->next_node;
}

r = malloc(sizeof(struct own_node));

r->globalID = save_node->globalID;
r->owning_proc = myid;
r->data_location = save_node->data_location;

for (k=0;k<10;k++)
        r->neighboring_nodes[k]=save_node->neighboring_nodes[k];

r->internal_or_peripheral = 'p';

for (k=0;k<10;k++)
        r -> shadow_for_procs[k] = -1;

for (k=0;r -> neighboring_nodes[k]!=-1;k++) // Populating shadow_for_procs[]
arr...

{
        insert_shadow_flag=0;
        if (output_arr[r -> neighboring_nodes[k] - 1] != myid)
        {
                for (j=0;r -> shadow_for_procs[j]!=-1;j++)
                {
                        if (output_arr[r -> neighboring_nodes[k] -1]==r-
>shadow_for_procs[j])
                                insert_shadow_flag=1;
                }

                if (insert_shadow_flag==0)
                        r -> shadow_for_procs[j] = output_arr[r ->
neighboring_nodes[k] - 1];
        }
}

r->next_node=NULL;

if (*peripheral_head != NULL)
{
        temp_p->next_node = r;
}
else
```

```
                {
                        *peripheral_head = r;
                }

                free(save_node);
        }

        if (flag == 0)
        {
                prev = temp;
        }
}

// Remove the migrating node from the peripheral node list...
temp = *peripheral_head;
prev = *peripheral_head;

for (;temp!=NULL;temp=temp->next_node)
{
        flag=0;
        if (temp->globalID == migrating_node)
        {
                flag = 1;

                // Get all the neighbors of the migrating node...
                for (k=0;k<10;k++)
                {
                        neighbors_of_migrating_node[k]=-1;
                }
                for (k=0;temp->neighboring_nodes[k]!=-1;k++)
                {
                        neighbors_of_migrating_node[k]=temp->neighboring_nodes[k];
                        num_neighbors_of_migrating_node++;
                }
        }

        if (flag==1)
        {
                if (temp == *peripheral_head)
                {
                        temp1= temp->next_node;
                        free(temp);
                        *peripheral_head = temp1;
                }
                else
                {
                        prev->next_node = temp->next_node;
                        free(temp);
                }

                break;
        }

        prev = temp;
}

// Updating the shadow_for_procs[] array for the peripheral nodes...
for (temp_p=*peripheral_head;temp_p!=NULL;temp_p=temp_p->next_node)
{
        for (j=0;j<10;j++)
                temp_p -> shadow_for_procs[j] = -1;
```

```
            for (k=0;temp_p -> neighboring_nodes[k]!=-1;k++) // Populating shadow_for_procs[]
arr...
            {
                insert_shadow_flag=0;
                if (output_arr[temp_p -> neighboring_nodes[k] - 1] != myid)
                {
                    for (j=0;temp_p -> shadow_for_procs[j]!=-1;j++)
                    {
                        if (output_arr[temp_p -> neighboring_nodes[k] -1] ==
temp_p->shadow_for_procs[j])
                            insert_shadow_flag=1;
                    }

                    if (insert_shadow_flag==0)
                        temp_p -> shadow_for_procs[j] = output_arr[temp_p ->
neighboring_nodes[k] - 1];
                }
            }
        }

        // Send all the neighbors of migrating node...

        for (k=0;k<num_neighbors_of_migrating_node;k++)
        {
            send_buff[k].globalID = neighbors_of_migrating_node[k];

            // Get location of the node from hash table to access the data for the same...
            hash_map=(int)(pow(3,neighbors_of_migrating_node[k]))%HASH_TABLE_LENGTH;

            for (hash_temp=hash_table[hash_map];hash_temp!=NULL;hash_temp=hash_temp-
>next_node)
            {
                if (neighbors_of_migrating_node[k] == hash_temp->globalID)
                {
                    send_buff[k].data = hash_temp->data_location->data;
                    break;
                }
            }
        }

