DESIGN AND PERFORMANCE EVALUATION OF MIGRATION-BASED SUBMESH ALLOCATION STRATEGIES IN MESH MULTICOMPUTERS

GOH LEE KEE

(B.Eng.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

I would like to thank my supervisor, Dr. Bharadwaj Veeravalli, for all the advices, suggestions and recommendations he has given me during the course of my research. If not for his guidance, I may not be able to complete my research within such a short period of time.

I would also like to express my appreciation to Mr. David Koh for helping me whenever I have any problems regarding the computer and equipment that I need for my research.

Last but not least, my thanks goes out to all my friends at the Open Source Software Laboratory (OSSL) who have made my stay here a pleasant and fruitful one.

Contents

Summary

In this thesis, the problem of processor allocation on mesh-based multicomputer systems is considered. The thesis first discusses the problem of fragmentation that arises as a result of using contiguous processor allocation strategies. Next, a comprehensive survey of various existing contiguous allocation strategies found in the literature is made.

In order to minimize the overall processing time of the tasks and the fragmentation caused by contiguous processor allocation strategies, this thesis employs the idea of fusing migration as a part of allocation to design efficient task allocation algorithms. In the proposed schemes, task migration is used whenever required to improve the problem of fragmentation. To this end, three efficient schemes are proposed to improve the performance of first-fit allocation strategies commonly used in practice. The first scheme, called the First-fit mesh bifurcation (FFMB) scheme, attempts to start the search for a free submesh from either the bottom-left corner or the top-left corner of the mesh so as to reduce the amount of fragmentation in the mesh. The next two schemes, called the Online dynamic compaction - Single corner (ODC-SC) and Online dynamic compaction - Four corners (ODC-FC) schemes, use task migration to improve the performance of existing submesh allocation strategies.

Rigorous simulation experiments are performed based on practical workloads as reported in the literature to quantify all the proposed schemes and compare them against standard schemes existing in the literature. Based on the results, clear recommendations are made on the choice of the strategies and their performances are demonstrated on several influencing parameters. Several illustrative examples are also provided for the ease of understanding.

List of Tables

List of Figures

Chapter 1

Introduction

A multicomputer is a computer which consists of nodes (or processors) that execute several parallel tasks (or jobs) simultaneously. Most multicomputers are disjoint-memory machines, constructed by joining nodes via network links. Each node usually contains a microprocessor and its own private memory, and it cannot directly access other nodes' memories. A host computer that is connected to the multicomputer is usually responsible for locating free nodes to be allocated to incoming tasks and releasing them when the tasks complete their execution. Thus the problem of allocating a group of nodes for a request is referred to as *processor allocation problem.*

There are various topologies that are used in the implementation of multicomputers. Examples are the mesh, the hypercube and the torus. The mesh is one of the most popular topology due to its simplicity, regularity and scalability. It has shown a high potential as a supercomputer at a much lower cost for the parallel execution of various algorithms such as image processing, matrix multiplication and partial differentiation. Many prototype and commercial systems, such as the Tera Computer System [18], Intel Touchstone Delta [14], Intel Paragon XP/S [10]and J-Machine [11], have been built based on the mesh topology.

1.1 Types of Processor Allocation Strategies

There are different processor allocation strategies that can be used in mesh multicomputers to allocate free nodes to the incoming tasks. These strategies can be classified into two main types: *contiguous allocation* and *non-contiguous allocation.* In contiguous allocation [3, 8, 9, 12, 15, 16], the nodes that are allocated to each of the incoming task must be physically adjacent. In addition, most systems that use contiguous allocation also require the allocated nodes to form a subgraph of the original topology. For example, in mesh multicomputers, the tasks are usually allocated to submeshes. On the other hand, in non-contiguous allocation [5], the allocated nodes need not be physically adjacent to one another. Figure 1.1 shows an example of contiguous and non-contiguous allocation in a 6×6 mesh multicomputer. Two tasks, one requiring 15 nodes and the other requiring 12 nodes, need to be allocated. If contiguous allocation is used, the tasks may be allocated as shown in Figure 1.1(a). If non-contiguous allocation is used, the mesh may look

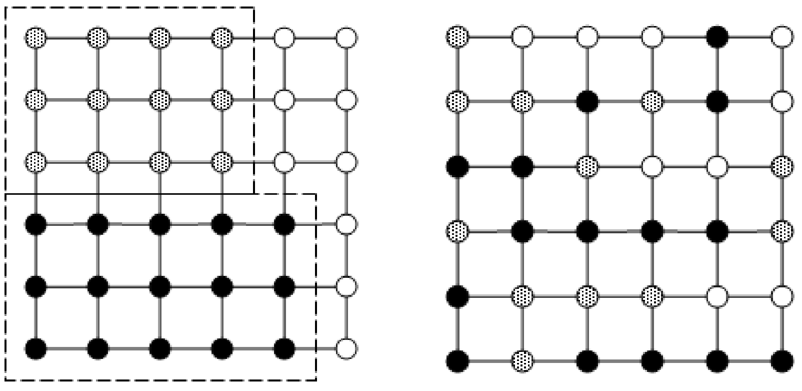

(a) Contiguous Allocation (b) Non-contiguous Allocation

Figure 1.1: Example of contiguous and non-contiguous allocation

like the one shown in Figure 1.1(b).

Contiguous allocation suffers from the problem of fragmentation due to its constraint of contiguity. Often, when a task requests for a submesh for allocation, there may be sufficient available nodes to satisfy the request but because these nodes do not form a contiguous submesh of the required size, they cannot be allocated to the task. Although non-contiguous allocation does not have this problem, it introduces communication interference since the same network link may be shared by different tasks for communication. This introduces delay and uncertainty to the execution time of the tasks. This thesis will focus on contiguous allocation only.

There has been many literature that propose different submesh allocation strategies for contiguous allocation. These strategies can be grouped into two different

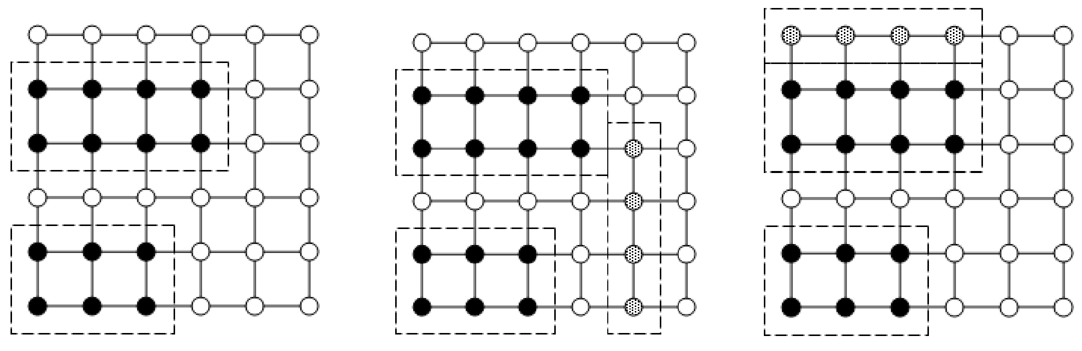

(a) Before allocation (b) First-fit allocation (c) Best-fit allocation

Figure 1.2: Example of first-fit and best-fit allocation

categories: *first-fit* and *best-fit.* In first-fit strategies [3, 8, 12, 15, 16], the mesh is searched in a particular order and the first free submesh that is found is allocated. The best-fit strategies [3, 9, 12] try to find a free submesh that results in the least amount of fragmentation in the mesh. For example, Figure 1.2(a) shows the allocated nodes in a 6×6 mesh before allocating the next task which requires a 1×4 submesh. If a first-fit strategy is used, it allocates the first free submesh that it finds, as shown in Figure 1.2(b). If a best-fit strategy is used, it will find all the available free submeshes in the mesh and allocates the free submesh that results in the least fragmentation. In this case, the submesh at the top-left corner may be allocated, as shown in Figure 1.2(c).

The first-fit strategies are fast, but they tend to cause a significant amount of fragmentation due to their first-fit nature. On the other hand, the best-fit strategies perform better in terms of reducing fragmentation, but they tend to incur higher overheads since they usually need to locate all possible free submeshes and then decide which free submesh is the best candidate.

1.2 Improving Performance in First-fit Strategies

In this thesis, three different strategies to improve the first-fit strategies proposed in the literature are designed. Firstly, a scheme to reduce the amount of fragmentation introduced by first-fit strategies is designed by starting its search from one of two corners instead of just one as seen by all the first-fit strategies. The advantage of this method is that there is very little additional overhead incurred. Next, two efficient online task allocation schemes fused with migration strategies are proposed to improve the performance of current submesh allocation strategies. One of the attractive aspects of the schemes is that both the schemes do not require all the tasks to be suspended when task migration is performed. Instead, only the task that is being migrated is suspended while the other tasks continue their computation until it is their turn to be migrated. In addition, the migration is contention-free as all the links that are used to migrate the tasks are free. The first scheme tries to move the tasks towards the bottom-left corner of the mesh so as to create a larger contiguous area of free nodes on the upper and right side of the mesh. The second scheme tries to move the tasks towards all the four corners of the mesh to create a contiguous area of free nodes in the middle of the mesh.

It is beyond the scope of this thesis to consider all the best-fit strategies. However, the two task migration schemes are extended to one of the most commonly used best-fit strategies (busy-list strategy to be described later) and the performance improvement will be shown.

The idea of fusing migration in realizing efficient task allocation algorithms is novel to the literature. Thus, as an efficient solution to the task allocation problem, task migration is used whenever required, depending on the decision taken in the proposed schemes. This is in view of improving the fragmentation problem (as explained in detail in Chapter 3 for all the strategies) and also maximizing the throughput (number of tasks that can be successfully processed by the system) of the mesh system.

The rest of the thesis is organized as follows. In Chapter 2, definitions and notations will be introduced. These notations will be used throughout the thesis. The system model and timing components are also described in this chapter. Related work on submesh allocation strategies will be presented in Chapter 3. In Chapter 4, a thorough description of the proposed schemes will be presented. Simulation results will be shown and discussed in Chapter 5. Finally, we will conclude the thesis in Chapter 6.

Chapter 2

Preliminaries

In this chapter, some notations and definitions of the terminology used in the thesis will be presented. The system model as well as the timing components of processor allocation will also be described.

2.1 Notations and Definitions

A two-dimensional mesh system can be represented as $M(W, H)$, comprising $W \times H$ nodes (or processors). A node in column i and row j is denoted by $\langle i, j \rangle$ where $0 \leq i < W$ and $0 \leq j < H$, with the bottom-left corner node having the address $\langle 0, 0 \rangle$. A submesh, denoted by $S(w, h)$, is a $w \times h$ rectangular submesh comprising $w \times h$ nodes such that $0 < w \leq W$ and $0 < h \leq H$. The address of a submesh can be denoted by $\langle x_1, y_1, x_2, y_2 \rangle$ where $\langle x_1, y_1 \rangle$ is the bottom-left corner node of the

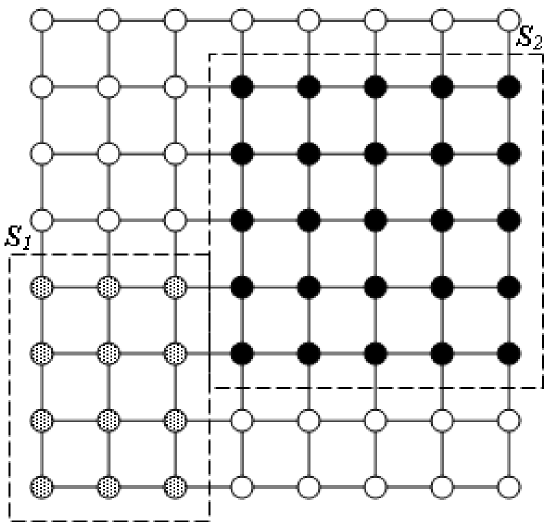


Figure 2.1: Example of an 8×8 mesh

submesh while $\langle x_2, y_2 \rangle$ is the top-right corner node of the submesh. The bottom-left corner node is usually referred to as the *base node* of the submesh $S(w, h)$. Figure 2.1 shows an 8×8 mesh denoted by $M(8, 8)$. The submesh at the bottom-left corner of the mesh is denoted by $S_1(3, 4)$ and has the address $\langle 0, 0, 2, 3 \rangle$ while the second submesh is denoted by $S_2(5, 5)$ and has the address $\langle 3, 2, 7, 6 \rangle$. The base node of $S_1(3, 4)$ is at $\langle 0, 0 \rangle$ while the base node of $S_2(5, 5)$ is at $\langle 3, 2 \rangle$.

