TASK MAPPING IN MANY-CORE SYSTEMS

A Project Report
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in
The Department of Electrical and Computer Engineering

by
Santhosh Ramaiah
B.E., Visvesvaraya Technological University, India, June 2012

Acknowledgments

I take this opportunity to thank my advisor and committee chair, Dr. Jerry Trahan, for his invaluable advice, suggestions, and support during the course of this work. It is very inspiring to see his painstaking attention to detail and commitment to students. I have been fortunate to have his insightful guidance.

I would like to thank my committee members: Dr. Ramachandran Vaidyanathan, and Dr. Suresh Rai. Thanks for their patience in reading this project report and their comments. I would also like to thank other faculty for their advice and inspiration.

I want to express my immense gratitude to my parents, Ramaiah S T and Savitha M. They raised me and poured me with love unconditionally and have always been supporting and encouraging me during my Master's program. Words are not enough to express my love to you. Last but not least, I would like to thank all my friends who supported me and made this journey memorable.

Table of Contents

List of Tables

List of Figures

Abstract

The past several decades have experienced tremendous growth in the computing industry, thanks to advances in IC technologies. The current trend is leading from single core chips to many-core chips with as many as several hundred cores on a single chip. Communication between cores is passed through a network of routers to handle bandwidth constraints and latency. This parallel computation enables us to run computationally intense applications with ease. As chip manufactures try to integrate large number of cores on a single chip, energy consumption of these chips keeps growing. One relevant problem is mapping applications onto many-cores aiming at low communication energy consumption. Our main aim of this project is to propose and compare algorithms for obtaining low energy mapping onto many-cores. To cope with complex dynamic behavior of applications, it is relevant to perform task mapping during runtime, hence, heuristic approaches are taken instead of exhaustive search of the solution space. Some heuristic-based algorithms proposed previously consider placing tasks in order of largest communication volume. But the Placed Communication First (PCF) mapping algorithm introduced by Trahan and Elbidweihy and refined here performs task mapping based on largest volume of communication to already placed tasks rather than overall communication volume. Experimental results show that PCF provides average reduction of 41% communication cost compared to other algorithms on real application task graphs. To explore the problem space, we also consider randomly generated task graphs and make use of a scheduler with region selection in addition to mapping, showing similar results. The PCF mapping algorithm outperforms other mapping algorithms in many situations.

Chapter 1: Introduction

Almost since the computer era began, the computer industry has been producing high performance parallel computing. The distinguishing feature of parallel computation is performing multiple calculations simultaneously. Early computer systems through 1970-1990 exploited several types of parallelism. In *bit-level parallelism*, multiple bits are computed during the execution of a single instruction. In *instruction level parallelism*, multiple instructions are executed simultaneously.

Gordon Earle Moore, cofounder of Intel, predicted in 1965 that the "number of transistors in a dense integrated circuit doubles approximately every two years" [19]. This famous Moore's Law held good as processor manufacturers were able to produce generations of processors with a doubling of clock rate as the number of transistors doubled [19]. High-level programming languages and sophisticated compilers were able to convert these hardware advancements to performance without much changing the sequential code. However, the increase in clock speed caused higher power consumption and heat dissipation [20]. This led to development of multi-core chip designs able to run at lower frequency, thus reducing heat and energy consumption [20].

Many-core systems consisting of hundreds of cores on a single chip are on the horizon. This massively parallel computational resource can be used for solving highly complex and computationally demanding applications, such as real-time audio/video encoding, image processing, facial recognition engines, on-demand data encryption, real time data processing, and many more [17].

1.1 Network on Chip Device

A network-on-chip (NoC) is an intra-chip communication infrastructure, proposed as replacement to bus based designs [12]. State-of-the-art many-core systems deploy a NoC to enable data transfer between processing elements or cores. NoCs have replaced bus-based interconnects that are predominantly found in single and multi-core chips because NoC can avoid congestion when multiple cores communicate with each other. In a mesh-based topology, each router connects to a core, and the routers are connected by communication channels or links [13, 5, 6, 14]. A router and a core are placed in a defined region called a tile [13]. The performance scalability, modularity, and communication parallelism make NoCs a promising communication resource for future many-core chips [15].

1.2 Mapping Problem

An application intended to run on a many-core is typically described as a set of concurrent tasks where each task runs on one core. Due to the large number of cores in a many-core system, we can run multiple applications simultaneously. For a many-core system consisting of hundreds of cores, the problem of efficiently scheduling applications is a significant challenge. Scheduling consists of region selection and mapping. The region selection phase selects a region large enough to hold all tasks in an application. The mapping phase places these tasks in cores in selected region. The problem of mapping tasks to cores so as to minimize communication energy consumption is NP-Hard [2]. Due to this, various heuristic-based approaches have been employed for finding a good solution. In this work, we propose and evaluate mapping algorithm for its performance and communication energy costs. Our work focuses on the online approaches

where information about applications is not known until they arrive. The detailed explanation regarding mapping is given in Chapter 3.

We consider several existing algorithm and focus on the Placed Communication First (PCF) algorithm. This work refines these algorithm to improve their performance. To analyze these algorithms, we run experiments for both randomly generated task graphs and real application task graphs. On average, the communication cost of other mapping algorithm is 72% more than with PCF and execution time for other algorithms is 30% faster than with PCF for real application task graphs. The results for randomly generated task graphs show that the PCF mapping algorithm yields better communication cost than other mapping algorithms in most situations. We also made use of a scheduler with region selection in addition to mapping, and the results show that PCF algorithm outperforms other algorithms in this context as well.

1.3 Prior Work

In the literature, a number of previous works addresses the energy consumption of many-cores. In [14], Marculescu *et al*. described several research problems in NoC architectures and applications. They divided the problems into five categories: application characterization, communication paradigm, communication infrastructure, analysis, and solution evaluation. On application characterization of the target application and its associated traffic patterns, having a good model for application architecture and application partitioning helps to find a better solution under performance and energy constraints. Toward optimizing energy consumption, the authors described problems in application mapping, application scheduling and traffic modeling. On communication paradigm, they described problems in packet routing, switching technique, QoS and congestion control, and power and reliability. In communication infrastructure, they

gave problem definitions in design of topology, router, channel, and floor planning. The analysis and solution evaluation were described mainly in terms of power consumption.

Marcon *et al*. [13] compared both static and dynamic mapping techniques such as exhaustive search (ES), simulated annealing (SA), and tabu search (TS), and proposed two greedy heuristics called largest communication first (LCF) and greedy incremental. They also considered mixed approaches of both a stochastic method (SA, TS) and LCF. They ran their simulation on a Java-based environment considering different many-core size and application traffic patterns similar to real applications. The authors obtained energy savings up to 40% for their mixed approach giving better performance than other algorithms considered.

Chou *et al*. [5, 6] proposed dynamic mapping of tasks to homogenous many-cores with multiple voltage levels. Their incremental run-time approach gives solutions for region selection and application mapping. Dealing with the region selection problem for the incremental mapping process, they needed to look at both minimizing the communication cost of the incoming application and, at the same time, minimizing the communication cost overhead of any additional incoming application. For this they compared mapping algorithms such as Euclidean minimum (EM), fixed center (FC), neighbor frontier (NF), random frontier (RF), worst case (WC), and best case (BC) heuristics and found that the NF heuristic is most suitable for region selection.

Mandelli *et al*. [11] and Ost *et al*. [17] gave a unified approach for evaluating dynamic mapping heuristics and proposed task mapping heuristics targeting energy consumption in many-core architectures. The authors gave two heuristics DN and LEC-DN. The DN heuristic maps the requested task as closely as possible to the already mapped task with which it communicates

based on proximity (in number of hops) cost function. The LEC-DN similarly does the same as DN, but employs both proximity and communication energy. They also included a pre-map function to map multiple tasks to the same core. Their results show that the algorithm reduces 51% of energy cost on average and with execution time overhead of 18% when compared with mapping one task per core. Sahu and Chattopadhyay [18] surveyed various dynamic and static mapping algorithms. The authors also evaluated algorithm performance for real application task graphs. Bender *et al*. [1] gave a different approach to map jobs to cores while reducing the number of communication hops in a bandwidth-constrained environment. The algorithms used are variants of the Manhattan median algorithm.

Another way of reducing power consumption is by reducing the voltage of cores. Seo *et al*. [20], He and Mueller [8], and Niu *et al*. [16] talked about various ways of scaling voltage and switching of cores to reduce energy consumption. Dynamic voltage (and frequency) scaling and dynamic power management are employed to obtain best trade-off between system performance and power consumption. Seo *et al*. [20] showed dynamic repartitioning of assigned tasks yields energy savings. Shieh and Pong [21] also took into consideration the overhead of voltage transitions and said that, taking this into consideration, one can achieve better results. Manolache *et al*. [12] and Mahabadi *et al*. [10] combined a mapping algorithm with voltage scaling techniques and applied it to real application task graphs. This authors claimed that this mixed approach reduces energy consumption.

Similarly, Carvalho *et al*. [2, 3, 4] evaluated both static and dynamic mapping techniques in a homogenous many-core environment. They proposed four heuristics: minimum average channel load (MACL), minimum maximum channel load (MMCL), path load (PL), and best neighbor (BN), concentrating on reducing channel usage peaks and reducing occurrence of hot

spots. Their approach of evaluation was based on channel occupancy and packet latency of the NoC architecture and achieved up to 31% smaller channel load and up to 22% smaller packet latency. Murali and De Micheli [15] proposed a heuristic (NMAP) for bandwidth constraints in links connecting cores, which is used to solve single path routing between cores to splitting the traffic between the cores to reduce bandwidth constraints on links. Srinivasan and Chatha [22] proposed a heuristic for both bandwidth and latency constraints called MOCA. They compared against NMAP and MILP formulation producing 14% energy saving less of the result obtained by other algorithms considered for bandwidth constraints and outperforming in latency constrained cases. Lee *et al*. [9] proposes an algorithm to schedule tasks under local memory constraints.

1.4 Outline

In Chapter 2, we will explain the basic system model of a network on chip architecture, application characteristics, and the NoC energy model that we use. Chapter 3 presents a detailed explanation and working of the various mapping algorithm. In Chapter 4, we will show experimental results that compare various algorithms used for mapping. The comparisons are based on applying mapping algorithms to randomly generated task graphs, real application task graphs, and, finally using a scheduler. Chapter 5 summarizes the results obtained and gives some areas to work on in the future.

Chapter 2: System Model

2.1 Network on Chip Architecture

This project focuses on a many-core system with a 2D mesh as interconnection among tiles. Each tile consists of a homogeneous core (or processing element) and a router. Only one task at a time can be run on a core. A minimal transmission control protocol (TCP) routing stack is assumed for communication between routers for transmitting both control and data messages. This TCP protocol ensures the communication is reliable and error checked.

The cores operate at the same voltage and frequency. As seen in Figure 2.1, a specialized processor is included in the NoC architecture, and it is optimized to run the operating system. The operating system includes an application scheduler with a mapping manager (MM) as a part of the scheduler. The MM runs a dynamic mapping algorithm to map the incoming tasks of an application to various cores. This is a centralized mapping mechanism.

Functioning of Mapping Manager

Online scheduling handles applications as they arrive. The process of scheduling consists of region selection and mapping. The region selection uses the help of a free space manager. The main function of the free space manager is to gather necessary information relating to available resources of the cores. It basically keeps track of whether each core is occupied or free. The region selection algorithm searches the free space area to find a rectangle that is large enough to hold an application [7]. This work mainly focuses on mapping rather than region selection. For further information, refer to Elbidweihy and Trahan [7]. The mapping is done by a mapping manager (or mapper) after region selection. The mapping manager assigns the tasks of the

incoming application to cores in the selected region. Chapter 3 explains the mapping of tasks in detail.

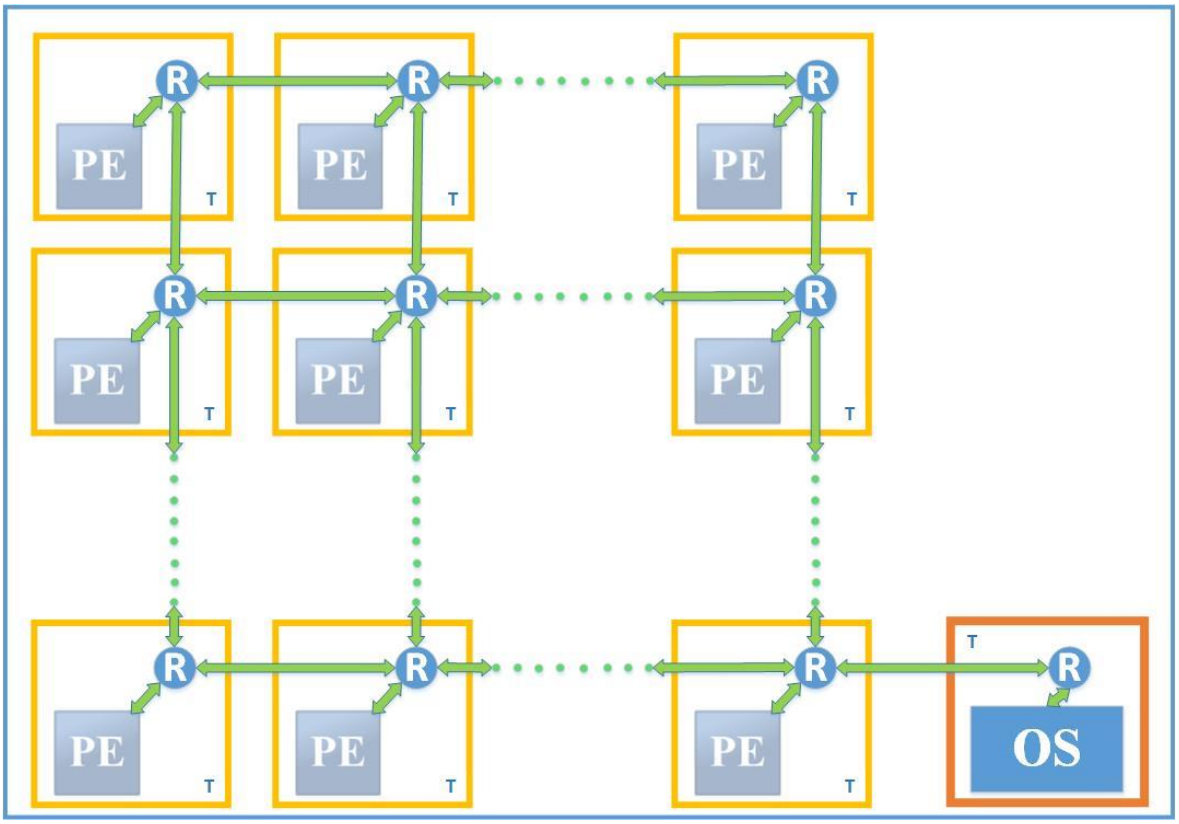


Figure 2.1: Homogenous 2D NoC mesh.