        MPI_Send(send_buff,num_neighbors_of_migrating_node,buffer_datatype,to_proc,1,comm);
    }
    else if (myid == to_proc)
    {
        for (k=0;k<MIGRATING_NEIGHBORS;k++)
        {
            recv_buff[k].globalID = -1;
            recv_buff[k].data = -1;
        }
        MPI_Recv(recv_buff,MIGRATING_NEIGHBORS,buffer_datatype,from_proc,1,comm,&status);

        for (k=0;recv_buff[k].globalID!=-1;k++)
        {
            hash_flag=0;
            hash_map=(int)(pow(3,recv_buff[k].globalID))%HASH_TABLE_LENGTH;

            for (hash_temp=hash_table[hash_map];hash_temp!=NULL;hash_temp=hash_temp-
>next_node)
            {
                if (recv_buff[k].globalID == hash_temp->globalID)
                {
                    // Node found in hash table, update the received node data...
```

```
                        hash_temp->data_location->data = recv_buff[k].data;
                        hash_flag=1;
                        break;
                }
        }

        // If no entry found in the hash table..., then update/add the data node, and add node
information in the hash table
        if (hash_flag==0)
        {
                temp_data = *global_data_head;
                for (;temp_data!=NULL;temp_data=temp_data->next_node)
                {
                        data_flag=0;
                        // Data node found, update it...
                        if (temp_data->globalID == recv_buff[k].globalID)
                        {
                                temp_data->data = recv_buff[k].data;
                                data_flag=1;
                                break;
                        }
                }

                // No data node found, add node information in the data list...
                if (data_flag==0)
                {
                        temp_data = *global_data_head;

                        while (temp_data->next_node!=NULL)
                                temp_data = temp_data->next_node;

                        p = malloc(sizeof(struct node_data));
                        p->globalID = recv_buff[k].globalID;
                        p->data = recv_buff[k].data;
                        p->next_node = NULL;

                        temp_data->next_node = p;
                }

                // Add node information in hash table...
                if (hash_table[hash_map]==NULL)
                {
                        hash_temp = malloc(sizeof(struct hash_node));

                        hash_temp->globalID=recv_buff[k].globalID;

                        for
(temp_data=*global_data_head;temp_data!=NULL;temp_data=temp_data->next_node)
                        {
                                if (temp_data->globalID == recv_buff[k].globalID)
                                        break;
                        }
                        hash_temp->data_location=temp_data;
                        hash_temp->next_node=NULL;
                        hash_table[hash_map]=hash_temp;
                }
                else
                {
                        hash_temp=hash_table[hash_map];
                        while (hash_temp->next_node!=NULL)
                        {
```

```
                                        hash_temp=hash_temp->next_node;
                                }

                                r_hash = malloc(sizeof(struct hash_node));

                                r_hash->globalID=recv_buff[k].globalID;

                                for
(temp_data=*global_data_head;temp_data!=NULL;temp_data=temp_data->next_node)
                                {
                                        if (temp_data->globalID == recv_buff[k].globalID)
                                                break;
                                }
                                r_hash->data_location=temp_data;
                                r_hash->next_node=NULL;
                                hash_temp->next_node=r_hash;
                        }
                }
        }

//      output_arr[migrating_node - 1] = to_proc;

        // Check if the peripheral nodes 'remain', if not, delete and insert in the internal node list...
        for (temp=*peripheral_head,prev=*peripheral_head;temp!=NULL;temp=temp->next_node)
        {
                p_flag=0;
                for (j=0;temp->neighboring_nodes[j]!=-1;j++)
                {
                        if (output_arr[temp->neighboring_nodes[j]-1] != myid)
                        {
                                p_flag=1;
                                break;
                        }
                }

                save_node = malloc(sizeof(struct own_node));

                if (p_flag == 0) // Remove peripheral node, and append the same to the internal node
list...
                {
                        if (temp == *peripheral_head)
                        {
                                temp1 = temp->next_node;
                                save_node = temp;
                                *peripheral_head=temp1;
                        }
                        else
                        {
                                prev->next_node = temp->next_node;
                                save_node = temp;
                        }