Below are the definitions of some of the terminology and notations that will be used throughout the thesis.

Definition 1. A node is said to be *free* if it is not allocated to any task. A submesh is called a free submesh if all the nodes in the submesh are free. An allocated submesh is one in which all the nodes are allocated to the same task.

Definition 2. *Internal fragmentation* is said to occur if more nodes are allocated to a task than required.

Definition 3. *External fragmentation* is said to occur if there are sufficient number of free nodes to satisfy a request for a submesh, however, these free nodes do not form a free submesh of the required dimension.

Definition 4. *Virtual fragmentation* is said to occur if an allocation strategy fails to recognize a free submesh even though a free submesh exists.

Definition 5. The *boundary value* of a free node is the sum of the number of allocated neighbors and mesh boundary points on which a particular free node lies. The boundary value of a free submesh is the sum of the boundary values of all the nodes in the periphery of the free submesh.

Definition 6. The *residence time* or *processing time*, denoted as $t_{process}$, is the total time that a task resides in the mesh system for processing. This is the time taken for the task to finish its computation after it has been allocated to a submesh.

Definition 7. The *delay time*, denoted as t_{delay}, is the time interval between the instant when a task arrives and when it is allocated. This includes the waiting time as well as the *search time* (time required by a particular allocation strategy to locate a free submesh to be allocated to the task).

Definition 8. The *response time*, denoted as $t_{response}$, is the time interval from the instant when a task arrives to the system until it is deallocated after processing.

2.2 System Model and Timing Components

As done in the literature [3, 8, 9, 12, 15, 16], without loss of generality, it is assumed that all the requests arrive to the mesh system at node $\langle 0, 0 \rangle$ following some particular distributions. Each task that is submitted for processing is defined as a tuple comprising of the amount of CPU time required for processing. This CPU time is defined in terms of the number of nodes to be used and the residence time. Besides specifying the number of CPUs needed, it exactly specifies the dimension of the required submesh as $w \times h$. Further, it is assumed that a task submitted to the system can be executed by any node in the system.

Next, several timing components that are involved in servicing a request will be described. In addition, the timing components that are variable in length and influence the overall processing time of a task will also be identified. The process of scheduling a task involves searching for a submesh of the required dimension, allocating the nodes of that submesh to the task, and after the residence time, deallocating the allocated submesh (releasing the nodes of the submesh and declaring them as free nodes).

When a request for a submesh arrives at the mesh system, it may or may not be serviced immediately as the system may be busy serving previous requests. The request may therefore need to wait for a period of t_{wait} time units before it is being serviced. The system then consumes t_{search} time units (a variable component) to locate a free submesh. If a free submesh is found, then the system takes another $t_{maintainAlloc}$ time units to allocate this submesh to this task. It should be noted that this time also includes the time for updating any records used in the book-keeping activities such as updating and maintaining all the structures (e.g. R-Array, busy array etc). Therefore the total delay time for a request that succeeds in finding a free submesh in the first attempt is given by:

$$t_{delay} = t_{wait} + t_{search} + t_{maintainAlloc} \tag{2.1}$$

However, if the mesh is fragmented or nearly fully occupied, the system may not be able to find a free submesh on the first attempt. Once this happens, the request will be put into a queue for reallocation in the future. The requests in the reallocation queue will have to wait for t_{queue} time units (variable component) for the next submesh to be deallocated from the mesh. Each time a submesh is deallocated, the system will go through the reallocation queue and attempt to find a free submesh for each of the requests. If a request can now be satisfied, it is removed from the reallocation queue. Otherwise the request remains in the reallocation queue and waits for the next deallocation event. Suppose a request requires C attempts before a free submesh is found. The total delay time is therefore given by:

$$t_{delay} = t_{wait} + \sum_{i=1}^{C} t^{i}_{search} + \sum_{j=1}^{C-1} t^{j}_{queue} + t_{maintainAlloc} \tag{2.2}$$

where t^{i}_{search} is the time taken to search the mesh during the i-th attempt and t^{j}_{queue} is the amount of time that the request spends in the reallocation queue between the j-th and $(j+1)$-th attempt after it fails to locate a free submesh during the j-th attempt.

When a request is satisfied, the free submesh that is found will be allocated to the task and the submesh will reside in the mesh for the duration of the processing of the task $t_{process}$. After the task finishes its computation, the submesh will then be released and the system takes another $t_{maintainDealloc}$ time units to update and maintain all the structures. The total response time is therefore given by:

$$t_{response} = t_{delay} + t_{process} + t_{maintianDealloc} \tag{2.3}$$

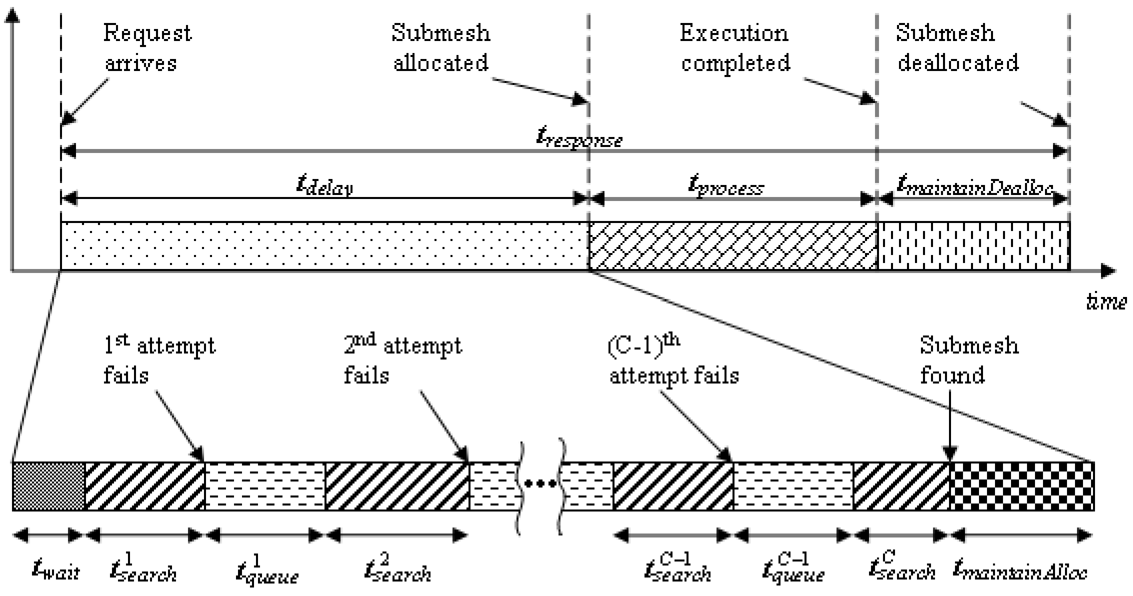


Figure 2.2: Timing components during the servicing of a request

where t_{delay} is given by (2.2). Figure 2.2 shows the timeline from the instant a request arrives at the mesh system to the instant it is removed from the system.

Chapter 3

Existing Strategies

In this chapter, some of the most recent and commonly used allocation and migration strategies and their workings will be described in a brief style for the purpose of continuity. For a more detailed analysis of these strategies, the reader may refer to their respective references.

3.1 Task Allocation Strategies

3.1.1 First-fit Strategies

This section describes a class of processor allocation strategies, referred to as first-fit strategies, as mentioned in Chapter 1.

A. **Two-Dimensional Buddy**: The two-dimensional buddy scheme was proposed by Li and Cheng [16] based on the traditional buddy strategy [19]. This strategy can only be used for a square mesh of side $W \times W$ where $W = 2^n$ and n is a positive integer. In this scheme, $(n + 1)$ free submesh lists are maintained, with the k-th list containing the free submeshes of size $2^k \times 2^k$, where $0 \leq k \leq n$. When there is a request for a submesh of size $2^i \times 2^i$, the scheme will look for a free submesh in the i-th list. If no free submesh is found in this list, the scheme will proceed to locate a free submesh in the $(i + 1)$-th list. If a free submesh is found, it will be divided into four $2^i \times 2^i$ submeshes. One of these submeshes will be allocated to the task and the rest will be inserted into the i-th list. During deallocation, the released submesh will be inserted into the appropriate list and if its three buddy submeshes are also in the list, they will be combined together to form a larger submesh and inserted into the appropriate list.

The two-dimensional buddy scheme has its limitations. It can only allocate square submeshes whose sides can be expressed in powers of 2. As a result, internal fragmentation occurs as usually more nodes are allocated to a particular task than required.

B. **Frame Sliding**: The frame sliding strategy was proposed by Chuang and Tzeng [15] to rectify the problem of internal fragmentation caused by the two-dimensional buddy scheme. In this strategy, a submesh of the exact size will be allocated to the tasks so that no extra number of nodes will be allocated.

The strategy starts searching from the lowest and leftmost free node in the mesh. The strategy will check if all the nodes in the current frame are free. If not, the frame will slide horizontally towards the right side of the mesh with a stride equal to the requested submesh width. When the frame reaches or exceeds the right boundary of the mesh, it will slide vertically with a stride equal to the requested submesh height. The frame will then start to slide towards the left side of the mesh. The sliding of the frame will continue until a free submesh is found or when the strategy has finished searching the whole mesh.

Although the frame sliding strategy eliminates internal fragmentation, it still suffers from virtual fragmentation due to its incomplete submesh recognition capability. There may exist free submeshes in the mesh system which the strategy is unable to recognize.

C. **Zhu's First Fit**: Zhu [12] introduced a First Fit strategy to resolve the problem of incomplete submesh recognition capability as with the frame sliding strategy. In the First Fit strategy, a busy array is used to describe the availability of nodes in the mesh system. When a request for a submesh arrives at the system, the strategy will derive a free-base array from the busy array by scanning it twice. In this strategy, every occupied submesh generates two domains, referred to as bottom and left coverage comprising nodes that cannot be considered as a free base. Thus, in the first scan, the left coverage is identified and in the second scan, the bottom coverage is identified and allocation to the current task is carried out. It may be noted that the second scan comes to an immediate termination as soon as a first free base is identified.

Zhu's First Fit strategy performs better than the frame sliding strategy due to its better submesh recognition capability. When a task requests a submesh $S(i, j)$, the strategy will be able to find a free submesh of size $i \times j$ if it exists in the mesh. However, the strategy does not attempt to locate a free submesh of size $j \times i$ and so its submesh recognition capability is still incomplete.

D. **Adaptive Scan**: Ding and Bhuyan [8] modified the frame sliding strategy into the adaptive scan strategy. In the adaptive scan strategy, the frame slides with an adaptive horizontal stride and a vertical stride of 1. In addition, for a request of a submesh $S(i, j)$, the strategy will first search for a submesh of size $i \times j$. If no free submesh can be found, it will search the mesh again for a free submesh of size $j \times i$.

The adaptive scan strategy eliminates the problem of virtual fragmentation, but it still suffers from external fragmentation due to its first-fit nature.