(OS-Operating System, PE-Processing Element, R-Router, T-Tiles)

2.2 Application Characteristics

An incoming application is represented by a *communication weighted graph* (CWG) [13]. The nodes represent the different tasks and edges represent the inter-task communication. The volume of communication between a pair of tasks is represented by the weight of the corresponding edge. Edges are undirected, and the weight on an edge captures communication volume in both directions. Figure 2.2 shows an example CWG.

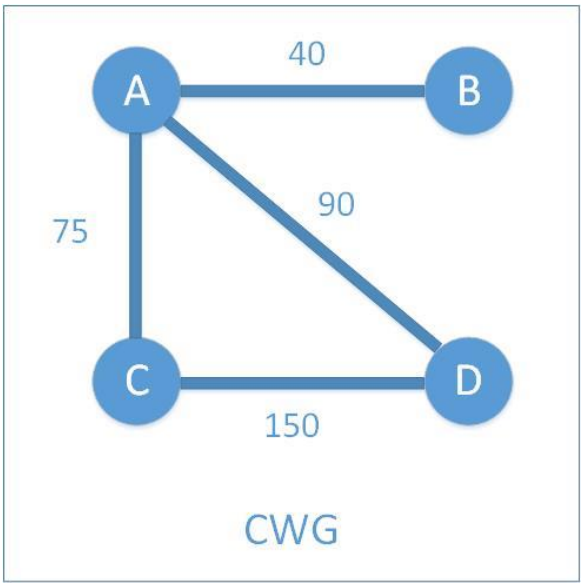


Figure 2.2: Example CWG.

2.3 NOC Energy Model

Energy consumption occurs in all elements including interconnection network, routers, and cores. This work focuses only on dynamic energy, as mapping algorithms do not affect static energy. The dynamic energy is constituted mainly by inter-process communication. The communication energy that occurs between cores is measured in terms of bit transitions. This work employs the bit energy consumption model [24], which is a well-accepted energy model, used for instance in [13, 5, 6, 15]. In this model, E_{bit} is an estimate of the overall dynamic energy consumption due to bit transition from state '0' to '1' or vice versa. E_{bit} is the sum of dynamic energy consumed on interconnects, logic gates, and router buffers. E_{Rbit} is the energy consumed by a routing buffer, E_{Cbit} is the dynamic energy consumed on interconnect between each router and its local cores, and E_{Lbit} is the dynamic energy consumed on interconnect between adjacent tiles. The dynamic energy consumed sending one bit from tile *i* to tile *j* that are *h* hops apart is given in Equation 1.

$$E_{bit}(i, j) = (h + 1)\, E_{Rbit} + 2 \times E_{Cbit} + h \times E_{Lbit} \quad (1)$$

For a given many-core architecture, E_{Rbit}, E_{Cbit}, and E_{Lbit} are constant. Because we are seeking to compare different mapping algorithms on the same many-core, we can therefore simplify Equation (1) to Equation (2). Here the number of hops is equal to the Manhattan distance between tile *i* and tile *j*.

$$E'_{bit}(i, j) = h \quad = \mathrm{MD}\,(i, j) \quad (2)$$

For tasks *a* and *b* mapped to cores *i* and *j*, respectively, the communication cost between them is the product of weight $w(a, b)$ of that edge and the Manhattan distance between core *i* and core *j*, as shown in Equation (3).

$$E'_{task}(a, b) = w\,(a, b) \times \mathrm{MD}\,(i, j) \quad (3)$$

For a given application with *n* tasks, the communication energy is the summation of energy consumed by all pairs of communicating tasks, as shown in Equation (4).

$$E_{App} = \sum_{a=1}^{n} \sum_{b=1}^{n} E'_{task}(a, b) \quad (4)$$

Finally, the total amount of communication energy consumption occurring in a many-core is the summation of dynamic energy consumed all applications.

Chapter 3: Mapping Algorithms

The mapping algorithms in this work focus on an online setting. Here, the available time to map the tasks of an application is limited, hence the algorithms cannot evaluate all the mapping possibilities. We take a heuristic-based approach to get solutions for the mapping problem. In this work, an arriving application includes information regarding total number of tasks and their communication weights. The scheduler (in general terms for region selection) looks into this information and selects a region large enough to hold all tasks in an application. This region is a rectangle for the mapping algorithms considered here.

The scheduler passes the CWG of an application and dimensions of the selected region to the mapper. The mapper with this information performs mapping of all the tasks to cores in the selected region. This chapter presents some existing mapping algorithms and notes improvements that we have made to their efficiency.

Implementation Details

The input to a mapping algorithm consists of a CWG in the form of a weighted adjacency matrix T of n (total number of tasks in an application) tasks, with tasks sorted in decreasing order of their communication volume (total amount of communication occurring for each task t_i to all other tasks in the application), and an $r \times c$ mesh of cores such that $rc \geq n$. The output consists of, for each task t_i, mapping $map\ (t_i) = (p_i, q_i)$, where (p_i, q_i) is the location of the core to which t_i is mapped. The implementation details for algorithms in this chapter are in the appendix.

3.1 Euclidean Minimum

In the Euclidean Minimum (EM) mapping technique, a task is placed at the free core with minimum Euclidean Distance to the center of the allocated cores [6]. Initially, the center of the allotted mesh is chosen for mapping the first task, as shown in Figure 3.1. As for placing the next task onto the mesh of cores, we update the center (p_c, q_c) by recalculating the arithmetic mean of the locations of the mapped cores, then select a free core (p, q) with minimum Euclidean distance to the updated center. The pseudo-code is described in Appendix B. Figure 3.1 shows the sequence of task mapping for a randomly generated task graph, in which initially the task starts to grow in the fourth quadrant of core mesh. This causes the center of the arithmetic mean of the mapped core to be pulled to the fourth quadrant, hence filling up the fourth quadrant and then growing upwards.

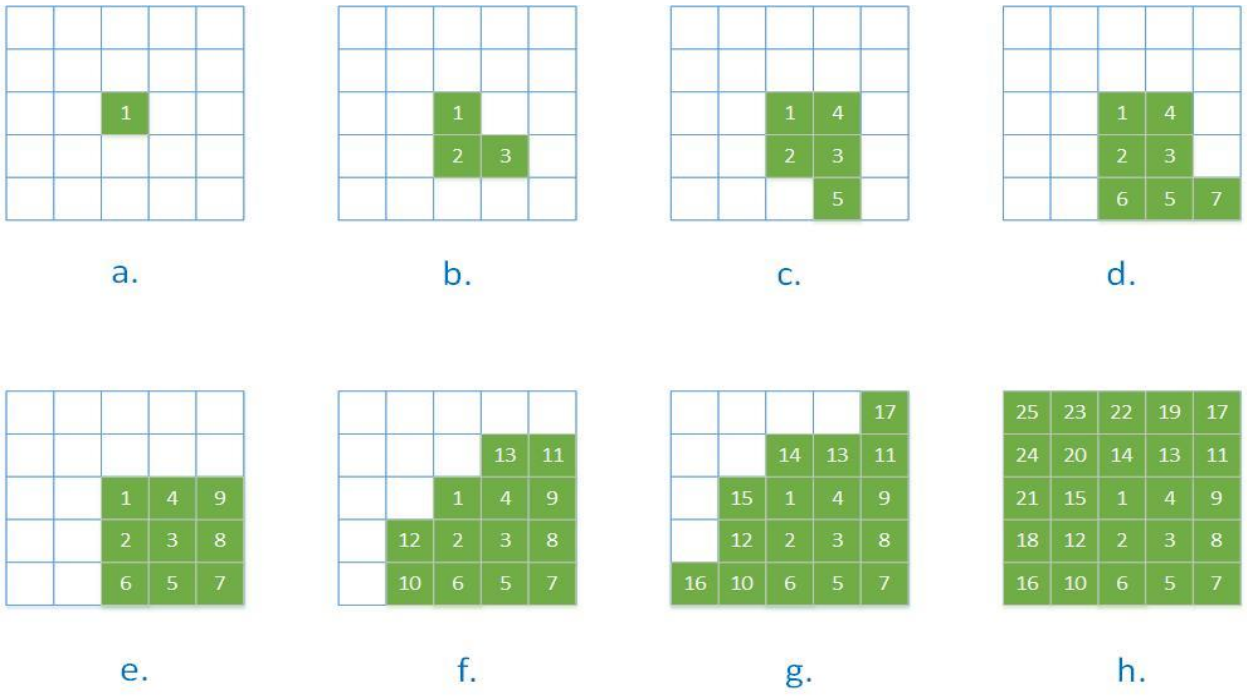


Figure 3.1: Illustration of Euclidean Minimum algorithm on 5 x 5 mesh.

3.2 Fixed Center

The Fixed Center (FC) technique places each task after the first at the free core with minimum Manhattan Distance to the first allocated core [6]. Set the location of the first mapped core as the fixed center (p_c, q_c) and then start mapping other tasks. Unlike the Euclidean Minimum algorithm, the fixed center maps tasks around the center of mesh of cores, as shown in Figure 3.2. The pseudo-code is described in Appendix C. The implementation is similar to Euclidean Minimum but the center is fixed and Manhattan distance is considered instead of Euclidean distance. The performance of this algorithm is similar to that of Euclidean Minimum.

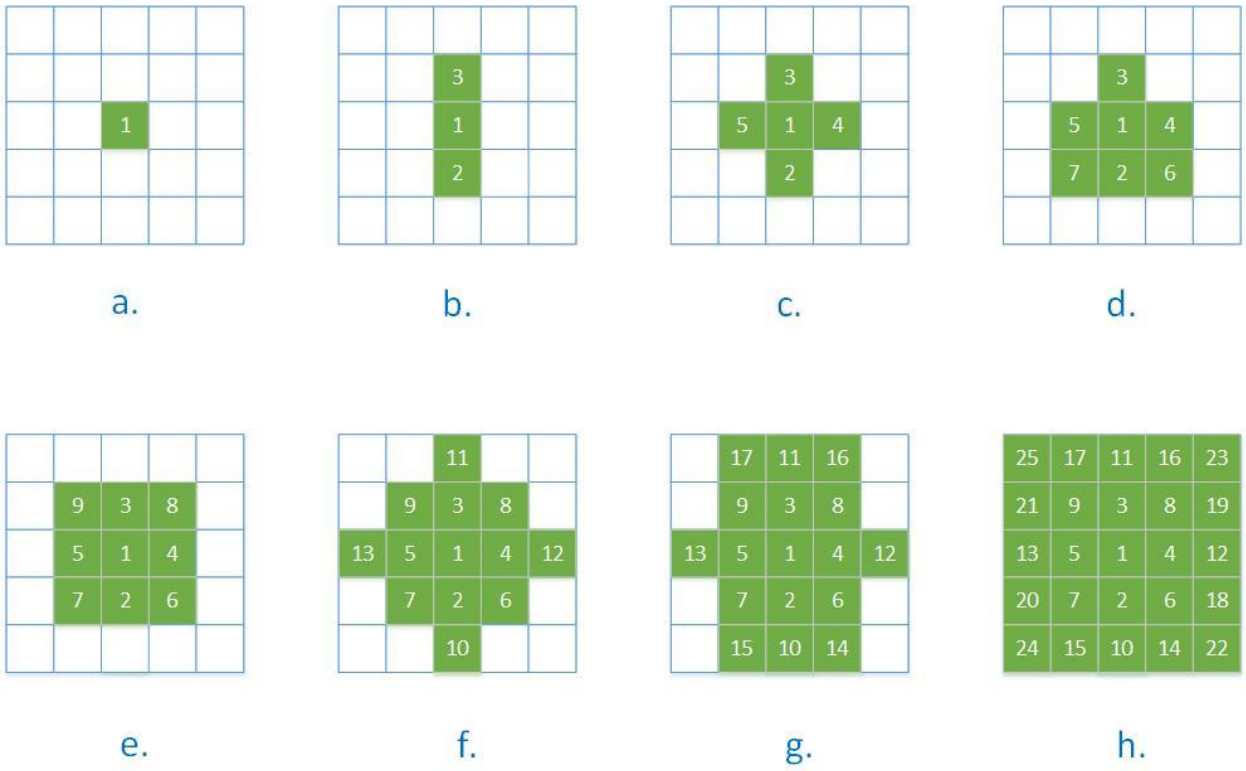


Figure 3.2: Illustration of Fixed Center algorithm on 5 x 5 mesh.

