Submesh Allocation in a Multitasking Distributed Shared Memory Multiprocessor

T. Liu, W-K. Huang, F. Lombardi and L. N. Bhuyan

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Tel. (409)845-9640
E-mail: {bhuyan, lombardi}@cs.tamu.edu

ABSTRACT

This paper proposes an efficient submesh allocation strategy based on a free list and compares the performance of the proposed strategy with all the existing strategies. The novel feature of the free list in the proposed scheme is its overlapped nature, i.e. free submeshes in the list may overlap each other. The trade-offs between an overlapping and non-overlapping free list for a mesh system are discussed. Allocation and deallocation algorithms are presented in this paper, both with a low complexity in $O(N_F^2)$, where N_F is the number of free submeshes in the free list. Extensive simulation results show that our approach is superior to all the previous strategies, yielding a lower waiting time and a substantial improvement in allocation time.

Index Term: Submesh Allocation, Deallocation, Mesh, Free List, Busy List, Overlapping.

[0]This research is supported by a grant from the Texas ATP.

1 Introduction

The two-dimensional (2D) mesh topology has become popular for distributed memory multiprocessors (DMMs) because of its simplicity, regularity, and suitability for VLSI implementation. DMMs can be based either on message passing or shared memory paradigms. There are quite a few 2D mesh-based commercial or prototype systems built recently. Typical examples are the Intel Paragon [1] for a message passing system and the Stanford DASH [2] for a shared memory system. The Cray T3D [3] is also an example of a distributed shared memory multiprocessor that employs a 3-D mesh as an interconnection network. In a multitasking DMM (both shared and distributed memory), the operating system not only specifies which job will be executed next, but it also must specify which processors would execute the job.

In the current implementations, while the processor allocation has been an integral part of the message passing operating system [1, 4], it has remained an optional feature in the distributed shared memory implementation [5]. It is well known that inefficient allocation of processors and data in a distributed shared memory system causes increased memory access overhead due to the differences in local and remote memory access times. As an example, a remote access in the Butterfly GP1000 is 10 times slower than the local memory access [5]. In a multitasking environment, the intertask interferences in the network produce large queuing delays and degrade the performance [5, 6]. The same is true for multiprocessors like DASH [2] or Cray T3D [3]. As a result of this, an attempt has already been made to partition shared memory multiprocessors based on multistage interconnection networks (MIN) for efficient multitasking [6]. An architecture similar to Butterfly multiprocessor was considered for the study. In this paper, we develop a processor allocation scheme for 2D-mesh DMM's and show that the new scheme is much superior to the existing schemes [7]-[11]. The proposed scheme is applicable to both shared memory and message passing systems.

In a 2D mesh connected system, incoming jobs are allocated to submeshes of different sizes. The sizes of submeshes range from as few as one node to as many as all the nodes in the entire mesh. As the number of processors in mesh-based systems grows, designing efficient submesh allocation strategies becomes increasingly difficult, but interesting. We assume that there is always a host processor in the mesh connected system. The operating system in this host processor includes a task dispatcher that is responsible for allocating free submeshes to the incoming tasks. An efficient task dispatcher should use a strategy which can quickly find a free submesh whose size is just sufficient to meet the requirements of any incoming tasks.

A number of submesh allocation strategies have been proposed in the last few years [7]-[11]. Li and Cheng have proposed a Buddy based strategy [7] that is applicable only to square mesh systems whose side lengths must be a power of 2. Also, it always allocates a square submesh whose side is a power of 2 to an incoming job, regardless of the real needs of the incoming tasks. Thus, the Buddy strategy suffers from a large internal fragmentation. To solve this problem, Chuang and Tzeng have proposed a frame sliding (FS) strategy for meshes with arbitrary length and width [8]. The FS strategy totally eliminates internal fragmentation and also outperforms the Buddy strategy in terms of allocation time and processor utilizations. Zhu has proposed a First-Fit and a Best-Fit strategies which have better performance than FS strategy [9]. Ding and Bhuyan have proposed an adaptive scan (AS) strategy which has still smaller external fragmentation and better performance than the FS strategy [10]. Sharma and Pradhan have proposed a strategy which fully uses the "deallocation list" and outperforms all the previous schemes [11]. All these existing strategies are briefly discussed in section 3.

The proposed approach is based on a free list. In our strategy a free list, similar to the one in hypercube [12], is maintained. But, because of the mesh topology, creating the list is difficult, complex, and interesting. The free list used in the proposed strategy is an overlapped one, i.e. the free processors may belong to multiple free submeshes in the list. And it is a single list sorted by the increasing order of the submesh sidelength. We show that the complexity of our strategy is not higher than the existing strategies. The simulation results also show that our strategy performs better on allocation time and average waiting delay compared to the existing strategies.