E. **Leapfrog (First-Fit)**: Recently, the leapfrog method was proposed by Wu, Hsu and Chou [3]. In this strategy, a statistical 2D-array called the R-Array or run-length array is used. R-Array has the same dimensions as the mesh. Each element in the R-Array stores an integer that represents the number of consecutive free or occupied nodes in that row starting from a node that is in the same position as the integer in the R-Array. For example, in a mesh $M(W, H)$, if the integer in $R\text{-}Array(i, j)$ is 3, it means that the nodes at $\langle i, j\rangle$, $\langle i+1, j\rangle$ and $\langle i+2, j\rangle$ are free while the node at $\langle i+3, j\rangle$ is occupied. If the integer in $R\text{-}Array(i, j)$ is -3, it means that the nodes at $\langle i, j\rangle$, $\langle i+1, j\rangle$ and $\langle i+2, j\rangle$ are occupied while the node at $\langle i+3, j\rangle$ is free. Hence, the name for the strategy.

By using the R-Array, the search time for finding a free submesh can be reduced since the strategy skipped the non-free nodes, therefore reducing the search space. The leapfrog method can be used to implement both first-fit and best-fit strategies.

3.1.2 Best-fit Strategies

Next, another class of processor allocation strategies, known as best-fit strategies, will be described.

A. **Zhu's Best Fit**: Zhu [12] also proposed another best-fit strategy in addition to the above described First Fit strategy. A busy array is used and the free-base array is constructed by scanning the busy array twice. However, unlike the First Fit strategy, the Best Fit strategy does not stop its search when it finds a free base node in the second scan. Instead, it continues to generate the entire free-base array. The strategy then tries to select the best-fit submesh. It will try to select the free base node that is at the corner of the smallest block of free base nodes. If more than one such free base node exists, it will select the base node that is surrounded by the least number of free base nodes.

   Just like the First Fit strategy, the Best Fit strategy still suffers from the problem of virtual fragmentation since it does not consider the free submesh of both orientations.

B. **Busy List**: Sharma and Pradhan [9] proposed the busy list strategy. In this strategy, a list of all the allocated submeshes in the mesh is maintained. The strategy will search for all the free submeshes that are either at the four corners of the mesh or next to any of the allocated submesh. Every time a free submesh is found, its boundary value is calculated. At the end of the search, the free submesh that has the highest boundary value will be allocated to the task.

Since the busy list method is a best-fit strategy, it significantly reduces the amount of fragmentation in the mesh. Most of the existing literature uses this strategy to compare the performance.

C. **Leapfrog (Best-Fit)**: The leapfrog strategy [3] can also be used to realize Zhu's Best Fit strategy. It uses the R-Array to construct the free-base array. Since the leapfrog method can skip the non-free nodes, it can construct the free-base array much faster. It then uses the same heuristics as Zhu's Best Fit strategy to choose the best-fit submesh.

3.2 Task Migration Strategies

In this section, a few task migration strategies that are used in mesh multicomputers will be described. These strategies assume that the destination submesh is known.

A. **Diagonal Scheme, Gathering-Routing-Scattering Scheme**: Yu, Chang and Chen [2] proposed a few schemes that can be used to migrate tasks in mesh multicomputers. The diagonal scheme tries to migrate the subtasks in phases. In each phase, all the subtasks to be migrated must reside in nodes that are not in the same row or column. In each phase, the subtasks will be migrated in the x-direction first followed by the y-direction. In this way, the routing in each phase will be congestion free.

A second scheme, called the gathering-routing-scattering scheme, tries to gather all the subtasks from the nodes in the same row to one particular node in that row. The nodes that collect the subtasks in each row must be in different columns. After the gathering operation, these nodes will be routed to the destination nodes. Upon reaching the destination nodes, the scattering operation will occur. In this operation, all the subtasks contained in the destination nodes will be redistributed to the other nodes in the same row.

A hybrid scheme which combines the above two schemes is also proposed. In this scheme, the mesh will be partitioned into several subpartitions. In each subpartition, the gathering operation is used to collect the subtasks into nodes. Next, the diagonal scheme is used to route the subpartitions in phases. Lastly, at the destination nodes, the scattering operation is carried out to redistribute the subtasks. The efficiency of this scheme will depend on how the mesh are divided into subpartitions.

B. **G-TMS, NO-TMS**: Wang and Chen [1] proposed the two schemes to migrate tasks in a two-dimensional mesh system. In the General Task Migration Scheme (G-TMS), the subtasks are migrated in phases. In the first phase, the subtasks at the boundary nodes of the source submesh are migrated in four different directions to the destination submesh. In subsequent phases, the subtasks at the boundary nodes of the remaining source submesh is again migrated in four directions to the destination nodes until all the subtasks are migrated. In the Near Optimal Task Migration Scheme (NO-TMS), the subtasks at the corner nodes and the nodes near the center line of the source submesh are migrated in the first phase. After the first phase, the scheme migrate the subtasks at nodes located at the outermost corner and near the center line of the remaining source submesh. Both the schemes are proven to be contention free in each phase.

There are certain constraints that must be met in order to use the two strategies. The source and destination submeshes must not be overlapped in the same columns or rows in order for the subtasks to migrate without contention. Another relieved constraint allows the submeshes to be overlapped in either the same rows or columns but not both. In this case, the subtasks have to use longer paths to migrate and the two submeshes have to be at a certain distance away from each other in order to accommodate these paths.

Besides task migration in mesh multicomputers [1, 2], there are also several literature related to task migration in hypercube multicomputers [4, 6, 13, 17]. However, a common assumption in all the literature is that the destination submeshes or subcubes are known and the strategies concentrate on finding disjoint parallel paths between the source and destination for the hypercube and mesh multicomputers.

In a sense, in all the existing task migration strategies, the problem of task allocation is never considered as a primary problem. The goal in these migration problems is to determine the best possible parallel paths to reach a destination from a source by assuming that task allocation has already taken place. Some works assume that task allocation uses compaction strategies which are secondary to task migration problems. The literature in task migration concentrates on determining optimal routes to the destination and does not care about the searching process of a destination submesh.

The above argument serves as a considerable motivation in this research to fuse these task allocation and migration steps. Thus, as an efficient solution to the task allocation problem, task migration should be used whenever required, depending on the decision (as will be explained later in Chapter 4). This is in view of improving the fragmentation problem and also maximizing the throughput of the mesh system.

Chapter 4

Proposed Algorithms

In this chapter, a few different strategies on task allocation will be presented. First an allocation strategy, referred to as *First-fit mesh-bifurcation (FFMB) strategy*, is presented. This strategy improves all the so far first-fit strategies described in Chapter 3. Secondly, two different task migration schemes that can be used on the above FFMB task allocation scheme are presented to improve its performance.

4.1 First-fit Mesh-bifurcation (FFMB) Strategy

This proposed strategy aims to improve the performance of first-fit allocation strategies described in Chapter 3. First-fit strategies have shorter search time but produce a significant amount of fragmentation due to their first-fit nature. On the other hand, best-fit strategies choose a free submesh that contributes least to the fragmentation in the mesh and so fragmentation is reduced. However, the search time of best-fit strategies is very high as it involves finding all the possible free submeshes in the mesh system and then deciding on which free submesh contributes least to fragmentation. In some situations, for example when the mesh size is very large, the overhead incurred by the long search time overweighs the benefits of reducing fragmentation in the system.

The strategy that will be proposed in this section aims to improve the performance of first-fit allocation strategies without incurring extra search time and reduces external fragmentation. The key idea behind the scheme is as follows.

Nearly all first-fit strategies start their search for a free submesh from the bottom-left corner of the mesh, moving rightwards in the same row until the right side of the mesh is reached. They will then move upwards and continue their search in the new row. The proposed strategy starts the search for a free submesh in one of two positions: *bottom-left corner* (BLC) and *top-left corner* (TLC) of the mesh. Thus, the mesh is bifurcated into two domains and the search starts from one of the above mentioned respective nodes. While the practice of initiating a search from the BLC is common, the choice on initiating from the TLC is not arbitrary. This choice was mainly to minimize the fragmentation and improving the search time. The starting position of the search depends on the number of free nodes in the bottom and top half of the mesh. If there are more free nodes in the bottom half of the mesh, the strategy will start its search from the BLC of the mesh, just like the original first-fit strategies. However, if there are more free nodes in the top half of the mesh, the strategy starts its search from the TLC of the mesh, moving rightwards and downwards as it continues its search.

By using two starting points instead of one, the proposed strategy hopes to achieve two things. Firstly, when the search starts from the corner of one half in which there are more free nodes, there is a higher chance that the strategy will be able to find a free submesh faster. Consider a mesh system in which nearly all the nodes in the bottom half of the mesh are occupied while most of the nodes in the top half are free. By starting the search from the TLC, a free submesh can be found faster when compared to starting the search from the BLC. Secondly, the strategy hopes to reduce the amount of external fragmentation by allocating the submeshes near the bottom and top edge of the mesh so that the original first-fit configuration shifts towards more of a best-fit configuration.

Thus in order to realize this strategy, two counts are typically maintained, each comprising the number of free nodes for each half of the mesh. With the above description, the search time is expected to reduce by a factor of 0.5. However, the complexity remains unaltered and is the same as the existing first-fit strategies described in the literature. This search time will be captured in the simulation

Task	Requested Submesh Size	Arrival Time	Processing Time
1	3×2	0	2.5
2	6×4	1	3.5
3	3×3	2	3.5
4	3×4	4	4
5	5×5	5	4

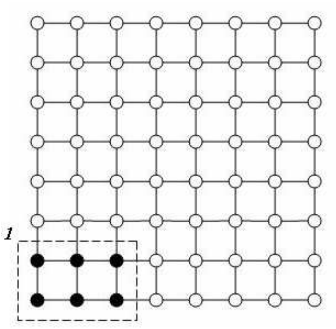
Table 4.1: Dimensions of requested submesh, arrival and processing time of the 5 tasks that arrive at $M(8,8)$

studies during the task allocation phase and the simulation will also show that external fragmentation is minimized to a large extent with the FFMB strategy. The following example clarifies the workings of the proposed strategy and shows that external fragmentation is minimized to a large extent.

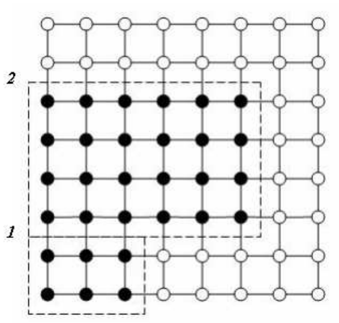
4.2 Illustrative Example for FFMB

Consider the situation when 5 tasks arrive to a mesh $M(8,8)$. The arrival times, the dimensions of the requested submeshes and the processing times are shown in Table 4.1. Suppose the leapfrog (first-fit) method is used to allocate the submeshes. For the purpose of illustration, it is assumed that the search time (t_{search}) and the maintenance time required for allocation ($t_{maintainAlloc}$) and deallocation ($t_{maintainDealloc}$) are negligible. The leapfrog (first-fit) strategy always starts its search from the BLC of the mesh. When Task 1 arrives at $t = 0$, it is allocated to the submesh at $\langle 0,0,2,1 \rangle$. Task 2 arrives at $t = 1$ and gets allocated to the submesh at $\langle 0, 2, 5, 5\rangle$. However, when Task 3 arrives at $t = 2$, it cannot be allocated to the mesh even though there are enough free nodes. This task is inserted into a queue so that it can be allocated later. At $t = 2.5$, Task 1 is deallocated. Task 3 in the queue is still unable to locate a free submesh for allocation. Task 4 arrives at $t = 4$ and similarly it cannot locate a free submesh and is inserted into the queue. Task 2 is deallocated at $t = 4.5$ and now both Task 3 and Task 4 in the queue are able to locate a free submesh at $\langle 0, 0, 2, 2\rangle$ and $\langle 3, 0, 5, 3\rangle$ respectively. When Task 5 arrives at $t = 5$, it is unable to locate a free submesh in the system and is inserted into the queue. This task has to wait until both Task 3 and Task 4 are deallocated from the system before it is able to be allocated to the submesh at $\langle 0, 0, 4, 4\rangle$. Figure 4.1 shows the phases during the allocation and deallocation of tasks using the leapfrog (first-fit) strategy.