3.3 Neighbor-Aware Frontier

Neighbor-Aware Frontier (NF) places a task at the free core on the frontier of allocated cores with minimum number of free neighbors [6]. The frontier list consists of free cores that are adjacent to mapped cores. An illustration of the algorithm is shown in Figure 3.3. The task placement may look similar to Fixed Center but there is a lot of difference on how a task is placed. The pseudo-code is described in Appendix D. The implementation is similar to Euclidean Minimum, but instead of considering distance to average center the NF algorithm uses the frontier list to count neighbors. Based on the count of neighbors, it places the task onto a free core. The performance of the algorithm is similar to Euclidean Minimum.

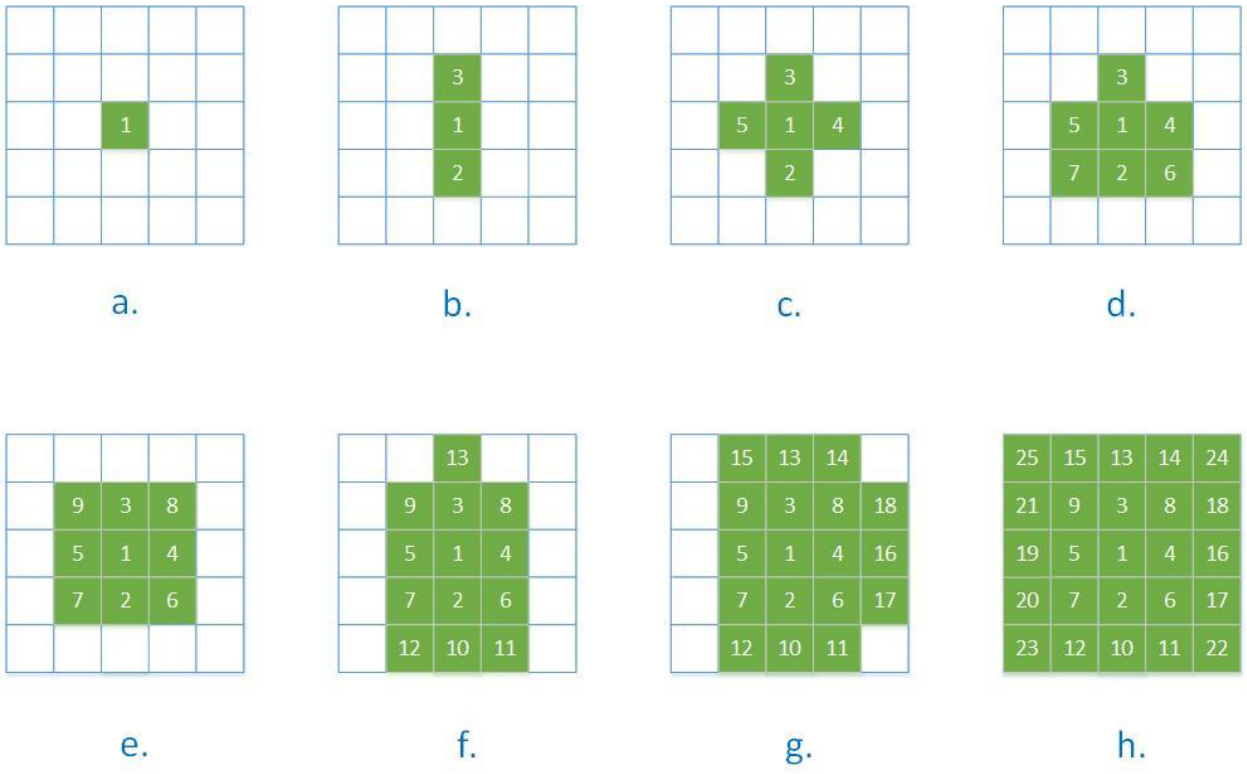


Figure 3.3: Illustration of Neighbor-Aware Frontier algorithm on 5 x 5 mesh.

3.4 Largest Communication First

Largest Communication First (LCF) algorithm attempts to match the number of edges incident on a task to the number of links to a core. So, it tries to map tasks with ≥ 4 edges to cores with 4 links, tasks with 3 edges to cores with 3 links, and tasks with ≤ 2 edges to cores with 2 links. LCF also gives priority to tasks with the most communication for mapping over tasks with less communication [13]. The LCF algorithm is given in Table 3.1.

```
// 1nd Step
createTheCoreTypeList (NoC Topology);
// 2rd Step
assignTasksToCoresTypes ();
// 3th Step
mapTasksToNoC (listOfTasksAssignedTo4PortCore);
mapTasksToNoC (listOfTasksAssignedTo3PortCore);
mapTasksToNoC (listOfTasksAssignedTo2PortCore);
mapTasksToNoC (remainingTasks);
```

Table 3.1: Algorithm for LCF [13].

The LCF algorithm consists of three steps. The first step is to create a set of cores of different types, based on number of links to neighbors in the NoC architecture. For example, a mesh has three core types, with 2, 3, and 4 links (*C2*, *C3*, and *C4*). LCF will use these sets to associate task communication needs with the core communication capacity [13]. The second step assigns each task to a core type, based on whether it has at most two, three, or at least four edges to other tasks in the application graph. If the number of edges for a task is ≤ 2, and the number of available cores with two links is greater than zero, then this task is assigned to list *A2* for this core type *C2* and number of cores with two links is decremented. However, if no cores with two links are available, then the task is pushed to waiting list *W2*. Similarly, the algorithm handles tasks with 3 or ≥4 edges [13].

The last step is the mapping of tasks to cores, considering the type of each core. LCF algorithm employs four methods to implement this step. First, tasks in *A4* are mapped to cores with 4 links. Second, tasks in *A3* are mapped to cores with 3 links. Third, tasks in *A2* are mapped to cores with 2 links. Finally, the remaining tasks (in *W4*, *W3*, and *W2*) are mapped to available cores, regardless of type [13]. During the mapping procedure there are three possible situations. The first situation is where a task has no communication with placed tasks. For this, we will place the task at the free core nearest to NoC center. The second situation is where a task communicates with a single placed task. For this situation, we will place the task in the free core nearest to the communicating task. The last situation is when a task communicates with more than one placed task. For this, we will place the task at the core resulting in lowest energy. For further information, refer to [13].

The in-depth description is given in pseudo-code in Appendix E. Figure 3.4 shows the sequence of task placement resulting from LCF, which differs significantly from previous algorithms. An LCF mapping depends on the specific CWG graph of an application. In the preceding examples, 1 through 25 were the tasks in order of decreasing communication volume. In Figure 3.4, however, they are tasks in the order in which they were placed (*A4* before *A3* before *A2* before *W4*, etc.). In Figure 3.4 (f) we can see clearly by its placement that task 10 communicates with 3 or more tasks. But tasks 11, 12, and 13 communicate with only other two tasks.

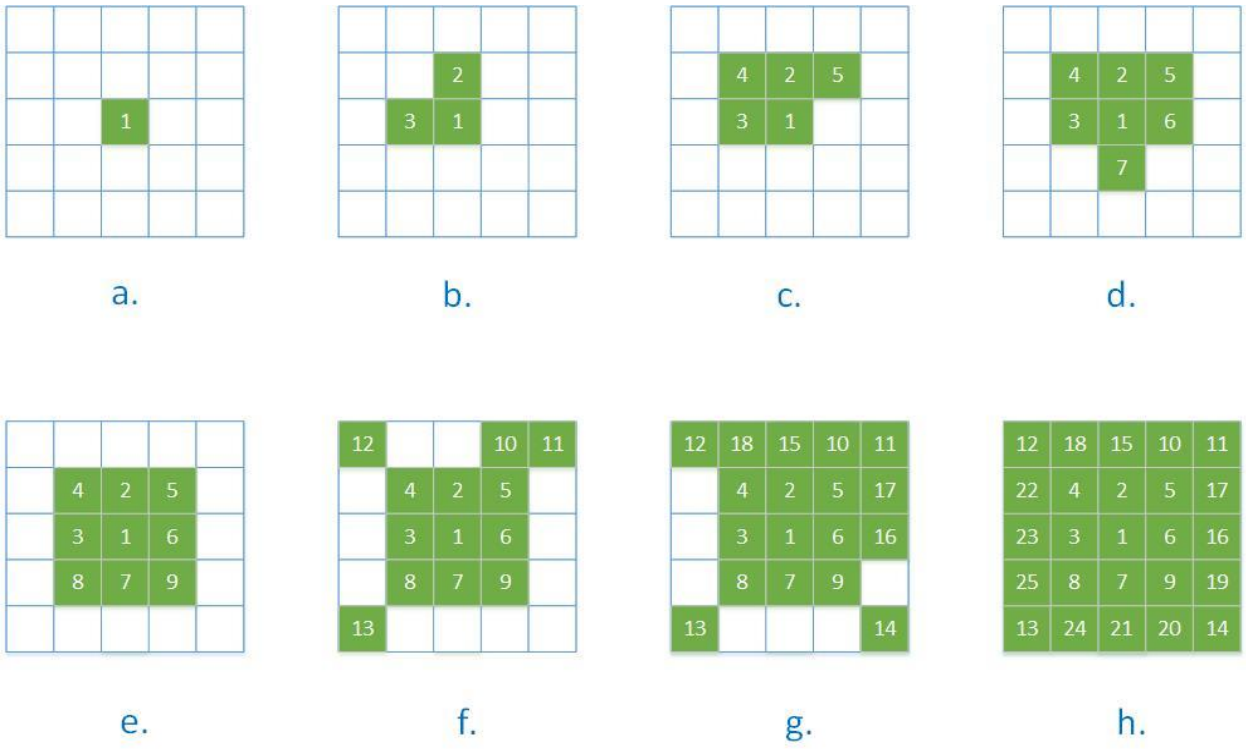


Figure 3.4: Illustration of Largest Communication First algorithm on 5 x 5 mesh.

3.5 Placed Communication First

The main focus of Placed Communication First (PCF) [23] is to place a task with a large amount of communication to already-placed tasks rather than waiting behind tasks with less communication to placed tasks (even if they have a large overall communication volume). This method closely parallels that of Murali and De Micheli [15] and Ost *et al*. [17], but implementation details differ from theirs. Initially, place the task with largest overall communication volume. Afterwards, among the unplaced tasks, choose and place the one with largest communication volume to the tasks that have already been placed.

While placing a task at this stage PCF evaluates all the frontier list of cores for communication energy cost and then selects the core with lowest cost to place the chosen task. In the implementation, array *communplaced* contains the unplaced tasks and also details the communication volume to already placed tasks. This communication volume to placed tasks is updated after every placement of a task. However, if the next chosen task is not communicating with any other placed task, then PCF chooses a free core which has a minimum Manhattan distance to the fixed center. PCF iterates through tasks until placing them all.

Our contribution PCF are as follows. First, we added a communication adjacency list (*commun_adjlist*) representation of the CWG to the communication adjacency matrix so as to reduce the computational overhead of maintaining list *communplaced*. Second, PCF algorithm selected the core at which to place a task based on energy cost to the one placed task with which it communicates to most. To increase energy savings, we consider placing a task at the free core with lowest cost with respect to all of the already placed tasks. Third, several array structures were modified to run the algorithm faster in MATLAB implementation. The detailed pseudocode

is given in Appendix F. Figure 3.5 gives the sequence of task placement resulting from PCF. The figure shows a clear distinction to other algorithms, as it does not follow the same pattern as other algorithms. The placement of tasks is purely based on largest communication volume to placed tasks.

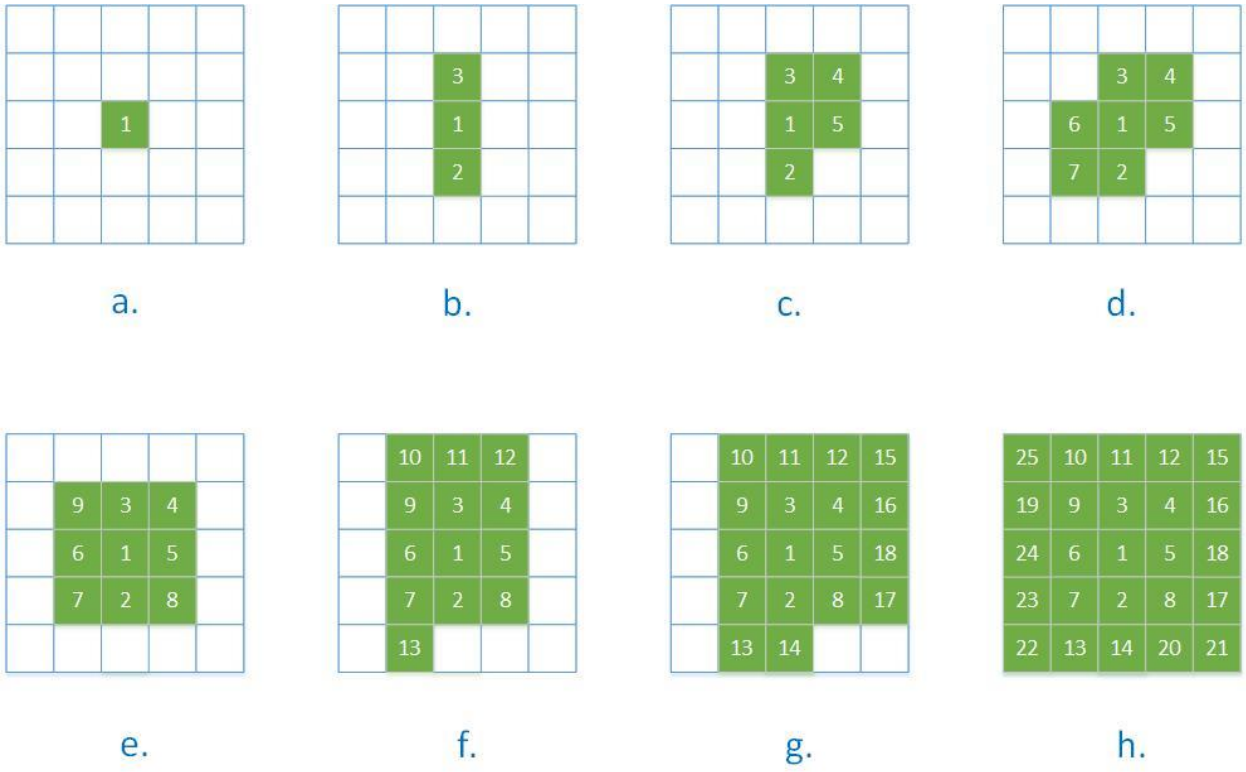


Figure 3.5: Illustration of Placed Communication First algorithm on 5 x 5 mesh.