This paper is organized as follows. In the following section we introduce notations and definitions used by the submesh allocation strategies. In section 3, we give a brief discussion of the existing submesh allocation strategies. Our approach is presented in section 4 which includes the proposed algorithms and detailed examples. Section 5 provides complexity analysis and simulation results to compare with other strategies. Finally, conclusions are given in section 6.

2 Preliminaries

A two-dimensional mesh system $M(l, w)$ consists of $l \times w$ nodes structured as a $l \times w$ two-dimensional rectangular grid. A node or processor at column a and row b is addressed as (a, b). A non-boundary node (a, b) is connected to its four neighbors: $(a-1, b)$, $(a+1, b)$, $(a, b-1)$ and $(a, b+1)$, while boundary nodes have two or three neighbors depending on their positions on the corners of the mesh system. A submesh $S(l', w')$ in $M(l, w)$, denoted by $[x, y, l', w']$, is made of $l' \times w'$ nodes in the rectangular region defined by the lower left corner (x, y) to the upper right corner $(x + l' - 1, y + w' - 1)$ in $M(l, w)$.

Definition 1. A *free submesh* is a submesh in which all nodes are not occupied by allocated jobs. A *busy submesh* is a submesh in which all of its nodes are allocated to a job.

Definition 2. A *free list* is an ordered list of all free submeshes. The free list used in this paper is sorted by increasing order of shorter edge of submeshes. For those with the same shorter edges, it is ordered by increasing order of longer edge. A free node in $M(l, w)$ may belong to two or more free submeshes in the free list, i.e. submeshes in the free list may be overlapped. The length of the list is denoted as N_F.

Definition 3. A *busy list* is a list of all busy submeshes of the allocated jobs. The length of the list is denoted as N_B.

Definition 4. *The boundary value* of a free submesh is the number of nodes neighboring both with busy nodes of other jobs and with the border of the mesh system.

Definition 5. *Fragmentation* is said to occur when an incoming task can not be allocated to a free submesh even though the number of free processors in the system is more than that required by the incoming task. *Internal fragmentation* happens when more processors than what is required are allocated to a task. *External fragmentation* occurs when there is no free submesh large enough to be allocated to the incoming task even with a sufficient number of free processors. *Virtual fragmentation* occurs when there exists a large enough free submesh, but the system fails to recognize it.

3 Previous Strategies

The *two-dimensional buddy* strategy (2DB) was proposed in [7]. The 2DB is based on the traditional buddy strategy in memory management [13]. However, this strategy requires the mesh system as well as the incoming requests to be square with sides being a power of 2. Thus, it is not applicable for general mesh systems such as the Intel Touchstone Delta [14] and the Intel Paragon XP/S [1]. Additionally, this strategy can cause excessive internal fragmentation because an incoming task has to be allocated a square submesh whose sidelength is a power of 2.

The *Frame Sliding* (FS) strategy [8] was proposed to overcome the above limitation of the 2DB strategy. It is applicable to any mesh system and causes no internal fragmentation. However, the FS strategy has the drawback of virtual fragmentation due to its incomplete submesh recognition capability. It fails to recognize free submeshes, because it searches for free submeshes in only certain horizontal and vertical strides.

In [9], two strategies were proposed as *First-Fit* (FF) and *Best-Fit* (BF). Both the FF and BF strategies are applicable to mesh systems of any size and can allocate a submesh of the exact size to an incoming task, thus eliminating the internal fragmentation experienced by the 2DB strategy. Also, they avoid virtual fragmentation for some scenarios associated with the FS strategy. However, both FF and BF suffer from virtual fragmentation in many cases due to their incomplete submesh recognition capability.

The *Adaptive Scan* (AS) strategy [10] improved the FS strategy and eliminated the virtual fragmentation associated with the FS strategy. The AS uses scanning instead of sliding with fixed stride to search for the the candidate free submeshes. Partitioning processors into coverage and reject sets were introduced to reduce the search space. Rotating the orientations of incoming tasks was first used in the free submesh searching. This operation has been also used by later approaches [11]. The AS strategy has better performance than the FS and 2DB strategies. However, the improvement is limited by its first-fit nature and thus is not significant.

The *Busy-List* (BL) strategy was proposed recently [11]. A busy list is maintained in the system to keep records of tasks which are running in the system. By using the BL, the search of free submesh for a new incoming task is based on the busy list as opposed to all other strategies which are based on a two-dimensional array with each bit corresponding to a processor. This makes BL to perform more efficiently than the others because a set of processors' occupancy is obtained from each operation on the busy list while only one processor's usage status is know by each operation on the array. However, its allocation algorithm has a time complexity of $O(N_F^3)$, where N_F is the length of the busy list, this is compared to $O(N)$ for the previous strategies, where N is the number of processors. This is due to its duplicated submesh recognition procedure.