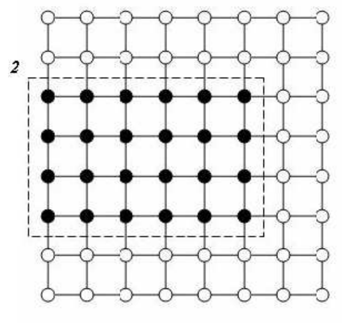
Next the FFMB strategy is applied to the leapfrog (first-fit) strategy. Task 1 is allocated to the submesh at $\langle 0, 0, 2, 1\rangle$ just like in the previous case. However, when Task 2 arrives at $t = 1$, it is allocated to the submesh at $\langle 0, 4, 5, 7\rangle$. This is because the FFMB strategy starts the search of a free submesh from the TLC since there are more free nodes in the top half of the mesh system. When Task 3 arrives at $t = 2$, it starts the search from the BLC of the mesh as there are now more free nodes in the bottom half of the mesh. A free submesh at $\langle 3, 0, 5, 2\rangle$ can be located and allocated to Task 3 immediately. Next, Task 1 will be dellocated at $t = 2.5$.



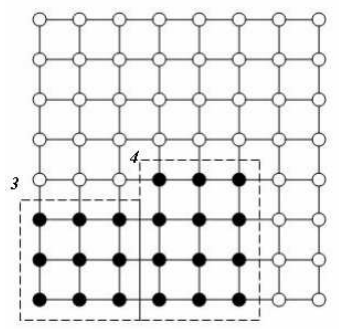
(a) $t = 0$, Task 1 allocated



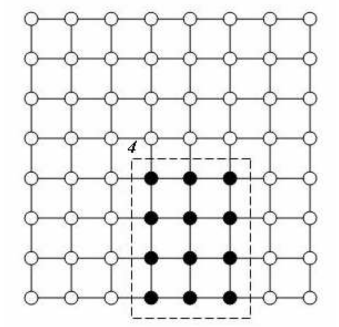
(b) $t = 1$, Task 2 allocated



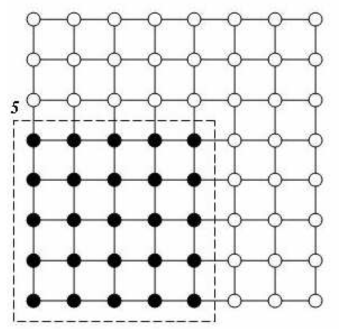
(c) $t = 2.5$, Task 1 deallocated



(d) $t = 4.5$, Task 2 deallocated, Task 3 reallocated, Task 4 reallocated



(e) $t = 8$, Task 3 deallocated



(f) $t = 8.5$, Task 4 deallocated, Task 5 reallocated

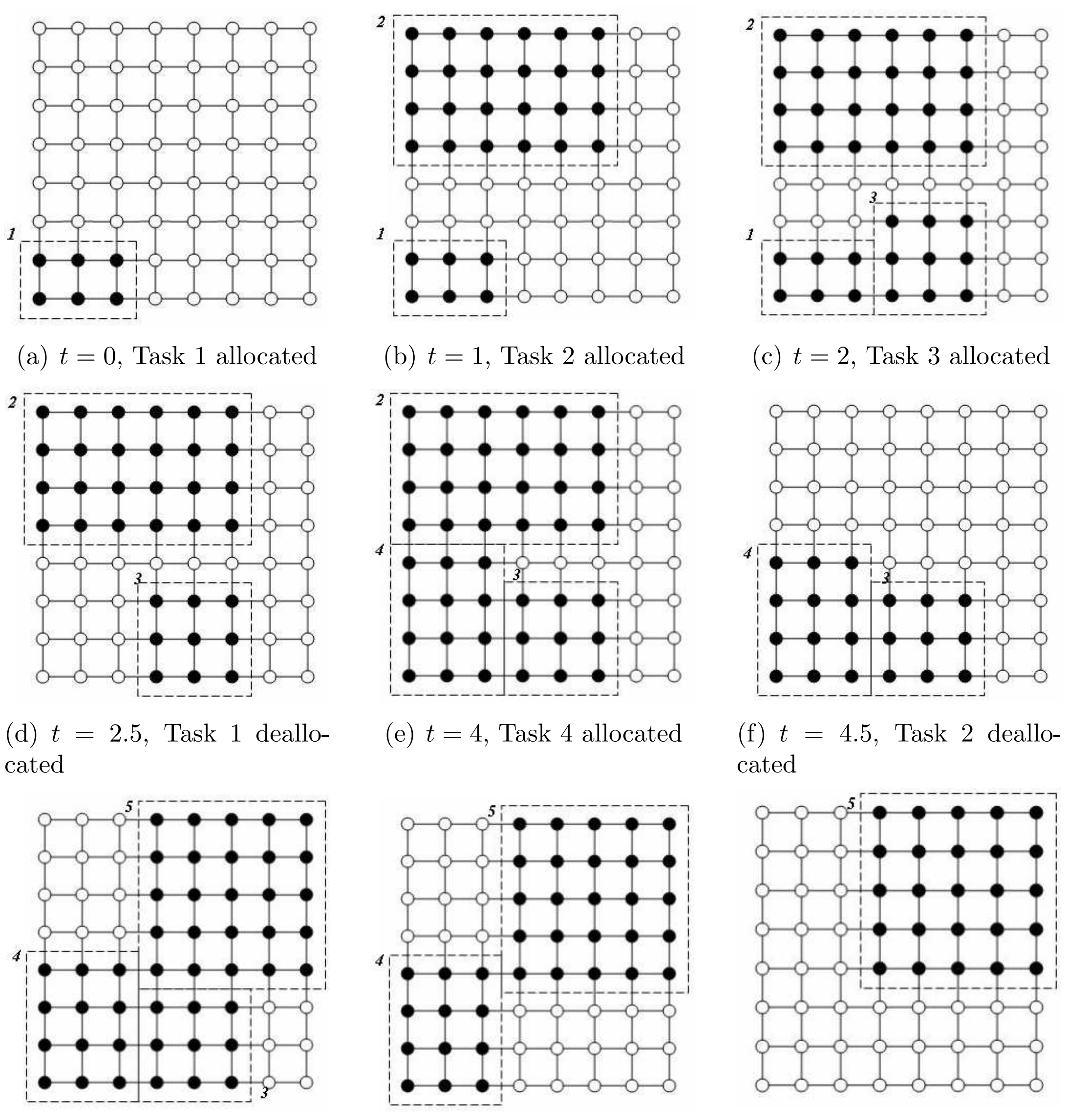
Figure 4.1: Allocation and deallocation of tasks using the leapfrog (first-fit) strategy

When Task 4 arrives at $t = 4$, the strategy can also locate a submesh at $\langle 0, 0, 2, 3 \rangle$. At $t = 4.5$, Task 2 is deallocated from the mesh. Lastly, Task 5 arrives at $t = 5$ and is allocated to the submesh at $\langle 3, 3, 7, 7 \rangle$. Figure 4.2 shows the phases during the allocation and deallocation of tasks using the leapfrog (first-fit) strategy with the FFMB strategy.

From the above example, it can be observed that when the FFMB strategy is used, all the 5 tasks are able to locate a free submesh the moment they arrive to the mesh system. Therefore, the FFMB strategy is able to reduce the amount of delay time and external fragmentation in the mesh so that all the tasks can be accommodated.

4.3 Online Dynamic Compaction: Single Corner Task Migration (ODC-SC) Strategy

After a series of successive allocation and deallocation of submeshes, the mesh system will still suffer from a significant amount of external fragmentation even if a best-fit allocation strategy is used. In order to rectify this problem, a compaction or task migration strategy is required to rearrange or reschedule the active tasks in the mesh so as to create a larger area of contiguous free nodes for future allocation. The disadvantage of performing compaction or task migration is that the active tasks in the mesh have to be suspended for the period of task migration and they can only continue after the task migration has been completed. Previous literature [1, 2, 4, 6, 13, 17] assumes that the destination submesh is known a priori and concentrates on finding parallel paths between the source and destination submeshes so that task migration can be completed in a shorter amount of time. This therefore reduces the overhead incurred for suspending the active tasks. The proposed strategy concentrates on locating a destination submesh such that when an active task is to be migrated towards the destination submesh, only the task



(a) $t = 0$, Task 1 allocated (b) $t = 1$, Task 2 allocated (c) $t = 2$, Task 3 allocated

(d) $t = 2.5$, Task 1 deallocated (e) $t = 4$, Task 4 allocated (f) $t = 4.5$, Task 2 deallocated

(g) $t = 5$, Task 5 allocated (h) $t = 5.5$, Task 3 deallocated (i) $t = 8$, Task 4 deallocated

Figure 4.2: Allocation and deallocation of tasks using the leapfrog (first-fit) strategy with FFMB

to be migrated needs to be suspended. All other active tasks in the mesh will continue with their computation during the migration of this task.

The proposed task migration strategy will be described using the leapfrog (first-fit) method as the underlying task allocation strategy. However, it should be noted that the task migration strategy can be applied to any of the allocation strategies that are described in Chapter 3. Consider the case when a request for a submesh $S(w, h)$ arrives to the mesh system $M(W, H)$. The system first uses the leapfrog strategy to search for a free submesh. If a free submesh is found, it is allocated to that task. If no free submesh is found and there are enough free nodes in the mesh to satisfy the request, the task migration scheme is employed to migrate the running tasks. After task migration, the system uses the leapfrog strategy again to search the mesh for a free submesh. Since the amount of external fragmentation is reduced after task migration, there is a higher chance that the system is now able to satisfy the request. This is one of the properties exploited in the design of the proposed strategy. The task migration strategy is also employed at most once between any two successive deallocation of submeshes. This is to ensure that no excessive task migration will be done since each time the tasks are migrated, a significant amount of overhead is incurred. In addition, there will not be any significant improvement to the condition of fragmentation in the mesh if task migration is performed more than once between two successive deallocations. The algorithm for the task migration scheme is described in Figure 4.3. Below are a few terms that are used in the algorithm:

(a). *MigrationDone:* Boolean flag that indicates whether task migration has been performed since the last deallocation event. This flag is set to false after every deallocation event and set to true after task migration is performed. The use of this flag is to ensure that at most one task migration is done between two successive deallocations.

(b). *(BaseNode_x, BaseNode_y):* Integer variables that stores the coordinates of the base node of the new submesh found so far during each iteration of the task migration strategy.