In Figure 3.5 the numbering of task from 1 to 25 shows the order which a task is placed on to mesh of cores, but this order is not by overall communication volume. This is because the PCF algorithm places the task with large amount of communication with already placed tasks and does not consider communication with unplaced tasks.

Chapter 4: Experimental Results

4.1 Simulation Results for Mapping only

The applications in these experiments include both randomly generated task graphs and real application task graphs. The number of edges generated randomly for an application is controlled by *edge ratio* or *edge percentage*.

The *edge ratio* is the ratio of total number of edges to total number of tasks in an application. For example, assume application size is 100, then edge ratio of 0.2 yields (100 * 0.2) = 20 edges for the generated task graph.

The *edge percentage* is the percentage of total number of edges possible for a given number of tasks. For example, a task graph with 100 tasks can have at most (100 * ((100-1) / 2)) = 4950 edges. Then edge percentage of 20% yields (4950 * 0.2) = 990 edges for the task graph.

We use two measures to characterize the number of edges in a graph because of distortions in some cases. For example, when *edge ratio* is large (in range 20), and an application has a *small* number of tasks, the task graph is fully connected. The *edge percentage* is more suitable for representing applications with small number of tasks. Hence two characteristics are taken into account in this work to better (in terms of characteristics) construct randomly generated task graphs. Real application task graphs generally have low edge ratio or edge percentage.

The real application task graph benchmarks used are given in Figure 4.1 [18]. These are a) DVOPD, b) VOPD, c) MPEG -4, d) PIP, e) MWD, f) 263enc mp3dec, g) mp3enc mp3dec and h) 263dec mp3dec.

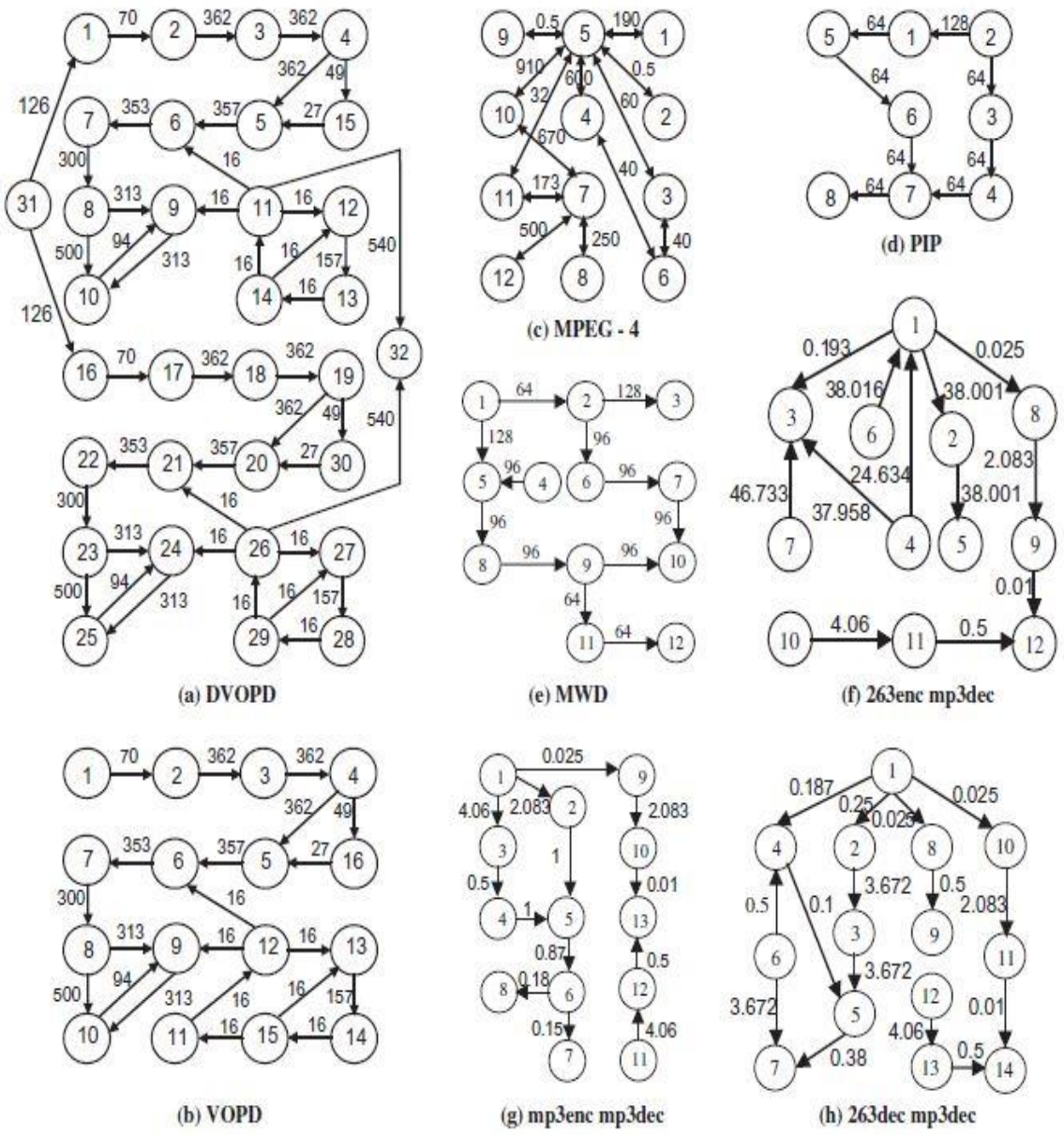


Figure 4.1: Application task graphs with communication bandwidth (MB/s) [18].

In Figure 4.2, the task graph communication of VOPD (Video Object Plane Decoder) is shown in detailed view showing the function of each task. This shows how an application is converted to a task graph with nodes and edges.

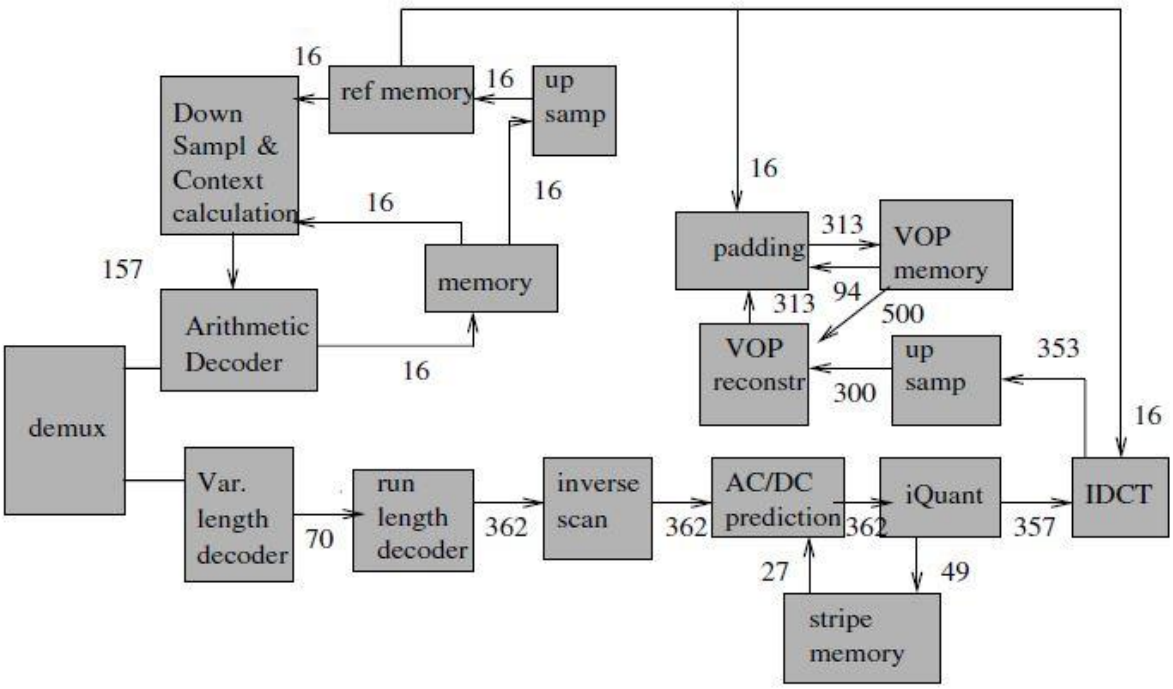


Figure 4.2: Block diagram of VOPD, with communication BW (in MB/s) [15].

- The various input parameters considered in this work are: *tasks per application*, *number of applications*, *min/max edge weight*, *number of runs per setting*, *size of NoC mesh*, *edge ratios* and *edge percentage*.
- The measures used for evaluation of mapping algorithms are as follows.
  - *Communication Energy Cost* is the total cost of communication occurring within a mapped application in a mesh of cores.
  - *Execution time* is the time taken to map all the tasks in an application by the mapping algorithm.
- For our results we normalize the above two measures to corresponding PCF results.

- *Normalized Cost* - ratio of communication energy cost obtained by other algorithms to communication energy cost obtained by PCF.
- *Normalized Time* - ratio of execution time obtained by other algorithms to execution time obtained by PCF.

➢ The mapping algorithms are run in MATLAB environment with 64bit- Intel core I5-2400 3.10 GHz processor and 8 GB Memory.

4.1.1 Simulation Results for Randomly Generated Task Graphs

We randomly generated task graphs using ranges of various parameter values. The *edge ratios* considered here are 0.5, 2, and 20. The *edge percentages* considered here are 5%, 10%, 20%, and 50%. The *number of tasks per application* is varied from 5 to 150. The minimum *edge weight* is always 1, and the *edge wieght* on an edge is a random number between the min and max. The *number of runs per setting* is 210, which means for a given set of parameters 210 different applications are generated and the reported results are the average of the results over these 210 runs.

The simulation results for randomly generated task graphs are shown in Figures 4.3, 4.4, 4.5, and 4.6. The normalized cost and time are calculated for certain test cases and tabulated as shown. In Figure 4.3 (a-f) the cost and time are calculated for maximum edge weight of 1 (that is, every edge has weight of 1).

The results show that for normalized cost, the PCF algorithm gives greater energy savings at lower edge ratios in which the graph is less connected. The cost saving increases as the number of tasks in an application increases. From normalized time, the PCF algorithm takes longer time when compared with most other mapping algorithms.

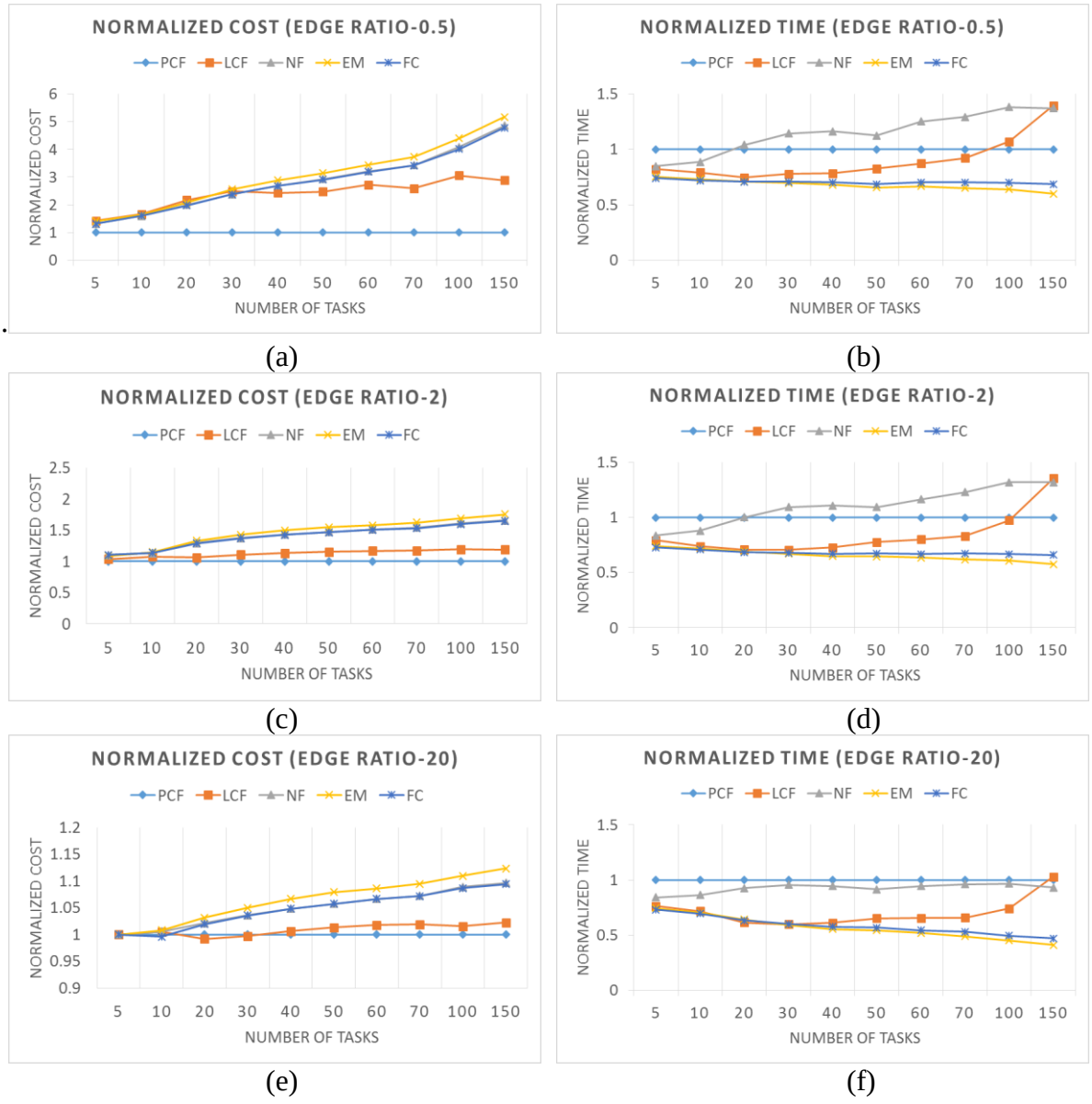


Figure 4.3: Normalized cost and time with edge weight equal to 1.