Our approach is based on maintaining a free list. An existing free list strategy for the hypercube [12] is briefly summarized here. The free list consists of $(n+1)$ lists of available subcubes, with one list per dimension. A k-cube is allocated to an incoming job by searching the free list of dimension k. If the list is empty, then a higher dimension subcube is decomposed and allocated. For the deallocation, the released subcube is merged with all possible adjacent cubes to form the largest possible disjoint subcube(s). The list is updated accordingly. The time complexities of allocation and deallocation are $O(n)$ and $O(n2^n)$, respectively. As compared with the other subcube allocation schemes, the free list strategy has an advantage of lower complexity in the allocation process. The process of deallocation, however, is much complex than its allocation process. This is mainly due to the characteristics of the hypercube architecture.

4 Proposed Approach

Submesh allocation consists of finding a free submesh for the incoming job. Therefore, a good submesh allocation strategy should have the following features: small allocation and deallocation times to make the system load low, small fragmentation and thus, high system utilization. The proposed strategy maintains a list of all free submeshes, not covered by any other submesh. There may be an overlap among free submeshes in the list; this is required for reducing the allocation time compared with the approach of using a busy list only. Our approach consists of an *allocation algorithm* and a *deallocation algorithm*, plus an *insertion algorithm* which is used both in the allocation and deallocation algorithms.

The *free list* is the main data structure maintained by our strategy. It is kept by increasing order of the shorter edge of submeshes and for those with same shorter edges, by an increasing order of longer edges. For same shorter and longer edges, the order is arbitrary. This ordering of the free list is for a faster search in the allocation process. There are two choices to keep the free submeshes in the free list, one is non-overlapping among free submeshes as the free list strategy in hypercubes [12], the other is overlapping which we adopted. Unlike the free list in hypercubes, it is much more complex because of the partitioning process of a mesh. The advantage of non-overlapping as used in hypercubes is that after a free subcube is selected for allocation, only the fragments from this free subcube are needed to be inserted in the free list, no other free subcube is affected. But in a mesh system, the combination of two or more free submeshes must be considered for allocation when none of the free submeshes in the list is large enough for the incoming task. This is a very difficult task to consider, else virtual fragmentation will occur frequently and degrade the system performance. By using overlapping, it is possible to find an existing free submesh very efficiently, however, some submeshes in the free list have to be adjusted after a free submesh is allocated. Because of this characteristic, we have developed efficient allocation and deallocation algorithms using an overlapped free submesh list.

The basic idea of our allocation approach is to search the free list for the first free submesh which is equal to or larger than the size required by the new job, and then to adjust the free list affected by this allocation. The proposed allocation algorithm, which is presented in Figure 1, implements a linear search starting from the head of the list to its tail until the first free submesh large enough to hold the new job, is found. Then, if the free submesh is larger than the new job, a decision should be made concerning where to put the new job in the free submesh. The allocation to one of the four corners will result in a smaller fragmentation to the system. The decision is then reduced to choosing one among eight positions: four corners, vertically and horizontally. The maximum of the boundary values is used for choosing the corner.

As stated previously, submeshes in the free list may be overlapping. Therefore, after allocating the new job, some free submeshes in the free list may be overlapped with the newly allocated job and thus it needs an adjustment. To accomplish this, first, the whole free list is checked again and all submeshes overlapping with the located job are found. Then, each found free submesh is split and the free fragments are re-inserted into the free list.

The deallocation process of a job produces some free submeshes and expands some free submeshes which are already in the list. The proposed algorithm, which is shown in Figure 2, consists of maintaining a free list of all possible free submeshes. First, we set the departing job as a free submesh and expand it towards the four directions: up, down, left and right. Since the expansion may get different submeshes by starting from different

Algorithm Allocation
begin
Step 1. Locate the smallest free submesh which can hold the new job.
for each submesh F_i in the free list
if F_i can hold the new job then
$F = F_i$;
delete F_i from the free list;
break;
if no free submesh can fit the new job
return Fail;
Step 2. Calculate the boundary values at each corner of F.
for each job in the busy list and on the borders of the mesh system
if it borders F
increase the corner values up to the corresponding corners;
allocate the new job at the corner with the maximum boundary value;
there are at most two free fragments in F after allocating the new job,
put them (if they exist) into the stack;
Step 3. Check and adjust all free submeshes which are overlapped with the allocated new job
for each submesh in the free list
if it overlaps with the allocated new job
delete it from the free list;
put the fragments into the stack;
Step 4. Insert free submeshes from the stack into the free list.
for each submesh in the stack
insert it into the free list;
end