(c). *PositionNo:* All the allocated submeshes can be identified by a position number. The position number of each submesh can be calculated as follows:

$$PositionNo = (y' \times W) + x' + 1 \tag{4.1}$$

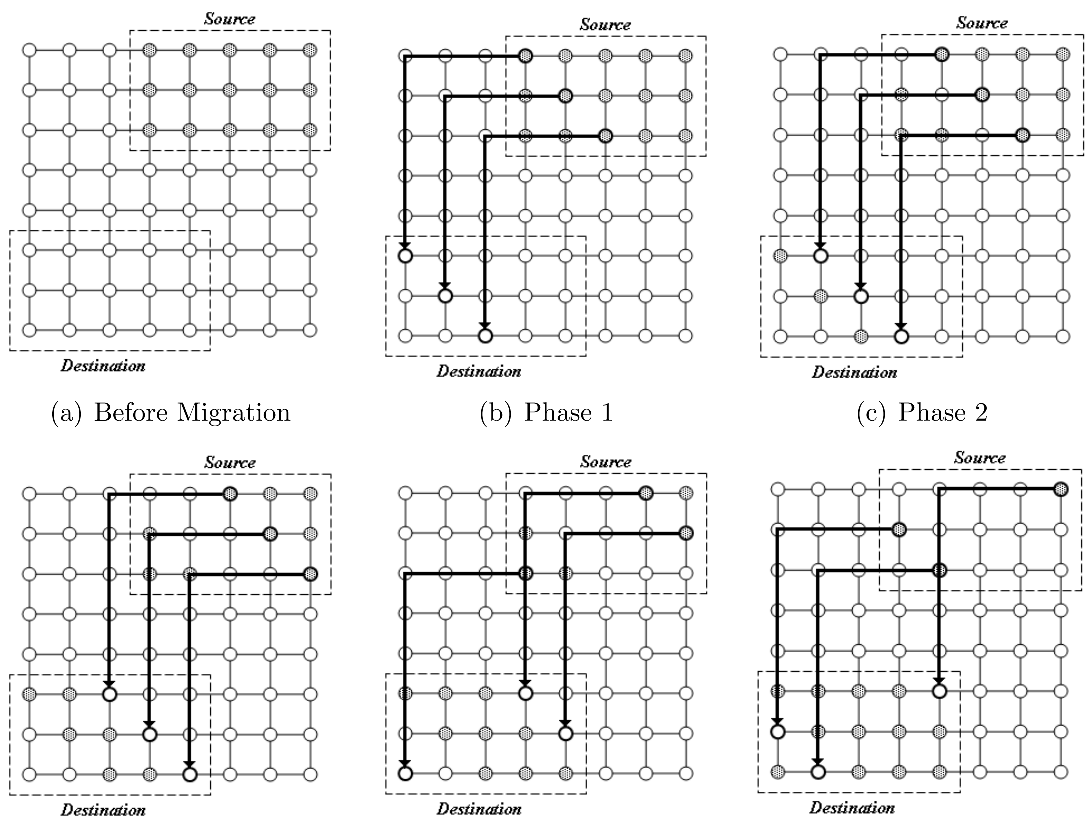
where (x', y') are the coordinates of the base node of the submesh. For example, the position numbers in a 5×4 mesh is shown below:

$$\begin{pmatrix} 16 & 17 & 18 & 19 & 20 \\ 11 & 12 & 13 & 14 & 15 \\ 6 & 7 & 8 & 9 & 10 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

```
Algorithm ODC-SC

Step 1: Check whether there are enough free nodes to satisfy the request.
        If (not enough free nodes)
            Go to Step 5.
        Else {
            Search for a free submesh using the leapfrog allocation strategy.
            If (free submesh is found)
                Go to Step 4.
            Else {
                If ($MigrationDone = true$)
                    Go to Step 5.
                Else {
                    Assign $MigrationDone \leftarrow true$
                    Assign $CurSubmesh \leftarrow$ allocated submesh with lowest $PositionNo$
                    Let the address of this allocated submesh be $\langle x_1, y_1, x_2, y_2 \rangle$.
                    Assign $CurNode\_x \leftarrow x_1$, $CurNode\_y \leftarrow y_1$ and go to Step 2.
        }}}

Step 2: Assign $BaseNode\_x \leftarrow CurNode\_x$
        Assign $BaseNode\_y \leftarrow CurNode\_y$
        While (all nodes adjacent to the left edge of the allocated submesh are free) {
            Assign $BaseNode\_x \leftarrow BaseNode\_x - 1$ /* Shift allocated submesh leftwards */
        }
        While (all nodes adjacent to the bottom edge of the allocated submesh are free) {
            Assign $BaseNode\_y \leftarrow BaseNode\_y - 1$ /* Shift allocated submesh downwards */
        }
        If ($BaseNode\_x = CurNode\_x$) and ($BaseNode\_y = CurNode\_y$)
            Go to Step 3.
        Else {
            Update the list of allocated submeshes and the $R$-Array with the new position
            of the allocated submesh. Start the actual migration process to move the task
            from the source to destination submesh using the diagonal scheme [2].
        }

Step 3: Assign $CurSubmesh \leftarrow$ allocated submesh with next lowest $PositionNo$
        If ($CurSubmesh \neq NULL$) {
            Let the address of this allocated submesh be $\langle x_3, y_3, x_4, y_4 \rangle$.
            Assign $CurNode\_x \leftarrow x_3$, $CurNode\_y \leftarrow y_3$ and go to Step 2.
        } Else {
            Search for a free submesh again using the leapfrog allocation strategy.
            If (free submesh is found)
                Go to Step 4.
            Else
                Go to Step 5.
        }

Step 4: Allocate the located submesh to the task. Consider the next task in the allocation
        queue for scheduling. Go to Step 1.

Step 5: Insert the task into the reallocation queue for future allocation. Consider the next
        task in the allocation queue for scheduling. Go to Step 1.
```

Figure 4.3: Algorithm ODC-SC

The task migration algorithm tries to migrate the running tasks towards the BLC of the mesh so as to create a larger contiguous area of free nodes in the top and right side of the mesh. In Step 2 of the algorithm, the subtasks in the source submesh are migrated in phases to the destination submesh using the diagonal scheme [2]. In each phase, the subtasks are migrated using X-Y routing. In X-Y routing, a subtask that is to be migrated from $\langle a, b\rangle$ to $\langle c, d\rangle$ will first move along the row to $\langle c, b\rangle$ before moving along the column to $\langle c, d\rangle$. Figure 4.4 shows an example of migrating a task from the submesh at $\langle 3, 5, 7, 7\rangle$ to the submesh at $\langle 0, 0, 4, 2\rangle$ in an 8×8 mesh system using the diagonal scheme. From this example, it can be seen that in each phase, the paths of the subtasks to be migrated do not share any common links and therefore contention is avoided. Although the diagonal scheme ensures that the paths that are used in each phase do not share any common links with one another, it does not ensure that these paths do not share any common links with other tasks that are running in the mesh system. The proposed algorithm rectifies this problem by searching for the destination submesh in such a way that all the paths that will be used by X-Y routing to migrate the subtasks to the located destination submesh will be free. In this way, network contention is avoided and whenever a task is being migrated, all the rest of the running tasks do not need to be suspended and can continue with their execution.



(a) Before Migration (b) Phase 1 (c) Phase 2

(d) Phase 3 (e) Phase 4 (f) Phase 5

Figure 4.4: Example of task migration using the diagonal scheme

4.4 Online Dynamic Compaction: Four Corners Task Migration (ODC-FC) Strategy

A modified version of the ODC-SC strategy will be introduced in this section. This scheme is essentially the same as the previous scheme, except that instead of moving all the tasks towards the BLC of the mesh, the modified scheme moves the tasks towards all four corners of the mesh so as to produce a larger contiguous space of free nodes at the center of the mesh. The tasks are migrated towards all four corners of the mesh instead of only towards the BLC so that the final configuration of the mesh approaches that of a best-fit configuration and hence there will be less external fragmentation after migration as compared to the previous scheme. This strategy is particularly useful in yielding better response times when jobs arrive with short deadlines. However, implementation overheads do play a role in realizing this strategy.

In this modified scheme, the mesh is divided into four quarters. The bottom-left quarter is Quarter 1, followed by the bottom-right quarter, the top-right quarter, and finally the top-left quarter. Every allocated submesh will be in one of the four quarters. If most of the nodes in the allocated submesh are in Quarter 1, then that allocated submesh will be considered to be in Quarter 1. In this scheme, all the tasks in Quarter 1 will be migrated towards the bottom-left corner (BLC) of the mesh, while all the tasks in Quarter 2, 3 and 4 will be migrated towards the bottom-right corner (BRC), top-right corner (TRC) and top-left corner (TLC), respectively. In addition, each quarter will have its own set of position numbers.

For example, the position numbers in a 6 × 4 mesh is illustrated below:

$$\begin{pmatrix} 1 & 2 & 3 & | & 3 & 2 & 1 \\ 4 & 5 & 6 & | & 6 & 5 & 4 \\ - & - & - & - & - & - & - \\ 4 & 5 & 6 & | & 6 & 5 & 4 \\ 1 & 2 & 3 & | & 3 & 2 & 1 \end{pmatrix}$$

In the previous scheme, the position number of an allocated submesh refers to the position number of the base node (BLC node) of that allocated submesh. In this modified strategy, however, the position number of a submesh is calculated depending on the quarter in which the submesh resides. If the submesh resides in Quarter 1, its position number refers to the position number of the BLC node of the submesh. If the submesh is in Quarter 2, its position number is the position number of the BRC node of the submesh. For submeshes in Quarter 3 and 4, their position numbers refer to the position numbers of the TRC and TLC nodes respectively. Complete workings of the ODC-FC strategy is shown in Figure 4.5.

The above algorithm tries to move the tasks in Quarter 1 first, followed by the tasks in Quarter 2, Quarter 3 and finally Quarter 4. In each quarter, each of the running tasks will be migrated in order of their position number in that particular quarter. Just as in the previous strategy, the algorithm ensures that all the paths

```
Algorithm ODC-FC

Step 1: Check whether there are enough free nodes to
        satisfy the request.
        If (not enough free nodes)
           Go to Step 9.
        Else {
           Search for a free submesh using the leapfrog
           allocation strategy.
           If (free submesh is found)
              Go to Step 8.
           Else {
              If ($MigrationDone = true$)
                 Go to Step 9.
              Else {
                 Assign $MigrationDone \leftarrow true$
                 Assign $Quarter \leftarrow 1$
                 Assign $CurSubmesh \leftarrow NULL$
                 While ($Quarter \leq 4$) {
                    Assign $CurSubmesh \leftarrow$ allocated
                    submesh with lowest $PositionNo$
                    located in $Quarter$
                    If ($CurSubmesh = NULL$)
                       Assign $Quarter \leftarrow Quarter + 1$
                    Else {
                       Let the address of this allocated
                       submesh be $\langle x_1, y_1, x_2, y_2 \rangle$.
                       Assign $CurNode\_x \leftarrow x_1$,
                       $CurNode\_y \leftarrow y_1$ and go to
                       Step 2.
                    }
                 }
              }
           }
        }
Step 2: Assign $BaseNode\_x \leftarrow CurNode\_x$
        Assign $BaseNode\_y \leftarrow CurNode\_y$
        If ($Quarter = 1$) or ($Quarter = 4$)
           Go to Step 3.
        Else {
           While (all nodes adjacent to the right edge
           of the allocated submesh are free) {
              Assign $BaseNode\_x \leftarrow BaseNode\_x + 1$
              /* Shift allocated submesh rightwards */
           }
        }
        Go to Step 4.
Step 3: While (all nodes adjacent to the left edge of
        the allocated submesh are free) {
           Assign $BaseNode\_x \leftarrow BaseNode\_x - 1$
           /* Shift allocated submesh leftwards */
        }
Step 4: If ($Quarter = 1$) or ($Quarter = 2$)
           Go to Step 5.
        Else {
           While (all nodes adjacent to the top edge
           of the allocated submesh are free) {
              Assign $BaseNode\_y \leftarrow BaseNode\_y + 1$
              /* Shift allocated submesh upwards */
           }
        }
        Go to Step 6.
Step 5: While (all nodes adjacent to the bottom edge
        of the allocated submesh are free) {
           Assign $BaseNode\_y \leftarrow BaseNode\_y - 1$
           /* Shift allocated submesh downwards*/
        }
Step 6: If ($BaseNode\_x = CurNode\_x$) and
        ($BaseNode\_y = CurNode\_y$)
           Go to Step 7.
        Else {
           Update the list of allocated submeshes and
           the $R$-Array with the new position of the
           allocated submesh. Start the actual
           migration process to move the task from
           the source to destination submesh using
           the diagonal scheme [2].
        }
Step 7: While ($Quarter \leq 4$) {
           Assign $CurSubmesh \leftarrow$ allocated submesh
           with next lowest $PositionNo$ in $Quarter$
           If ($CurSubmesh = NULL$)
              Assign $Quarter \leftarrow Quarter + 1$
           Else {
              Let the address of this allocated submesh
              be $\langle x_3, y_3, x_4, y_4 \rangle$
              Assign $CurNode\_x \leftarrow x_3$,
              $CurNode\_y \leftarrow y_3$ and go to Step 2.
           }
        }
        Search for a free submesh again using the
        leapfrog allocation strategy.
        If (free submesh is found)
           Go to Step 8.
        Else
           Go to Step 9.
Step 8: Allocate the located submesh to the task.
        Consider the next task in the allocation
        queue for scheduling. Go to Step 1.
Step 9: Insert the task into the reallocation queue for
        future allocation. Consider the next task in
        the allocation queue for scheduling. Go to
        Step 1.
```