In Figure 4.4 the normalizedcost and time are calculated for maximum edge weight of 10 (that is, every edge has weight randomly choosen in range 1-10). From Figure 4.4 we can say that the PCF algorithm gives a slight energy saving compared to Figure 4.3. Relative time taken to execute the PCF algorithm slightly increases due overhead of calculating communication energy with added weight to an edge.

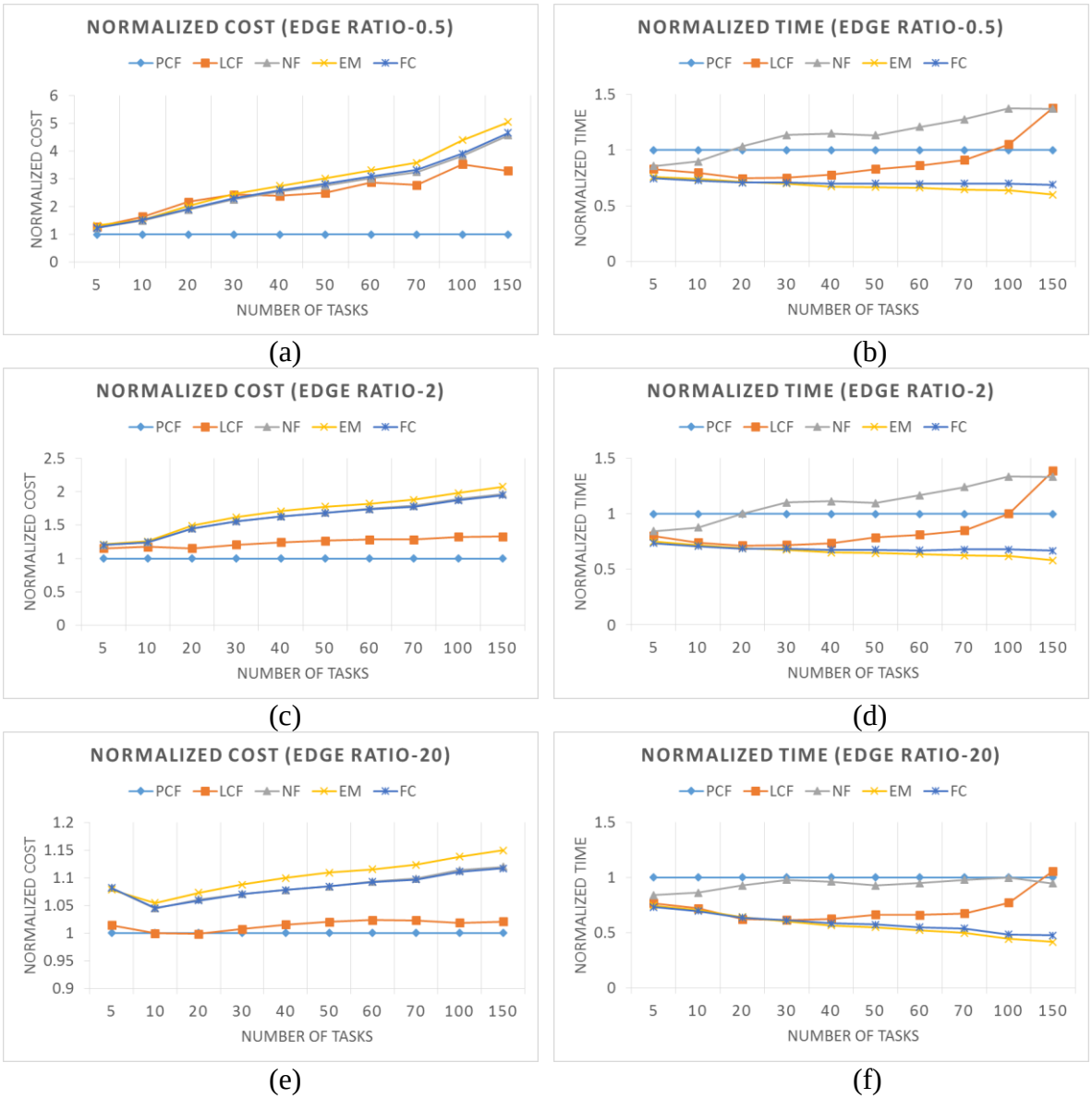


Figure 4.4: Normalized cost and time with edge weight range (1-10).

In Figure 4.5, the normalized cost and time are calculated for maximum edge weight of 100 (that is, every edge has weight randomly choosen in range 1-100). From Figure 4.5 we can say that the PCF algorithm gives better energy saving compared to Figure 4.3, because for other algorithms there is a slight up shift of lines in the graph. Relative time taken to execute the PCF algorithm remains similar to that of Figure 4.4 as there is no extra work to be performed.

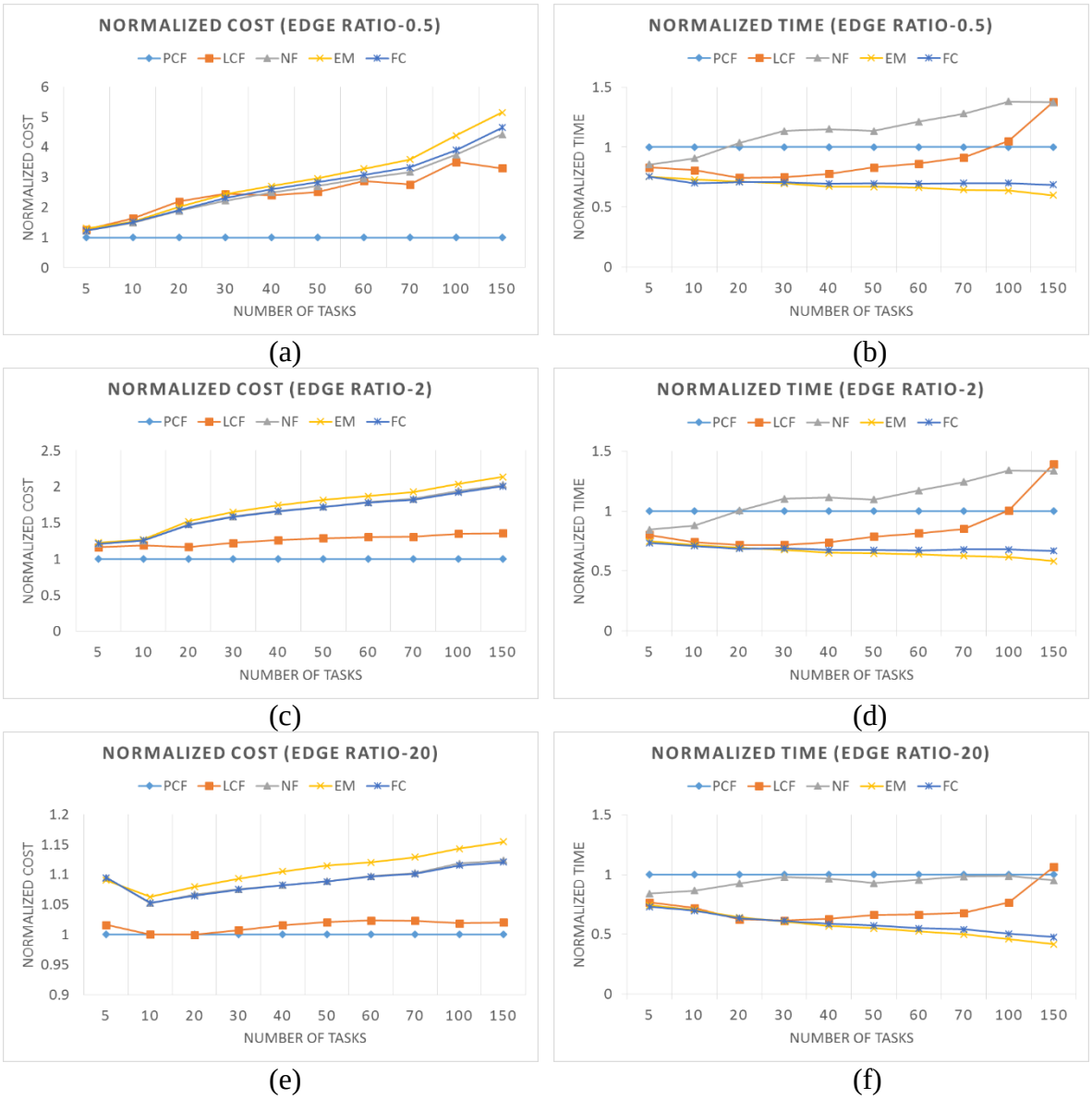
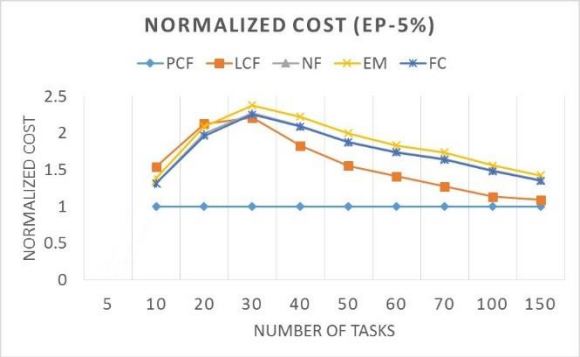


Figure 4.5: Normalized cost and time with edge weight range (1-100).

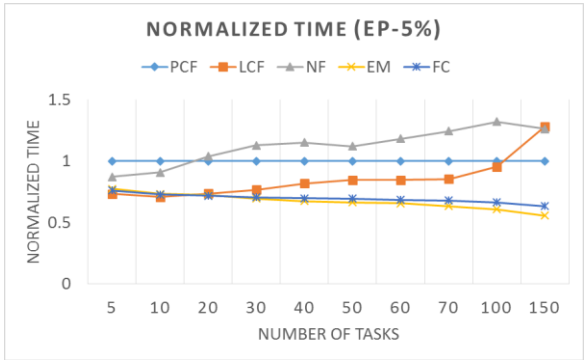
In above graphs considering *edge ratio*, notice the areas where there are fewer tasks per application. The above graphs show that there is less energy saving for applications with fewer tasks when compared to applications with more tasks. Recall that for large edge ratio, graphs with few tasks are completely connected, and so they are not as distinct as one would expect. Taking parameter *edge percentage* constructs graphs slightly differently.

The edge percentages can be used to analyze small applications. Figure 4.6 shows simulation results for various algorithms with respect to *edge percentage* and maximum *edge weight* of 1. The PCF algorithm gives greater energy saving for lower edge percentages and the energy saving decreases as the edge percentage and application size increases. Normalized time in the Figure 4.6 shows similar time varitions as Figure 4.3.

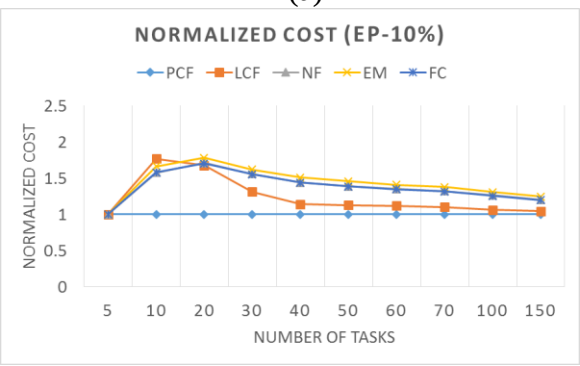
In graphs considering *edge percentage*, notice the areas where there are less tasks per application. The graphs in Figure 4.6 show that there is minimal energy saving for applications with more tasks when compared to applications with fewer tasks. Comparing Figure 4.6 with Figure 4.3 shows that the way we are representing number of edges in an application directly reflects as different results in the experiments performed. Hence we need to consider both measures *edge ratio* and *edge percentage* in better representing and analyzing application task graphs.