                        // Append to the internal node list...
                        temp_i = *internal_head;
                        if (*internal_head != NULL)
                        {
                                while (temp_i->next_node != NULL)
                                        temp_i=temp_i->next_node;
                        }

                        r = malloc(sizeof(struct own_node));
                        r->globalID = save_node->globalID;
                        r->internal_or_peripheral = 'i';
                        r->owning_proc = myid;
                        r->data_location = save_node->data_location;

                        for (j=0;j<10;j++)
                                r->neighboring_nodes[j] = save_node->neighboring_nodes[j];

                        for (j=0;j<10;j++)
                        {
                                r->shadow_for_procs[j] = -1;
                        }

                        r->next_node = NULL;

                        if (*internal_head != NULL)
                        {
                                temp_i->next_node = r;
                        }
                        else
                        {
                                *internal_head = r;
                        }
                }

                if (p_flag == 1)
                {
                        prev = temp;
                }
                free(save_node);
        }

        // Make an entry in the peripheral node list for the migrated task...
        temp_p = *peripheral_head;

        if (*peripheral_head != NULL)
        {
                while (temp_p->next_node != NULL)
                        temp_p=temp_p->next_node;
        }

        r = malloc(sizeof(struct own_node));

        r->globalID = migrating_node;
        r->internal_or_peripheral = 'p';
        r->owning_proc = myid;

        hash_map = ((int)pow(3,migrating_node))%HASH_TABLE_LENGTH;
        for (hash_temp=hash_table[hash_map];hash_temp!=NULL;hash_temp=hash_temp->next_node)
        {
                if (hash_temp->globalID == migrating_node)
                        break;
        }
        r->data_location = hash_temp->data_location;

        for (j=0;j<10;j++)
                r->neighboring_nodes[j]=-1;

        for (j=0;input_arr[migrating_node-1][j]!=0;j++)
                r->neighboring_nodes[j]=input_arr[migrating_node-1][j];

        for (j=0;j<10;j++)
```

```
                r -> shadow_for_procs[j] = -1;

            for (k=0;r -> neighboring_nodes[k]!=-1;k++) // Populating shadow_for_procs[] arr...
            {
                insert_shadow_flag=0;
                if (output_arr[r -> neighboring_nodes[k] - 1] != myid)
                {
                    for (j=0;r -> shadow_for_procs[j]!=-1;j++)
                    {
                        if       (output_arr[r       ->       neighboring_nodes[k]       -1]==r-
>shadow_for_procs[j])
                            insert_shadow_flag=1;
                    }

                    if (insert_shadow_flag==0)
                        r  ->  shadow_for_procs[j]  =  output_arr[r  ->  neighboring_nodes[k]  -
1];
                }
            }

            r->next_node = NULL;

            if (*peripheral_head != NULL)
            {
                temp_p->next_node = r;
            }
            else
            {
                *peripheral_head = r;
            }

            // Updating the shadow_for_procs[] array for the peripheral nodes...
            for (temp_p=*peripheral_head;temp_p!=NULL;temp_p=temp_p->next_node)
            {
                for (j=0;j<10;j++)
                    temp_p -> shadow_for_procs[j] = -1;

                for  (k=0;temp_p  ->  neighboring_nodes[k]!=-1;k++)  //  Populating  shadow_for_procs[]
arr...
                {
                    insert_shadow_flag=0;
                    if (output_arr[temp_p -> neighboring_nodes[k] - 1] != myid)
                    {
                        for (j=0;temp_p -> shadow_for_procs[j]!=-1;j++)
                        {
                            if   (output_arr[temp_p   ->   neighboring_nodes[k]   -1]   ==
temp_p->shadow_for_procs[j])
                                insert_shadow_flag=1;
                        }