Figure 4.5: Algorithm ODC-FC

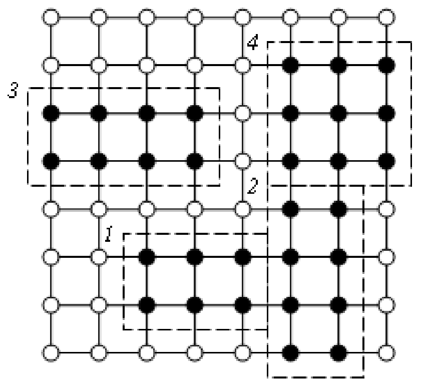


Figure 4.6: Mesh before task migration is employed

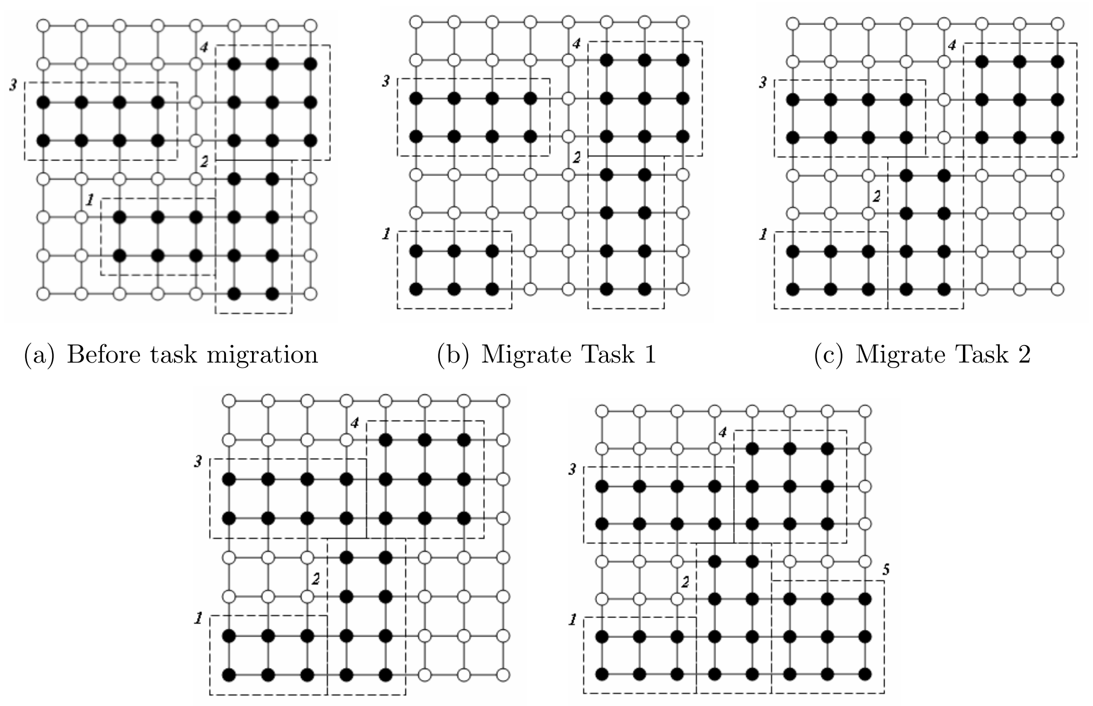
that are used to migrate the subtasks using X-Y routing are free in order to prevent network contention and allow all the other tasks to continue with their execution.

4.5 Illustrative Example for ODC-SC and ODC-FC

Consider the situation when the leapfrog (first-fit) strategy is used to locate free submeshes in a mesh $M(8,8)$. After some allocation and deallocation, the mesh becomes fragmented as shown in Figure 4.6.

Suppose the next task (Task 5) arrives and requests for a submesh $S(3,3)$. There are enough free nodes in the mesh to satisfy the request, but they do not form a submesh of dimension 3×3. Now if the ODC-SC strategy is employed, Task 1 will be the first task to be migrated since its position number of 11 is the smallest among all the tasks in the mesh, followed by Task 2, 3 and 4. Task 1 will locate a destination submesh at $\langle 0,0,2,1\rangle$ using the ODC-SC algorithm. It should be noted that all the paths used to migrate Task 1 from $\langle 2,1,4,2\rangle$ to $\langle 0,0,2,1\rangle$ using X-Y routing are all free and all the other running tasks in the mesh need not suspend their operations. Similarly, Task 2 will be migrated to the submesh at $\langle 3,0,4,3\rangle$. Task 3 will not be able to locate a destination submesh for migration. Finally Task 4 is migrated to the submesh at $\langle 4,4,6,6\rangle$. After task migration, the degree of external fragmentation is reduced and a free submesh can now be located at $\langle 5,0,7,2\rangle$ to be allocated to Task 5. The phases of using the ODC-SC strategy is shown in Figure 4.7. However, there still exists some degree of external fragmentation in the mesh. If another task (Task 6) arrives to the mesh requesting for a submesh $S(3,4)$, the mesh will not be able to allocate a free submesh to satisfy this request.

Next, the ODC-FC strategy is used instead of the ODC-SC strategy to allocate the tasks. The strategy will first try to migrate all the tasks in Quarter 1. Only Task 1 belongs to Quarter 1 since 4 out of 6 of its nodes are located in Quarter 1. This task will be migrated to the submesh at $\langle 0,0,2,1\rangle$. Similarly, Task 2 in Quarter 2, Task 4 in Quarter 3 and Task 3 in Quarter 4 are migrated to the submeshes at $\langle 6,0,7,3\rangle$, $\langle 5,5,7,7\rangle$ and $\langle 0,6,3,7\rangle$ respectively. After task migration, a free submesh at $\langle 3,0,5,2\rangle$ can be allocated to Task 5. The ODC-FC strategy results in less external fragmentation as compared to the ODC-SC strategy. Even when

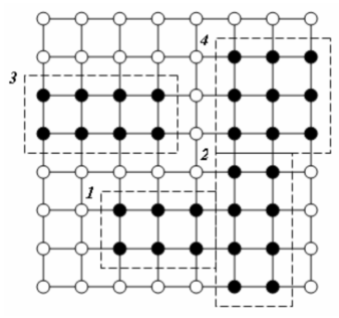


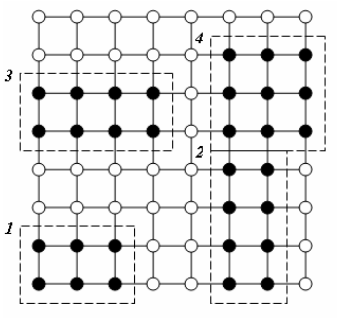
(a) Before task migration (b) Migrate Task 1 (c) Migrate Task 2

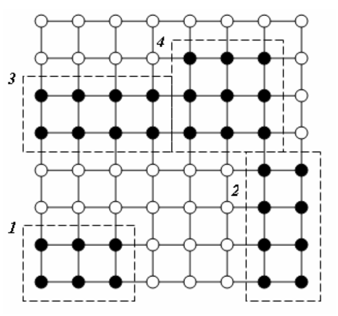
(d) Task 3 cannot be migrated, Migrate Task 4 (e) Allocate Task 5

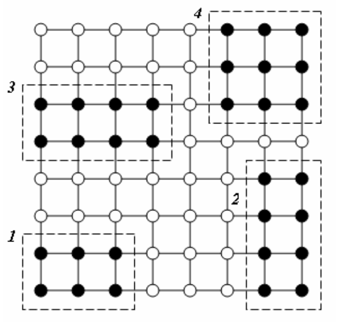
Figure 4.7: Phases of migration using the ODC-SC strategy

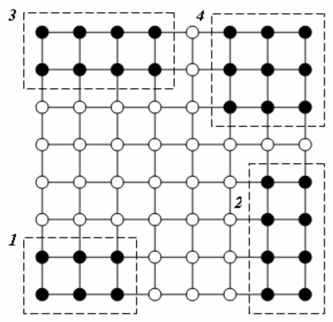
Task 6 arrives to the system, a free submesh can still be located at $\langle 0, 2, 2, 5 \rangle$ to be allocated to the task. The phases of migration using the ODC-FC strategy are shown in Figure 4.8.

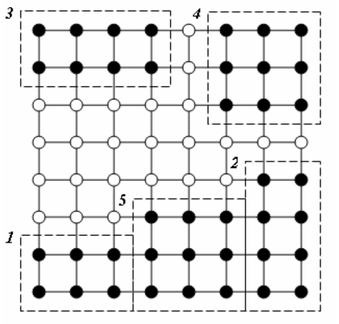

(a) Before task migration


(b) Migrate Task 1 in Quarter 1


(c) Migrate Task 2 in Quarter 2


(d) Migrate Task 4 in Quarter 3


(e) Migrate Task 3 in Quarter 4


(f) Allocate Task 5

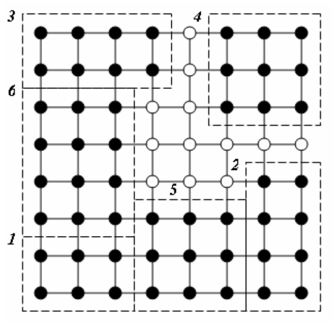

(g) Allocate Task 6

Figure 4.8: Phases of migration using the ODC-FC strategy

Chapter 5

Simulation Results and Discussions

In this chapter, the performance study on all the strategies proposed in Chapter 4 will be reported. The model that is described in Chapter 2 is used to compute all the individual components (arrival, search, allocation, processing and deallocation) in the performance evaluation of all the strategies to compare their behavior. The exact details on the implementation study will be presented in Appendix A. The key performance metric used in this study is the *average delay time* $\bar{t}_{delay}$[1], as defined in Chapter 2. The performance of the proposed strategies with respect to fragmentation, which is also an indicator of performance, will be highlighted as well.

[1]A bar indicates that this is an average of the delay defined in Chapter 2

5.1 Simulation Parameters

The performance of the proposed algorithms is studied using an event-driven simulation. The events in the simulation are allocation, deallocation and reallocation. Several parameters will be used in this simulation model. In this simulation study, all the parameters are generated randomly following some particular probability distributions. The width and height of the submeshes requested by the incoming tasks are derived from a normal distribution with mean $\bar{w}$ and $\bar{h}$ respectively. It is assumed that the processing time of the tasks follows an exponential distribution with a mean of $\bar{t}_{process}$ and the interarrival time of the tasks follows the exponential distribution with a mean that is determined by the system load s, where $0 < s \leq 1$. The system load is defined as follows:

$$s = \frac{\bar{w} \times \bar{h} \times \bar{t}_{process}}{W \times H \times \bar{t}_{interarrival}} \tag{5.1}$$

where $\bar{t}_{interarrival}$ is the mean interarrival time of the task. In this study, the system load will be varied and the parameter $\bar{t}_{interarrival}$ is tuned accordingly to generate a sequence of arrival events corresponding to a particular system load.