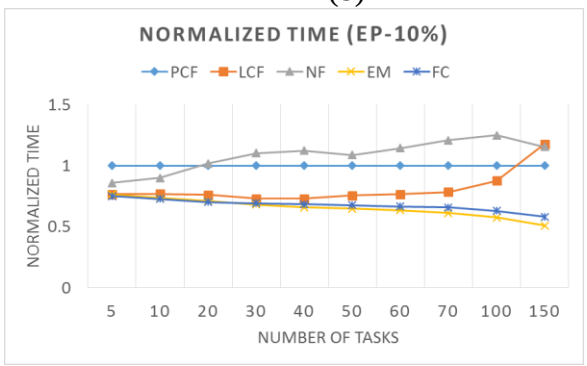
(a)



(b)



(c)



(d)

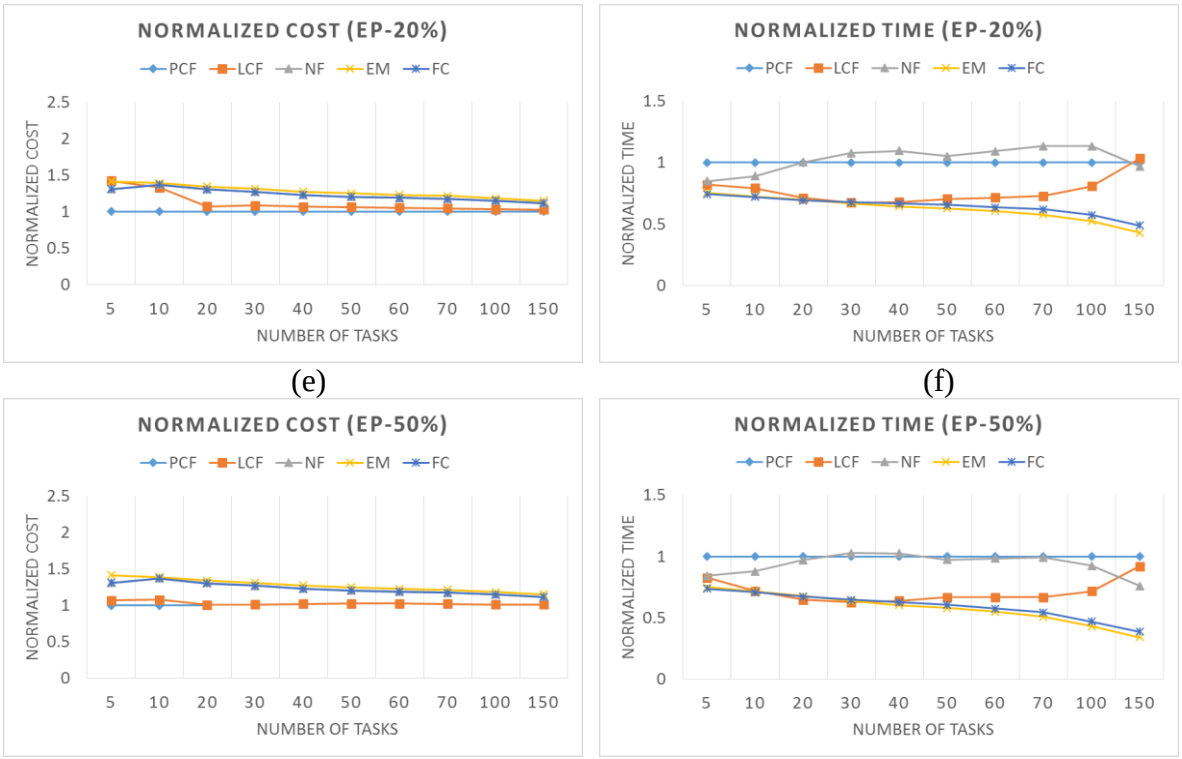


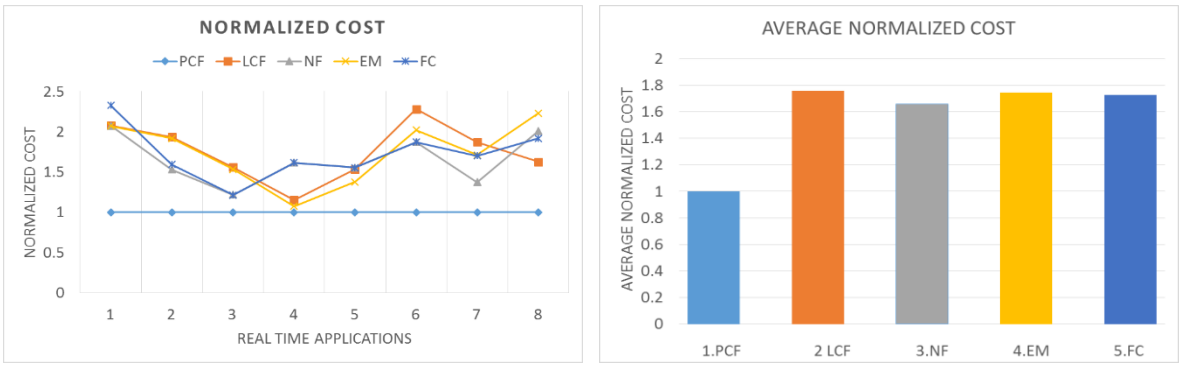
Figure 4.6: Normalized cost and time for different edge percentages.

4.1.2 Simulation Results for Real Application Task Graphs

We applied the mapping algorithms to task graphs of the application in Figure 4.1. To obtain timing accuracy, each mapping is run for more than 200 times and then the resulting execution times are averaged. Here the communication cost obtained by each mapping remains same. The resulting normalized cost and time are calculated based on the resulting outputs of mapping algorithms.

The simulation results indicate that based on normalized cost, the PCF algorithm outperforms other mapping algorithms. Figure 4.7 shows normalized cost of mapping algorithms considering real application tasks graph numbered 1 to 8 (*a* to *h* in Figure 4.1). On average, the

communication cost with other mapping algorithms is 72% more than with PCF. Figure 4.7 (b) gives average normalized cost for all the applications considered.

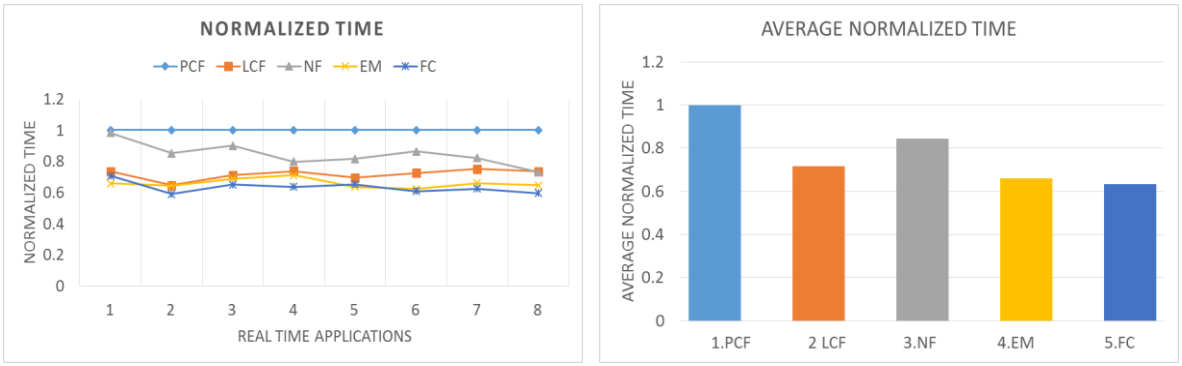


(a) (b)

Figure 4.7: Normalized cost of real application task graph.

Figure 4.8 shows the normalized time of mapping algorithms. Simulation results indicate that PCF runs much slower than other algorithms. On averge, the execution time for other algorithms is 30% faster than with PCF. Figure 4.8 (b) gives average normalized time for real application task graphs. Here Fixed Center runs fastest.

Hence, even though PCF runs slower than the other algorithms, the communication cost benefit is substantially higher. The various application task graphs considered here were decoders and encoders. PCF yields energy savings as the run time for these applications is much longer than the mapping execution time,



(a) (b)

Figure 4.8: Normalized time of real application task graph.

4.2 Simulation Results using Scheduler

The following experiments are run to test the mapping algorithms, behavior when attached with region selection. In previous sections, the normalized cost and time were measured for a fixed application size. Here we consider varying application size to simulate a real world online scenario. The scheduler tries to schedule applications as they arrive.

The simulation results below are obtained by using region selection and mapping. The initial parameters considered as follows.

- NoC size = 32 × 32. Number of applications = 300. Number of runs per experiment =210.
- Number of tasks per application = [10 - 100] (randomly selected).
- Region selection selects a region large enough to hold an application. This selection gives priority to rectangles with ratio of width and height closer to 1.
- Here we consider execution time as time taken to schedule an application.
- The experiment is run on random generated task graph with edge ratios of 0.5, 2, and 20, and also max edge weight (EW) of 100.

Figure 4.9 shows the normalized cost and time for different edge ratios and edge weights. These results indicate that a scheduler running the PCF mapping algorithm yields better energy savings than with other mapping algorithms. The normalized time shows that the execution time for other algorithms is upto 12% faster than with PCF. Figure 4.9 depicts similar results obtained in randomly generated task graphs, where the energy saving of PCF algorithm decreases as the edge ratio increases. The execution time varies slightly for the increase in edge ratios showing deviation from previous results, because here we consider execution time as time taken to schedule an application.

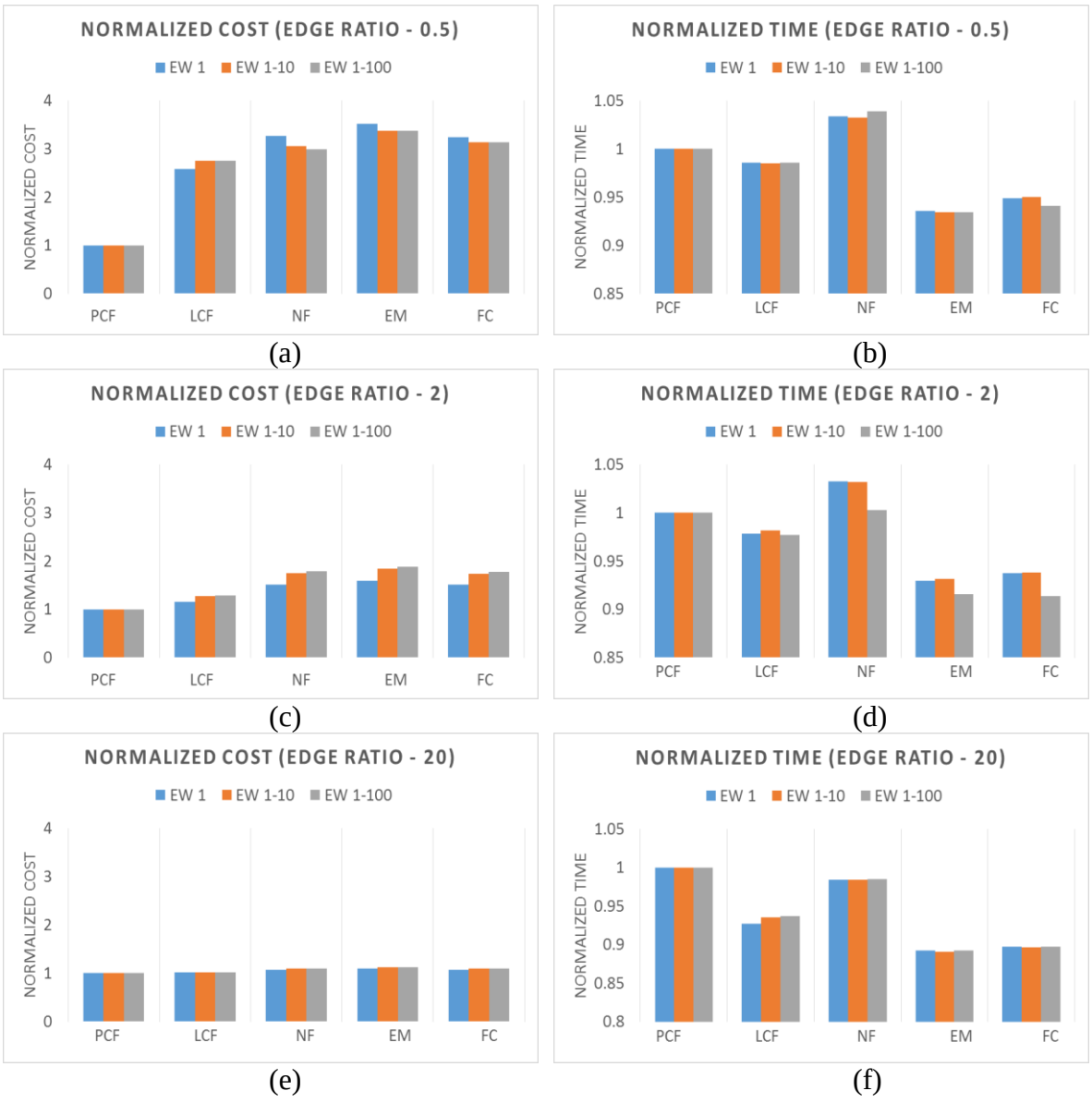


Figure 4.9: Normalized cost and time for scheduling under different edge ratios.