                        if (insert_shadow_flag==0)
                            temp_p  ->  shadow_for_procs[j]  =  output_arr[temp_p  ->
neighboring_nodes[k] - 1];
                    }
                }
            }
        }
        else
        {
        //      output_arr[migrating_node-1] = to_proc;

            // Updating the shadow_for_procs[] array for the rest of the procs...
```

```
		for (temp_p=*peripheral_head;temp_p!=NULL;temp_p=temp_p->next_node)
		{
			for (j=0;j<10;j++)
				temp_p -> shadow_for_procs[j] = -1;

			for (k=0;temp_p -> neighboring_nodes[k]!=-1;k++) // Populating shadow_for_procs[]
arr...
			{
				insert_shadow_flag=0;
				if (output_arr[temp_p -> neighboring_nodes[k] - 1] != myid)
				{
					for (j=0;temp_p -> shadow_for_procs[j]!=-1;j++)
					{
						if (output_arr[temp_p -> neighboring_nodes[k] -1] ==
temp_p->shadow_for_procs[j])
							insert_shadow_flag=1;
					}

					if (insert_shadow_flag==0)
						temp_p -> shadow_for_procs[j] = output_arr[temp_p ->
neighboring_nodes[k] - 1];
				}
			}
		}
	}

	MPI_Barrier(comm);

	// Update the buffer_size_for_communication array for all procs...
	for (j=0;j<num_procs;j++)
	{
		buffer_size_for_communication[j] = 0;
	}

	for (temp_p=*peripheral_head;temp_p!=NULL;temp_p=temp_p->next_node)
	{
		for (j=0;temp_p->shadow_for_procs[j]!=-1;j++)
		{
			buffer_size_for_communication[temp_p->shadow_for_procs[j]]++;
		}
	}
}
```

Appendix D

This appendix shows the header file used.

```
/* header.h */

# include <stdio.h>
# include <stdlib.h>
# include <math.h>
# include <time.h>
# include "mpi.h"

# define MAX_STR_LENGTH 250
# define COLUMN_WIDTH_OF_INPUT_ARRAY 10
# define HASH_TABLE_LENGTH 10
# define MAX_SIZE_FOR_RECVBUFFER 10
# define MIGRATING_NEIGHBORS 10

struct node_data
{
		int globalID;
		int data;
		int most_recent_data;
		struct node_data *next_node;
};

struct buffer_data_node
{
		int globalID;
		int data;
};

struct own_node
{
		int globalID;
		char internal_or_peripheral;
		int owning_proc;
		struct node_data *data_location;
		int neighboring_nodes[10];
		int shadow_for_procs[10];
		struct own_node *next_node;
};

struct hash_node
{
		int globalID;
		struct node_data *data_location;
		struct hash_node *next_node;
};

void InitializeGraph(int *, int *, int *);
void InitializeInputArray(int **, int, int, int *);
void InitializeOutputArray(int *, int);
void InitializeGlobalDataList(struct node_data **, int);
void InitializeNodeLists(struct own_node **, struct own_node **, struct node_data **, struct hash_node *[], int **, int
*, int, int, int *, MPI_Comm);
void InsertShadowsIntoHashTable(struct own_node **, struct node_data **, int *, struct hash_node *[], MPI_Comm);
void ComputeOverNodes(struct own_node **, struct own_node **, struct node_data **, struct hash_node *[], struct
buffer_data_node *[], int, int *,void (*)(), MPI_Comm, int, int);
```

```
void SimulatorFunction(struct node_data **, struct node_data **, int, int, int);
void CommunicateShadows(struct buffer_data_node *[], struct buffer_data_node *[], int *, int, struct node_data **, MPI_Comm);
void task_migrate(int, int, int, int *, struct own_node **, struct own_node **, struct hash_node *[], struct node_data **, int **, int, int *, MPI_Comm);
int GetLoadRebalancingParameters(double *, int **, int **, int, MPI_Comm);
int GetMigratingNode(struct own_node **, int, int, int **, int *);
int UpdateMigratingNodeInfo(int **, int, struct own_node **, int *, int **);
```