In addition, it is assumed that the mesh system uses wormhole routing to send messages from one node to another. It has been demonstrated that wormhole routing is distance insensitive [7]. The latency $t_{latency}$in sending one message with

l bytes to another node using wormhole routing is given by:

$$t_{latency} = t_s + t_x \times l \tag{5.2}$$

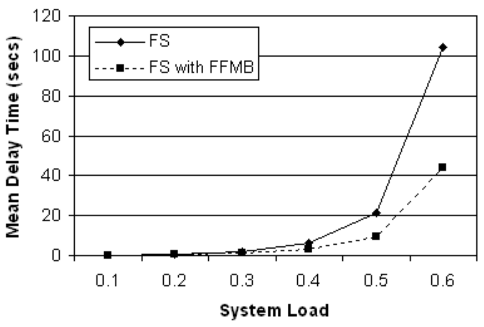
where t_s is the startup latency and t_x is the transmission time of one byte. In the proposed ODC-SC and ODC-FC strategies, the diagonal scheme [2] is used to migrate the subtasks in phases. The number of phases P required to migrate a subtask $S(w,h)$ is given by $max(w,h)$. The total latency $t_{migrate}$ in migrating a task from the source submesh to the destination submesh is therefore given by:

$$t_{migrate} = (t_s + t_x \times l) \times P \tag{5.3}$$

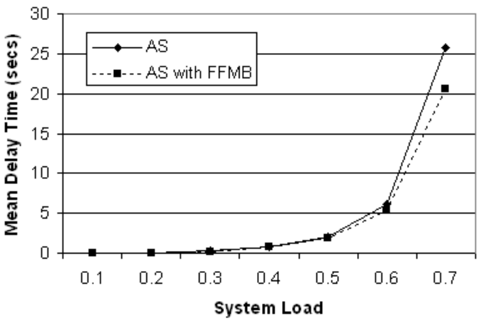
assuming that all the subtasks at each node have the same size of l bytes. Therefore each migrating task will be suspended for a period of $t_{migrate}$ during migration.

5.2 Performance in Minimizing Delay Time

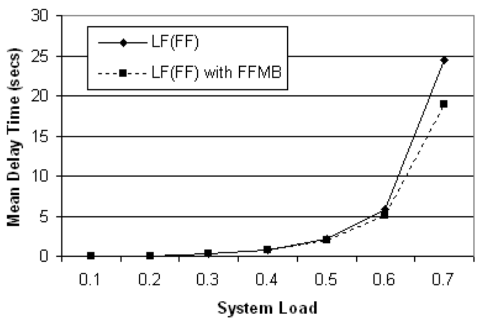
In this section, the performance of the proposed strategies in minimizing the mean delay time will be described.



(a) Frame sliding



(b) Adaptive scan



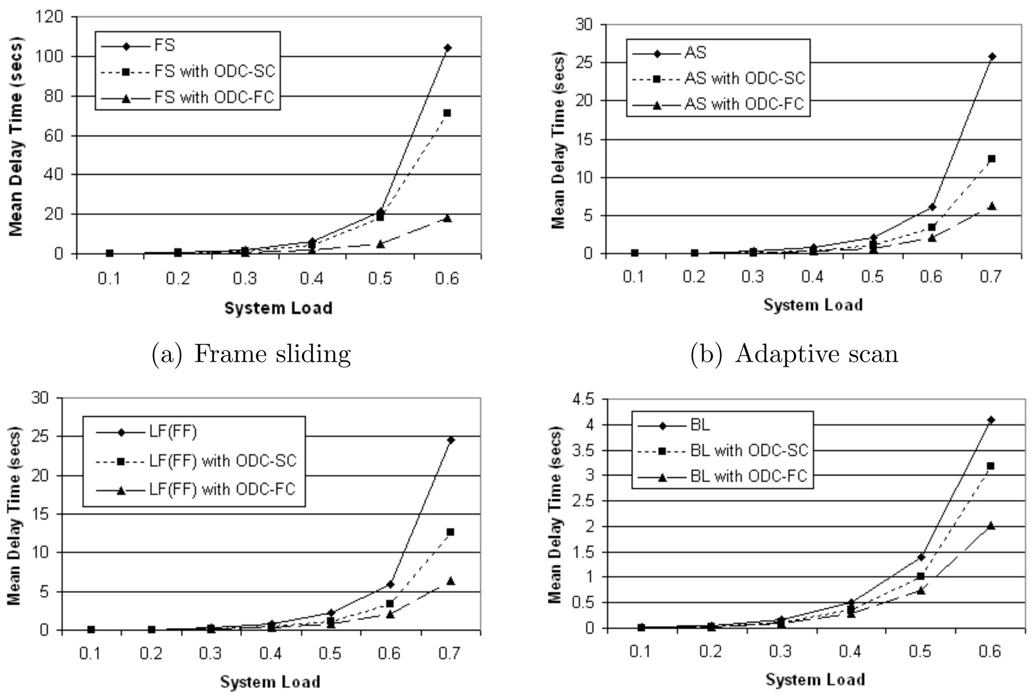
(c) Leapfrog (first-fit)

Figure 5.1: Performance of FFMB strategy

5.2.1 FFMB Strategy

First, the FFMB scheme is applied to the frame sliding, adaptive scan and the leapfrog (first-fit) strategies, on a mesh system $M(64, 64)$. Both the width and height of the requested submeshes have a mean of 16 and a variance of 8. The mean processing time is taken to be 10 seconds. The simulation is run for ten different sets of arrival events, with each set consisting of $10,000$ tasks, and the average is obtained. Figure 5.1 shows the mean delay time of the three strategies before and after applying our FFMB scheme at different system load.

From the figure, it can be seen that there is a significant reduction by 45 to 58 percent in the mean delay time when our FFMB scheme is used with the frame sliding strategy. However, similar effect is not apparent when the FFMB scheme is employed with the adaptive scan and the leapfrog (first-fit) strategies. From the simulation, both the adaptive scan and leapfrog (first-fit) strategies show an improvement of about 10 to 23 percent when the FFMB scheme is employed. The frame sliding is not an efficient strategy due to its incomplete submesh recognition capability as explained in Chapter 3, resulting in a large amount of virtual and external fragmentation. The proposed FFMB strategy is able to improve its performance by significantly reducing the amount of fragmentation in the mesh system and allowing more submeshes to be allocated to the tasks. On the other hand, both the adaptive scan and the leapfrog (first-fit) strategies already have complete submesh recognition capabilities built-in and hence introduce a significantly smaller amount of fragmentation to the mesh system as compared to the frame sliding strategy. Therefore the FFMB scheme is less influential in improving their performance. Also, from the figures, it is observed that the effect of the FFMB strategy thrives hard to render a good performance under higher system loads, which is to be expected of an efficient strategy.



(a) Frame sliding (b) Adaptive scan

(c) Leapfrog (first-fit) (d) Busy list

Figure 5.2: Performance of ODC-SC and ODC-FC strategies

5.2.2 ODC-SC and ODC-FC Strategies

Next, both the ODC-SC and ODC-FC strategies are applied to the frame sliding, adaptive scan, leapfrog (first-fit) and busy list strategies. The same set of parameters for the mesh considered earlier is used. In addition, the startup latency in migrating a task is taken to be 10 microseconds and the transmission time of one byte of message is 20 nanoseconds [1, 2]. Each subtask is assumed to be 1 kilobyte in size[2]. As before, ten sets of runs, each consisting of $10,000$ tasks, are carried out and an average value is obtained.

[2]These are typical values derived from [1, 2]

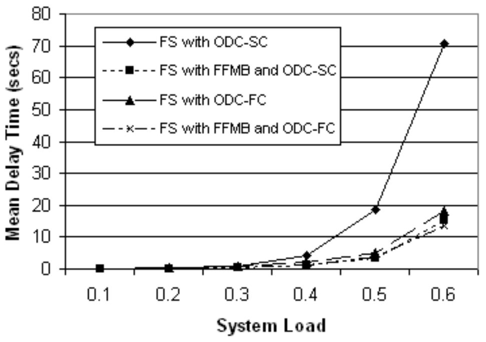
Figure 5.2 shows the mean delay time of the different strategies with and without the ODC-SC and ODC-FC schemes. As seen from the figure, the ODC-SC scheme is able to improve the performance of all the various allocation strategies. The frame sliding strategy shows an improvement of 13 to 54 percent when the ODC-SC strategy is used, while the adaptive scan and leapfrog (first-fit) both improves their mean delay time by 45 to 62 percent. Further, ODC-SC improves the busy list strategy by about 12 to 29 percent. The ODC-FC scheme performs even better for all the strategies. When used with the frame sliding strategy, the mean delay time is reduced by 65 to 82 percent. Both the adaptive scan and the leapfrog (first-fit) strategy shows a 65 to 76 percent reduction in mean delay time and the busy list strategy improves its performance by 20 to 51 percent.

From the results using the busy list strategy, it can be seen that the proposed ODC-SC and ODC-FC schemes can also be used with best-fit strategies. However, the improvement in performance is smaller for the busy list strategy as compared to the other strategies. This is due to the fact that the busy list strategy itself results in very little external fragmentation and therefore there is a limit to the improvement that can be realized. In addition, it should also be noted that the mean delay time when using the busy list with the ODC-SC scheme is approximately the same as when ODC-SC scheme is used with either the adaptive scan or the leapfrog (first-fit) strategy. This is also true when the ODC-FC strategy is used. Therefore, it can be seen that although the busy list strategy performs better than both the adaptive scan and leapfrog (first-fit) strategy without task migration, all the three strategies performs equally well when used together with either the ODC-SC or the ODC-FC schemes.

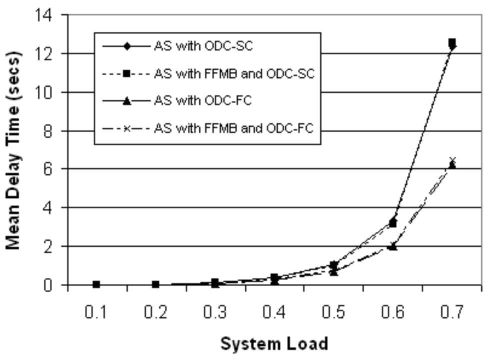
5.2.3 FFMB with ODC-SC and ODC-FC Strategies

Lastly, the FFMB scheme is employed together with either the ODC-SC or the ODC-FC scheme to the three first-fit allocation strategies and their performance is observed. Figure 5.3 shows the results of the study.

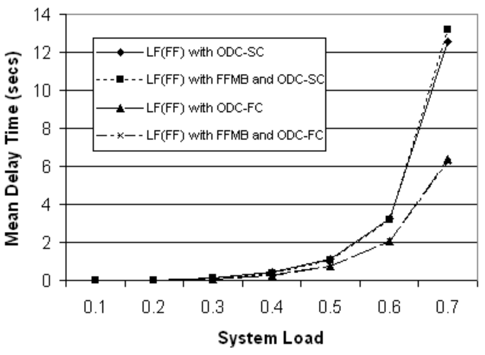
For the frame sliding strategy, there is an improvement of 62 to 82 percent if the FFMB scheme is used with the ODC-SC scheme, as compared to the case when the ODC-SC strategy is used alone. However, there is only about 24 to 30 percent improvement when the FFMB scheme is used with the ODC-FC scheme. Furthermore, for the adaptive scan and the leapfrog (first-fit) strategies, there is only a little difference in performance whether the FFMB scheme is used with the two task migration strategies. The mean delay time is still approximately the same when the FFMB scheme is used together with the ODC-SC or ODC-FC schemes, as compared to the case when no FFMB scheme is used. From these results, we can conclusively see that there is no need to use the FFMB scheme when either the ODC-SC or the ODC-FC strategy is employed. However, these results indicate



(a) Frame sliding



(b) Adaptive scan



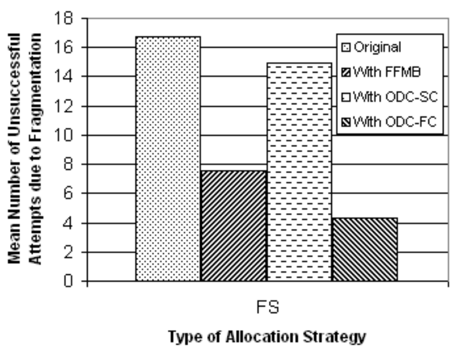
(c) Leapfrog (first-fit)

Figure 5.3: Performance of FFMB with ODC-SC and ODC-FC

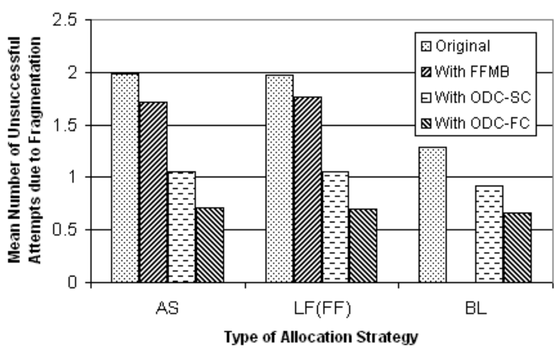
clearly that the FFMB strategy should be used to improve the performance of first-fit strategies only in the absence of task migration.