Chapter 5: Conclusions and Future Work

This work described several heuristic-based mapping algorithms targeting low-energy consumption. We first constructed a system model consisting of NoC target architecture, application characteristics, and NoC energy model. We focused on the placed communication first (PCF) algorithm which orders tasks for placement by amount of communication to already-placed tasks. We evaluated mapping algorithms with metrics like communication cost and execution time. Experimental results were obtained for applying mapping algorithms to randomly generated tasks graphs, real application task graphs, and using a scheduler.

The results show that PCF outperforms other mapping algorithm in most situations in terms of energy savings. On average, the communication cost with other mapping algorithms is 72% more than with PCF, and execution time for other algorithms is 30% faster than with PCF for real application task graphs.

Currently our work is based on the assumption of homogeneous many-core. In the future we plan to apply this techniques to heterogeneous many-cores. We also plan to consider the platforms allowing dynamic voltage and frequency scaling, as incorporating this technique can provide further energy savings. We also want to consider bandwidth constraints in links, for better routing of data through routers.

References

1. M. A. Bender, D. P. Bunde, E. D. Demaine, S. P. Fekete, V. J. Leung, H. Meijer, and C. A. Phillips (2008), "Communication-Aware Processor Allocation for Supercomputers," *Algorithmica*, vol. 50, no. 2, pp. 279–298.

2. E. L. de S. Carvalho, N. L. V. Calazans, and F. G. Moraes (2010), "Dynamic Task Mapping for MPSoCs," *IEEE Design & Test of Computers*, vol. 27, no. 5, pp. 26-35.

3. E. Carvalho, C. Marcon, N. Calazans, and F. Moraes (2009), "Evaluation of Static and Dynamic Task Mapping Algorithms in NoC-based MPSoCs," *Proc. Int'l. Symp. System-on-Chip* (SOC), pp. 87-90.

4. E. Carvalho and F. Moraes (2008), "Congestion-Aware Task Mapping in Heterogeneous MPSoCs," *Proc. Int'l. Symp. System-on-Chip* (SOC), 4 pp.

5. C.-L. Chou and R. Marculescu (2007), "Incremental Run-Time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels," *Proc. 5th IEEE/ACM/IFIP Int'l. Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 161-166.

6. C.-L. Chou, U. Y. Ogras, and R. Marculescu (2008), "Energy- and Performance-Aware Incremental Mapping for Networks on Chip With Multiple Voltage Levels," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1866-1879.

7. M. Elbidweihy and J. L. Trahan, (2009) "Maximal Strips Data Structure to Represent Free Space on Partially Reconfigurable FPGAs," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 4, pp. 349-366.

8. D. He and W. Mueller (2013), "A Heuristic Energy-Aware Approach for Hard Real-Time Systems on Multi-Core Platforms," *Microprocessors and Microsystems*, vol. 37, no. 8, Part A, pp. 858–870.

9. J. Lee, M.-K. Chung, Y.-G. Cho, S. Ryu, J. H. Ahn, and K. Choi (2013), "Mapping and Scheduling of Tasks and Communications on Many-Core SoC Under Local Memory Constraint," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 11, pp. 1748-1761.

10. A. Mahabadi, S. M. Zahedi, and A. Khonsari (2013), "Reliable Energy-Aware Application Mapping and Voltage–Frequency Island Partitioning for GALS-based NoC," *J. Comput. Sys. Sci.*, vol 79, no. 4, pp. 457-474.

11. M. Mandelli, A. Amory, L. Ost, and F. G. Moraes (2011), "Multi-Task Dynamic Mapping onto NoC-based MPSoCs," *Proc. 24th Symp. Integrated Circuits and Systems Design* (SBCCI '11), pp. 191-196.

12. S. Manolache, P. Eles, and Z. Peng (2005), "Fault and Energy-Aware Communication Mapping with Guaranteed Latency for Applications Implemented on NoC," *Proc. 42nd Design Automation Conf.*, pp. 266-269.

13. C. A. M. Marcon, E. I. Moreno, N. L. V. Calazans, and F. G. Moraes (2008), "Comparison of Network-on-Chip Mapping Algorithms Targeting Low Energy Consumption," *IET Computers & Digital Techniques*, vol. 2, no. 6, pp. 471-482.

14. R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote (2009), "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, pp. 3-21.

15. S. Murali and G. De Micheli (2004), "Bandwidth-Constrained Mapping of Cores onto NoC Architectures," *Proc. Conf. Design, Automation and Test in Europe*, vol. 2, 6 pages.

16. J. Niu, C. Liu, Y. Gao, and M. Qiu (2014), "Energy Efficient Task Assignment with Guaranteed Probability Satisfying Timing Constraints for Embedded Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2043-2052.

17. L. Ost, M. Mandelli, G. M. Almeida, L. Moller, L. S. Indrusiak, G. Sassatelli, P. Benoit, M. Glesner, M. Robert, and F. Moraes (2013), "Power-Aware Dynamic Mapping Heuristics for NoC-based MPSoCs Using a Unified Model-Based Approach," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, Article 75, 22 pp.

18. P. K. Sahu and S. Chattopadhyay (2013), "A Survey on Application Mapping Strategies for Network-on-Chip Design," *J. Systems Architecture*, vol. 59, no. 1, pp. 60-76.

19. R. R. Schaller (1997), "Moore's Law: Past, Present, and Future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52-59.

20. E. Seo, J. Jeong, S. Park, and J. Lee (2008), "Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors," *IEEE Trans. Par. Distrib. Sys*, vol. 19, no. 11, pp. 1540-1552.

21. W.-Y. Shieh and C.-C. Pong (2013), "Energy and Transition-Aware Runtime Task Scheduling for Multicore Processors," *J. Par. Distrib. Computing*, vol. 73, no. 9, pp. 1225-1238.

22. K. Srinivasan and K. S. Chatha (2005), "A Technique for Low Energy Mapping and Routing in Network-on-Chip Architectures," *Proc. Int'l. Symp. Low Power Electronics and Design* (ISLPED), pp. 387-392.

23. J. L. Trahan and M. Elbidweihy (2013), "Task Mapping Heuristics Descriptions," manuscript.

24. T. T. Ye, G. De Micheli, and L. Benini. 2002. "Analysis of Power Consumption on Switch Fabrics in Network Routers," In *Proceedings of the 39th annual Design Automation Conference* (DAC '02), pp. 524-529.

Appendix A: Input/Output Parameters

This section describes input/output parameters and several array structure and variables that are common to the mapping algorithms.

Inputs:

T – A weighted adjacency matrix of *n* (number of tasks in an application) tasks in descending order of communication volume plus other task information.

Size equal to $n \times (n+3)$.

Each row corresponds to a task.

Column – (1): Task IDs.

($2 - n+1$): Weighted adjacency matrix.

($n + 2$): Number of edges for a given task.

($n + 3$): Sum of edge weights (communication volume) for a given task.

r – Number of rows in selected region of mesh of cores.

c – Number of columns in selected region of mesh of cores.

($r \times c$): Represents dimension of selected mesh of cores such that $rc \geq n$.

Outputs:

Map – An $r \times c$ matrix consisting of task IDs, where $Map(p, q) = i$ indicates that core (p, q) holds tasks i after the mapping.

Variables:

Frontier – A matrix with size $r \times c$, where entry (i, j) indicates whether core (i, j) is occupied (2), free and adjacent to an occupied core (1), is free and not adjacent to an occupied core (0).

Y - An array consisting of positions of frontier cores. It gets updated for every placement of a task, with new frontiers added at the end.

distance - Stores distance between one node to another. Initially it is set to infinity.

temp_dist - Stores shortest distance temporarily.

c_row - Stores x axis value of center of the already placed cores mesh.

c_col - Stores y axis value of center of the already placed cores mesh.

t_row - Stores x axis value of a core in which a task is going to be placed.

t_col - Stores y axis value of a core in which a task is going to be placed.

Appendix B: Euclidean Minimum

EM (*T, r, c*)

initialize *Frontier* to all 0's ; array of frontier cores
Y = ∅ ; list of frontier cores
c_row = ⌈*r*/2⌉ ; row of center of mesh
; uses ceiling rather than floor for special cases *r* = 1 and *c* = 1
c_col = ⌈*c*/2⌉ ; column of center of mesh
map(1) = (*c_row*, *c_col*) ; places at the center of the mesh,
; but could choose a different starting point
Frontier(*c_row*, *c_col*) = 2
add (*c_row*, *c_col*) to *Y*
Frontier = MARKFRONTIER(*c_row*, *c_col*, *r*, *c*, *Frontier*, *Y*)
for *i* = 2 to *n*
distance = ∞
for each (*p*, *q*) ∈ *Y*
temp_dist = $\sqrt{(p - c\_row)^2 + (q - c\_col)^2}$; can use squared distance instead
; because all distances ≥ 1, so avoid taking square root
if *temp_dist* < *distance*
then
distance = *temp_dist*
t_row = *p*
t_col = *q*
map(*i*) = (*t_row*, *t_col*)
Frontier(*t_row*, *t_col*) = 2
Frontier = MARKFRONTIER(*t_row*, *t_col*, *r*, *c*, *Frontier*, *Y*)
(*c_row*, *c_col*) = CENTERMAPPED(*i*, *c_row*, *c_col*, *t_row*, *t_col*)
; find new center of mapped cores
return *map*

The function **MARKFRONTIER** updates the *Frontier* list based on recently placed task.

p - x axis value of recently placed task.

q - y axis value of recently placed task.

(r × c): Represents dimension of mesh of cores.

MARKFRONTIER (*p, q, r, c, Frontier, Y*)

; updates frontier node array and list Y after a task is mapped to (p, q)

```
remove (p, q) from Y
if p+1 ≤ r                    ; add N, S, E, W neighbors to frontier if not mapped
    if Frontier(p+1, q) == 0          ; only add to Y if not mapped and
                                      ;  not already marked as 1
        then
            Frontier(p+1, q) = 1
            add (p+1, q) to Y
if p–1 ≥ 1
    if Frontier(p–1, q) == 0
        then
            Frontier(p–1, q) = 1
            add (p–1, q) to Y
if q+1 ≤ c
    if Frontier(p, q+1) == 0
        then
            Frontier(p, q+1) = 1
            add (p, q+1) to Y
if q–1 ≥ 1
    if Frontier(p, q–1) == 0
        then
            Frontier(p, q–1) = 1
            add (p, q–1) to Y
if p+1 ≤ r and q+1 ≤ c    ; add NE, NW, SE, SW neighbors to frontier if not mapped
                          ; if not counting these as neighbors, then omit
    if Frontier(p+1, q+1) == 0        ; only add to Y if not mapped and
                                      ;  not already marked as 1
        then
            Frontier(p+1, q+1) = 1
            add (p+1, q+1) to Y
if p+1 ≤ r and q–1 ≥ 1
    if Frontier(p+1, q–1) == 0
        then
            Frontier(p+1, q–1) = 1
            add (p+1, q–1) to Y
if p–1 ≥ 1 and q+1 ≤ c
    if Frontier(p–1, q+1) == 0
        then
            Frontier(p–1, q+1) = 1
            add (p–1, q+1) to Y
if p–1 ≥ 1 and q–1 ≥ 1
    if Frontier(p–1, q–1) == 0
        then
            Frontier(p–1, q–1) = 1
            add (p–1, q–1) to Y
return Frontier
```

The function **CenterMapped** updates the *Frontier* list based on recently placed task.

j - It is the number of tasks mapped before the newest one.
old_row, old_col - Are the coordinates of the center before mapping a new task.
one_row, one_col - Are the coordinates of the newly mapped task.

CenterMapped (*j, old_row, old_col, one_row, one_col*)

; This procedure computes the new center by weighting the old center with the
; Number of cores used to construct it

$c\_row = \lceil (j \times old\_row + one\_row) / (j+1) \rceil$; update row of center of mapped cores
$c\_col = \lceil (j \times old\_col + one\_col) / (j+1) \rceil$; update column of center of mapped cores
; uses ceiling rather than floor for special cases $r = 1$ and $c = 1$
return *c_row, c_col*

Appendix C: Fixed Center

The implementation is similar to Euclidean Minimum except the formula for calculating distance between each neighbors is Manhattan Distance and the center is the fixed center of the region.

Changes to **EM**(*T*, *r*, *c*)

1. $temp\_dist = |p - c\_row| + |q - c\_col|$
2. Delete the call to **CenterMapped**.

Appendix D: Neighbor-Aware Frontier

Variables:

temp_neighb – A variable which stores number of free neighbors for a core

neighbors – A variable which stores least number of free neighbors. Initially it is set to 5 because, the number of neighbors is 4 in N, S, E, W direction.

NF (*T*, *r*, *c*)

```
initialize Frontier to all 0's     ; array of frontier cores
Y = ∅                              ; list of frontier cores; can also create without Y as in EM1
c_row = ⌈r/2⌉  ; row of center of mesh
c_col = ⌈c/2⌉  ; column of center of mesh
map(1) = (c_row, c_col)        ; places at the center of the mesh,
                                   ; But could choose a different starting point
Frontier(c_row, c_col) = 2
add (c_row, c_col) to Y
Frontier = MARKFRONTIER(c_row, c_col, r, c, Frontier, Y)
for i = 2 to n
    neighbors = 5  ; 5 is initialized because, there can only 4 be neighbors in N, S, W,
                       E direction
    for each (p, q) ∈ Y
        temp_neighb = COUNTNEIGHBORS(p, q, r, c, Frontier)
        if temp_neighb < neighbors
            then
                neighbors = temp_neighb
                t_row = p
                t_col = q
    map(i) = (t_row, t_col)
    Frontier(t_row, t_col) = 2
    Frontier = MARKFRONTIER(t_row, t_col, r, c, Frontier, Y)
return map
```

The function **CountNeighbors** updates *count* based on *Frontier* list.

p - x axis value of recently placed task.

q - y axis value of recently placed task.

($r \times c$): Represents dimension of mesh of cores.

CountNeighbors (*p, q, r, c, Frontier*)

; returns the number of unmapped neighbors of (p, q) in N, S, E, W directions

count = 0
if $p+1 \le r$; or "if $p < r$"
    if *Frontier*($p+1$, q) $\neq 2$
        then *count* = *count* + 1
if $p-1 \ge 1$; or "if $p \ge 2$" or "if $p > 1$"
    if *Frontier*($p-1$, q) $\neq 2$
        then *count* = *count* + 1
if $q+1 \le c$; or "if $q < c$"
    if *Frontier*(p, $q+1$) $\neq 2$
        then *count* = *count* + 1
if $q-1 \ge 1$; or "if $q \ge 2$" or "if $q > 1$"
    if *Frontier*(p, $q-1$) $\neq 2$
        then *count* = *count* + 1
return *count*

Appendix E: Largest Communication First

Variables:

*C*2 - An array which contains set of cores in $r \times c$ mesh with 2 links
*C*3 - An array which contains set of cores in $r \times c$ mesh with 3 links
*C*4 - An array which contains set of cores in $r \times c$ mesh with 4 links

*availableC*2 - An array which contains list of available C2.
*availableC*3 - An array which contains list of available C3.
*availableC*4 - An array which contains list of available C4.

*W*2 - An array which contains list of tasks with ≤2 links waiting for assignment.
*W*3 - An array which contains list of tasks with 3 links waiting for assignment.
*W*4 - An array which contains list of tasks with 4+ links waiting for assignment.

*A*2 - An array which contains list of tasks assigned to cores in *C*2.
*A*3 - An array which contains list of tasks assigned to cores in *C*3.
*A*4 - An array which contains list of tasks assigned to cores in *C*4.

U - An array which contains array of tasks in *T* sorted by communication volume.
R - An array which contains set of already mapped tasks.

LCF (*T*, *r*, *c*)

; 1st step
*C*2 = set of cores in $r \times c$ mesh with 2 links
*C*3 = set of cores in $r \times c$ mesh with 3 links
*C*4 = set of cores in $r \times c$ mesh with 4 links
*availableC*2 = size of *C*2
*availableC*3 = size of *C*3
*availableC*4 = size of *C*4

; 2nd step
*W*2 = ∅ ; list of tasks with ≤2 links waiting for assignment
*W*3 = ∅ ; list of tasks with 3 links waiting for assignment
*W*4 = ∅ ; list of tasks with 4+ links waiting for assignment
*A*2 = ∅ ; list of tasks assigned to cores in *C*2
*A*3 = ∅ ; list of tasks assigned to cores in *C*3
*A*4 = ∅ ; list of tasks assigned to cores in *C*4
U = array of tasks in *G* sorted by communication volume
R = ∅ ; set of already mapped tasks

```
for i = 1 to n
    if |Adj(u_i)| ≤ 2
        then
            if availableC2 > 0
                then
                    A2 = A2 ∪ {u_i}
                    availableC2 = availableC2 – 1
                else W2 = W2 ∪ {u_i}
    if |Adj(u_i)| = 3
        then
            if availableC3 > 0
                then
                    A3 = A3 ∪ {u_i}
                    availableC3 = availableC3 – 1
                else W3 = W3 ∪ {u_i}
    if |Adj(u_i)| ≥ 4
        then
            if availableC4 > 0
                then
                    A4 = A4 ∪ {u_i}
                    availableC4 = availableC4 – 1
                else W4 = W4 ∪ {u_i}


        ; 3rd step
For i = 1 to size of (A4)
    t = A4(i)
    ⟨map(t), C4, R⟩ = AssignTaskLCF(t, r, c, C4, R)
For i = 1 to size of (A3)
    t =A3(i)
    ⟨map(t), C3, R⟩ = AssignTaskLCF(t, r, c, C3, R)
For i = 1 to size of (A2)
    t = A2(i)
    ⟨map(t), C2, R⟩ = AssignTaskLCF(t, r, c, C2, R)
For i = 1 to size of (W4)
    t = W4(i)
    if |C4| > 0
        then ⟨map(t), C4, R⟩ = AssignTaskLCF(t, r, c, C4, R)
    else if |C3| > 0
            then ⟨map(t), C3, R⟩ = AssignTaskLCF(t, r, c, C3, R)
            else ⟨map(t), C2, R⟩ = AssignTaskLCF(t, r, c, C2, R)
For i = 1 to size of (W3)
    t = W3(i)

    if |C3| > 0
```

```
            then ⟨map(t), C3, R⟩ = AssignTaskLCF(t, r, c, C3, R)
        else if |C4| > 0
                then ⟨map(t), C4, R⟩ = AssignTaskLCF(t, r, c, C4, R)
                else ⟨map(t), C2, R⟩ = AssignTaskLCF(t, r, c, C2, R)

    For i = 1 to size of (A2)
        t = W2(i)

        if |C2| > 0
            then ⟨map(t), C2, R⟩ = AssignTaskLCF(t, r, c, C2, R)
        else if |C3| > 0
                then ⟨map(t), C3, R⟩ = AssignTaskLCF(t, r, c, C3, R)
                else ⟨map(t), C4, R⟩ = AssignTaskLCF(t, r, c, C4, R)
                    ; assumes that enough cores exist to map all tasks
    return map
```

The function **ASSIGNTASKLCF** places a task on to the free core based on C (2, 3, 4) list.

numCommun - A count of already placed tasks, with which new task communicating.

AssignTaskLCF (*t, r, c, H, R*)

```
; c – Weighted adjacency matrix.
; Maps task t to a core in set H in an r×c submesh, using the three situations described
        by Marcon et al.
; Returns index of cores to which t is mapped, updated H with that core removed, and
        updated R with that task added

numCommun = 0       ; count of already placed tasks with which t communicates
for each u ∈ R
    if (t, u) ∈ E     ; where E is the edge set of task graph G
        then
            numCommun = numCommun + 1
            communTask = u   ; identity of one placed task that communicates with t
if numCommun = 0
    then        ; map to core in H closest to center of mesh
        c_row = ⌈r/2⌉  ; row of center of mesh
        c_col = ⌈c/2⌉  ; column of center of mesh
        distance = ∞
        for each (p, q) ∈ H
            temp_dist = | p – c_row | + | q – c_col |
            if temp_dist < distance
                then
                    distance = temp_dist
                    t_row = p
                    t_col = q
```

```
        else if numCommun = 1
            then    ; map to core in H closest to the one communicating task;
                    ;   task communTask is that task
                distance = ∞
                for each (p, q) ∈ H
                        temp_dist = | p – map(communTask)_row |
                                + | q – map(communTask)_col |
                    ; let map(communTask)_row denote the row to which task
                    ;   communTask is mapped; similarly for map(communTask)_col
                    if temp_dist < distance
                        then
                            distance = temp_dist
                            t_row = p
                            t_col = q
            else        ; numCommun ≥ 2
                        ; evaluate all free cores, and assign to the location
                        ; with lowest communication cost
                        ; here, lowest communication cost translates to
                        ; lowest sum of Weighted Manhattan distances to already placed
                        ; tasks
                sumDistance = ∞
                for each (p, q) ∈ H
                    temp_energy_cost = 0
                    for each u ∈ R
                        temp_weight = c (weight of edge (t, u))
                        if (t, u) ∈ E       ; where E is the edge set of task graph G
                            then
                            temp_energy_cost = temp_energy_cost + ((| p – map(u)_row | +
                                                | q – map(u)_col |) * temp_weight)
                    if temp_energy_cost < sumDistance
                        then
                            sumDistance = temp_energy_cost
                            t_row = p
                            t_col = q
    map(t) = (t_row, t_col)
    H = H – {(t_row, t_col)}
    R = R ∪ {t}

    return ⟨map(t), H, R⟩
```

Appendix F: Placed Communication First

Variables:

communplaced - An array which contains set of all tasks. Value '-1' represents that task has been mapped to a core, value '0' represents a task not communicating with any placed task, and any positive value represents the communication volume of a task with already placed tasks.

U - An array, which contains tasks ID's which are not yet placed to core.

large - A variable which next best task to place. It is selected based on communication between placed tasks.

commun_adjlist - A matrix which contains adjacency list of input tasks and its edges. The first row of *commun_adjlist* consists of number of edges for each respective task. Size is $n \times (n+1)$.

task_id - A variable which contains index of task in array *T*.

mostcommun_vol – A variable which stores largest communicating task in *communplaced.*

large_degree - A variable which stores number of edges of a task *large.*

large_index_u - A variable which stores index of *large* in array *U.*

*temp_dist*_var - A variable which stores distance between two tasks.

temp_cost - A Variable which stores cost of task *large* with already placed tasks.

PCF (*T*, *r*, *c*)

initialize *Frontier* to all 0's ; array of frontier cores
Y = ∅ ; list of frontier cores
U = all tasks in *T* ; set of unmapped tasks
Initialize *communplaced* as an array of 0's with size(no of tasks) ; Initializing array for tasks which are already placed.

c_row = ⌈*r*/2⌉ ; row of center of mesh
c_col = ⌈*c*/2⌉ ; column of center of mesh
large = index of task in *U* with maximum communication
U = *U* – {*large*} ; delete the task which is placed
map(*large*) = (*c_row*, *c_col*) ; places at the center of the mesh,
; but could choose a different starting point
Frontier(*c_row*, *c_col*) = 2
add (*c_row*, *c_col*) to *Y*
Frontier = MARKFRONTIER(*c_row*, *c_col*, *r*, *c*, *Frontier*, *Y*)

commun_adjlist = array of lists with size (no of tasks); adjacency list of input task and its edges.

; Initialize *commun_adjlist* with its task sets.
; The first row of *commun_adjlist* consists of number of edges for each respective task.
temp_count = 2; variable to count
for *i* = 1 to *n*
task_id = *T* (*i*, 1)
commun_adjlist (*task_id*, 1) = *T*(*i*, *n*+*2*) *;* total no of edges for that task_id.
for *j* = 1 to *n*
if (task is communicating with other task t_j)
commun_adjlist(*task_id*, *temp_count*) = *j* ;place the communicating task in
temp_count = *temp_count*+1 ;the list.
end
end

; load "*communplaced*" with volume of communication between each
; task and the first placed task, with ID "large"

for *i* = 1 to *T*(1, *n*+2) ; get the no of edges for first task having largest communication
temp_ID= *commun_adjlist*(*large*, *i*+1) ; task ID's with which large is
; communicating
communplaced(*temp_ID*) = *T*(1, *temp_ID* + 1) ; initialize array of communication
; volume to already placed tasks
end

communplaced(*large*) = –1 ; mark task large as already placed

; Map task after first.

for *x* = 2 to *n*
; *x* just serves as a counter for this loop, not an index
mostcommun_vol = 0 ; initialization
large= *U*(1,1) ;ID of task in *U* with maximum communication for initialization.

```
for j = 1 to n
    if communplaced(j) > mostcommun_vol
        then
            mostcommun_vol = communplaced(j) ; get the
            large =j     ; identify task with largest communication volume
                         ;   to already placed tasks
    end
end
large_degree= commun_adjlist(large, 1) ; degree of communication, of large to
                                       ; its neighbors
large_index_u = 0;

for j = 1 to size (U, 1)
    if U(j, 1) == large
        then
            large_index_u = j   ; find the index of large in list of U
    end
end

if large_degree == 0
    then
                ; if no communication to placed tasks, then place at minimum
                ; Manhattan distance to fixed center
        distance = ∞
        for each (p, q) ∈ Y
            temp_dist = | p – c_row | + | q – c_col |        ; Manhattan distance
            if temp_dist < distance
                then
                    distance = temp_dist
                    t_row = p
                    t_col = q
    else
        ; if has communication to already placed tasks
        ; finds overall communication cost to each
        ; communicating placed task wrt all available free core,
        ; then place task at minimum cost to all placed tasks
        ; (Manhattan distance * weight of the edge) to map(Placed tasks)

        cost = ∞
        for each (p, q) ∈ Y
            temp_cost=0;
            for k= 1 to large_degree
                Placed_ID= commun_adjlist(large, k+1)
                if (communplaced(Placed_ID, 1) == -1) then
                temp_dist_var = | p – map(Placed_ID)_row | + | q –
                map(Placed_ID)_col |
```

```
                    temp_cost = temp_cost + (temp_dist_var * T(large,
                    Placed_ID+1))
                    ; let map(Placed_ID)_row denote the row to which
                    ;   task Placed_ID is mapped; similarly for map(Placed_ID)_col
                    end
                end

                if temp_cost < cost
                    then
                        cost = temp_cost
                        t_row = p
                        t_col = q
                end
            end  ;( for loop)
        end ; (if else loop)

        map(large) = (t_row, t_col)
        Frontier(t_row, t_col) = 2
        Frontier = MarkFrontier(t_row, t_col, r, c, Frontier, Y)
        communplaced(large) = –1                ; mark task large as already placed

        for j = 1 to commun_adjlist (large, 1)
                ; update communication volume of unplaced tasks to placed tasks by
                ;   adding the volume to the most newly placed task
            temp_ID= commun_adjlist (large, j+1)
            if communplaced(temp_ID) > –1       ; that is, if not already placed
                communplaced(temp_ID) = communplaced(temp_ID)
                                                + T(large, temp_ID+1)
            end
        end

    U(large_index_u, 1)= [ ]        ; empty large, from U array.

    return map
```