5.3 Performance in Reducing Fragmentation

Lastly, the performance of the proposed strategies in reducing the amount of fragmentation (virtual and external) in the mesh system will be evaluated. As described in our system model in Chapter 2, a request may make several attempts



(a) Frame sliding



(b) Adaptive scan, Leapfrog and Busy List

Figure 5.4: Effect of fragmentation in the mesh system under various strategies

in locating a free submesh in the mesh system before one is found. An attempt is unsuccessful when either there is not enough free nodes to satisfy the request or when a free submesh cannot be found due to fragmentation in the mesh system. Therefore the mean number of attempts per task that is unsuccessful due to fragmentation will be used as the performance metric to measure the amount of fragmentation in the mesh system. Simulation is carried out using a system load of 0.5 and the other parameters are the same as in the previous cases. Figure 5.4 shows the results obtained when the frame sliding, adaptive scan, leapfrog (first fit) and busy list strategies are used with the proposed strategies.

From the figure, it can be observed that the number of unsuccessful attempts for the frame sliding strategy is much higher than the rest of the strategies even when the proposed schemes are employed. This is due to the fact that the frame sliding strategy suffers from both virtual and external fragmentation while the other strategies only suffer from external fragmentation. Therefore the amount of fragmentation is much higher for the frame sliding strategy. The FFMB scheme is able to reduce the number of unsuccessful attempts due to fragmentation for all the three first-fit strategies. With an exemption of frame sliding strategy, the ODC-SC scheme improves the condition of fragmentation further. FFMB performs better than ODC-SC when used with the frame sliding strategy as it is able to reduce a larger amount of virtual fragmentation in addition to external fragmentation. Of the three schemes, ODC-FC obtains the best performance since it can reduce the largest amount of fragmentation with all the four allocation strategies used in the study.

Chapter 6

Conclusions

In this thesis, the problem of processor allocation is considered in mesh multi-computer systems. Firstly, a comprehensive survey of all the existing and useful strategies in the literature has been made. Then three novel strategies that could improve the performance of the first-fit strategies reported in the literature are proposed. One of the strengths in the design of the strategies is in fusing the task migration with the process of processor allocation to derive significant performance improvement. While there are related issues such as efficient search mechanisms and fragmentation in place, the strategies are shown to take care of these issues efficiently. While there is literature in analyzing the task migration and allocation independently, these are, in reality, mutually dependent events. This is the first time in the domain of processor allocation strategies that such an attempt in realizing the combined influence of task migration and allocation is undertaken.

Three efficient strategies that elicit this combined influence of task migration and allocation strategies are presented in this thesis. It has been shown that by using the strategies together with the existing strategies, the performance improvement is multi-fold. Clear recommendations have been made on the choice of the strategies to be used, especially when first-fit strategies are to be implemented. The impact of the strategies on one of the best-fit strategies is also shown. The recommendations also took into account the total amount of overheads involved in the process. Further, when the size of the mesh system is small, the strategies are an apt choice in maximizing the utilization of the system, as fragmentation is shown to be small with the design. Also, when one does not need migration owing to a small size network, the FFMB scheme becomes a natural choice. Thus, depending on the choice of the requirements, size of the network, amount of overheads, it has been shown that a judicious choice among the proposed strategies is able to render an improved performance. The rigorous simulation study which used practical workloads as reported in the literature, testified the entire workings of the strategies proposed.

The usefulness and importance of the proposed schemes is evident from the rigorous simulation study. The simulation results show that the proposed schemes indeed enhance the performance of the existing allocation strategies. When first-fit strategies are to be chosen over best-fit strategies due to overhead concerns, the proposed FFMB strategy can be employed to improve the performance of these first-fit strategies without incurring much additional overhead. The ODC-SC and ODC-FC strategies can be used to improve the first-fit strategies so that they can perform better than best-fit strategies that do not use task migration. Although the ODC-SC and ODC-FC strategies can also be used with best-fit strategies, they tend to introduce a significant amount of overhead on top of the large overhead that is already in place with best-fit strategies. In addition, the study shows that both first-fit and best-fit strategies perform equally well under ODC-SC and ODC-FC. Therefore, if ODC-SC or ODC-FC are to be employed, first-fit strategies can be used as they incur a smaller amount of overhead. Lastly, the study also shows that although ODC-FC incurs higher implementation overheads than ODC-SC, it performs better. Therefore, ODC-FC can be employed in situations where the tasks have shorter deadlines. In case where the tasks have soft deadlines, ODC-SC can be used to reduce the amount of overheads incurred during task migration.

There are a few plausible extensions to the study presented in this thesis. As it was clearly shown in the study that ODC-SC or ODC-FC indeed improves performance under task migration, it would be of natural interest to redesign best-fit strategies incurring minimum overheads so that the proposed strategies could be applicable to derive better performance. Secondly, although the study in this thesis is targeted at mesh architecture, which is commonly and widely used, performance study of the proposed strategies to arbitrary topologies would be interesting.

Bibliography

[1] N. -C. Wang and T. -S. Chen, ”Task Migration in All-Port Wormhole-Routed 2D Mesh Multicomputers,” *Proc. 7th Int'l Symp. on Parallel Architectures, Algorithms and Networks*, pp. 123-128, 2004.

[2] G. -J. Yu, C. -Y. Chang, and T. -S. Chen, ”Task Migration in n-dimensional Wormhole-Routed Mesh Multicomputers,” *Journal of Systems Architecture*, vol. 50, issue 4, pp. 177-192, Mar. 2004.

[3] F. Wu, C. C. Hsu, and L. P. Chou, ”Processor Allocation in the Mesh Multiprocessors Using the Leapfrog Method,” *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 3, pp. 276-289, Mar. 2003.

[4] N. -F. Tzeng and H. -L. Chen, "Fast Compaction in Hypercubes," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 1, pp. 50-56, Jan. 1998.

[5] V. Lo, K. J. Windisch, W. Liu, and B. Nitzberg, "Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 7, pp. 712-726, July. 1997.

[6] H. -L. Chen and N. -F. Tzeng, "On-Line Task Migration in Hypercubes Through Double Disjoint Paths," *IEEE Trans. Computers*, vol. 46, no. 3, pp. 379-384, Mar. 1997.

[7] L. M. Ni and P. K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, vol. 26, no. 2, pp. 62-76, Feb 1993.

[8] J. Ding and L. N. Bhuyan, "An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems," *Proc. 1993 Int'l Conf. Parallel Processing*, vol. 2, pp. 193-200, 1993.

[9] D. D. Sharma and D. K. Pradhan, "A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers," *IEEE Symp. on Parallel and Distributed Processing*, pp. 682-689, 1993.

[10] Intel Corp., *Paragon Network Queuing System Manual*, 1993.

[11] M. Noakes, D. Wallach, and W. Dally, ”The J-Machine Multi-Computer: An Architectural Evaluation,” *Proc. Int'l Symp. Computer Architecture*, pp. 224-235, 1993.

[12] Y. Zhu, ”Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers,” *J. Parallel and Distributed Computing*, vol. 16, pp. 328-337, Dec. 1992.

[13] G. -I. Chen and T. -H. Lai, ”Constructing Parallel Paths Between Two Subcubes,” *IEEE Trans. Computers*, vol. 41, no. 1, pp. 118-123, Jan. 1992.

[14] Intel Corp., *A Touchstone DELTA System Description*, 1991.

[15] P. -J. Chuang and N. -F. Tzeng, ”An Efficient Submesh Allocation Strategy for Mesh Computer Systems,” *11th Int'l Conf. on Distributed Computing Systems*, pp. 256-263, 1991.

[16] K. Li and K. -H. Cheng, ”A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System,” *J. Parallel and Distributed Computing*, vol. 12, pp. 79-83, 1991.

[17] M. -S. Chen and K. G. Shin, ”Subcube Allocation and Task Migration in Hypercube Multiprocessors,” *IEEE Trans. Computers*, vol. 39, no. 9, pp 1146-1155, Sep. 1990.

[18] R. Alverson et al., "The Tera Computer System," *Proc. Fourth ACM Int'l Conf. Supercomputing*, pp. 1-6, 1990.

[19] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421-431, 1977.

Appendix A

Implementation Details

The implementation details on the simulation study will be presented in brief. The performance of the proposed algorithms is studied using event-driven simulation. The events in the simulation are allocation, deallocation and reallocation. A set of allocation events are generated before the actual simulation is run and these events are generated following some probability distributions. These allocation events represent the arrival of the requests for a free submesh during the duration of the simulation. Each allocation event is a structure that contains information such as the arrival time of the allocation event, the dimensions of the submesh that is required and the processing time required by the task to complete its computation. These allocation events are inserted into an allocation queue in increasing order of their arrival time. There is also a deallocation queue and a reallocation queue. The deallocation queue contains deallocation events in increasing order of their arrival time. Similarly, the reallocation queue contains reallocation events in increasing order of their arrival time. These two queues will be empty at the start of the simulation.

A variable $sysTime$ is used in the program to track the overall system time. During the simulation, the program compares the first event in each of the three queues and chooses the event with the earliest arrival time as the next event. When more than one of the events have the same arrival time, the program first chooses the deallocation event, followed by the reallocation event and finally the allocation event. The $sysTime$ variable is then updated accordingly.

If the next event is an allocation event, it is removed from the allocation queue and the program will try to locate for a free submesh in the mesh system. If a free submesh can be successfully located, the system updates the structures such as R-arrays [3] and busy-arrays [12] and the total time taken to search for the free submesh as well as to update the structures are added to the $sysTime$ variable. The program then generates a deallocation event with an arrival time that is equal to the sum of the current system time and the processing time required by the task and inserts it into the deallocation queue. On the other hand, if no free submesh is found, only the time taken to search for the free submesh is added to the $sysTime$ variable. The program then generates a reallocation event with an arrival time that is equal to the arrival time of the next deallocation event. This ensures that all the reallocation events will be processed every time after a deallocation event has been processed.

If the next event is a deallocation event, it is first removed from the deallocation queue. The program then updates the structures so that the submesh that is allocated to this task is freed. The time taken to update the structures is added to the $sysTime$ variable.

If the next event is a reallocation event, it is first removed from the reallocation queue. The program then uses the same procedure as the allocation event to try to search for a free submesh. If a free submesh is found, the $sysTime$ variable is updated accordingly and a deallocation event is generated and inserted into the deallocation queue. Otherwise a new reallocation event is generated with a new arrival time and inserted back into the reallocation queue.

The simulation ends when all the three event queues are empty.