Figure 1: The Allocation Algorithm

```
Algorithm Deallocation
begin
  Step 1. Form four or less free submeshes based on the newly released job.
    for each submesh in the free list
        update west, east, north, south;
    for each submesh in the free list
        update west-up, west-down, east-up, east-down,
           north-left, north-right, south-left, south-right;
    Free_submesh1:
        [west:min(north-right, south-right)][west-down:west-up];
    Free_submesh2, if not equal to Free_submesh1:
        [max(north-left, south-left):east][east-down:east-up];
    Free_submesh3, if not equal to Free_submesh1,2;
        [south-left:south-right][south:min(west-up, east-up)];
    Free_submesh4, if not equal to Free_submesh1,2,3;
        [north-left:north-right][max(west-down, east-down):north];
    put the new free submeshes in the stack;
  Step 2. Adjust the submeshes in the free list corresponding to newly formed ones.
    For each new free submesh NF_i from Step 1. (at most four)
        For each submesh F_i in the free list
            if F_i is covered by NF_i
              delete F_i from the free list;
            else if any edge of F_i is covered by or bordering with NF_i
              extend F_i to the other end of NF_i;
              delete F_i from the free list and insert it into the stack;
  Step 3. Insert the free submeshes in the stack into the free list.
    For each submesh in the stack
        insert it into the free list;
end
```

Figure 2: The Deallocation Algorithm

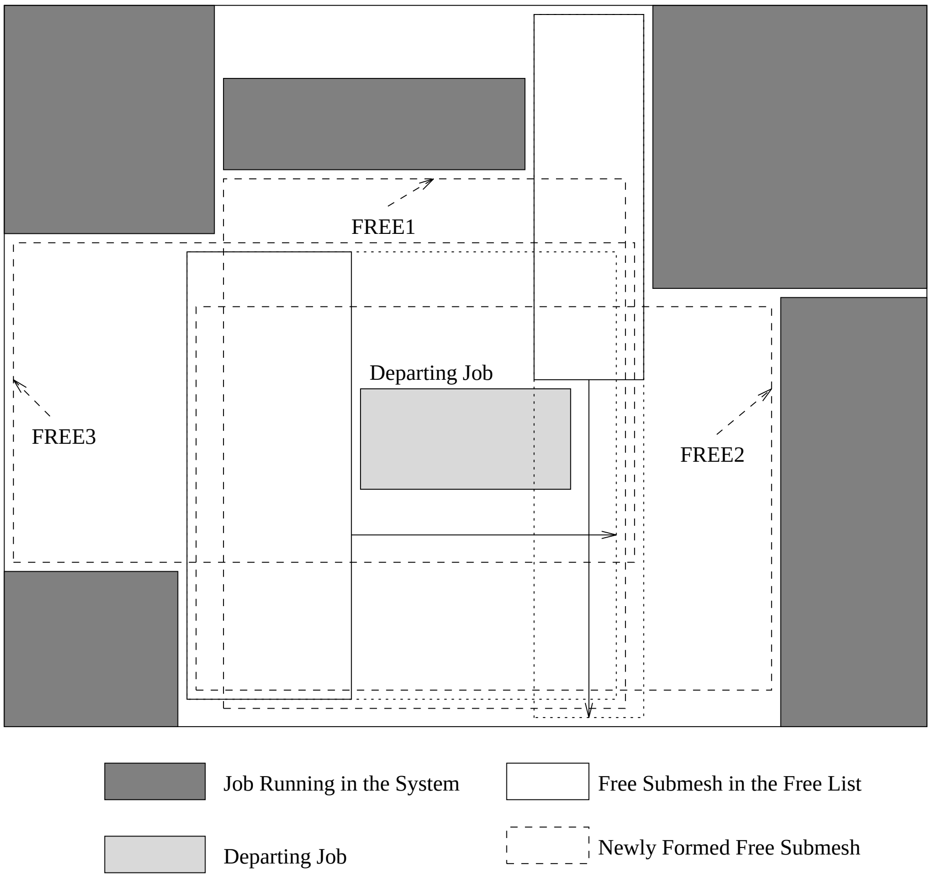

Figure 3: Description of the deallocation process.

directions, all four directions are used. This is explained in Figure 3. The rectangles with dotted lines are formed through the expansion of the departing job, FREE 1 is from starting on top, FREE2 is from down which is the same as from right, FREE3 is from left. In general, this will result in at most four different free submeshes. These free submeshes are not expandable and will be inserted into the free list later. Similar to the free list adjustment in the allocation procedure, the free list also needs to be adjusted at this point. If a free submesh is covered by any one of the newly formed free submeshes, then it should be deleted from the list, else if it was blocked from an expansion of one side only by the departing job, then it should be expanded and re-inserted into the free list. In Figure 3, the rectangles with solid lines are examples of such free submeshes in the free list. By expanding through the direction against the departing job, which is indicated as the solid line with an arrow in Figure 3, an adjusted free submesh is ready to be re-inserted into the free list.

```
Algorithm Insertion
begin
    Let NF be the free submesh to be inserted;
    if the free list is empty then
        put NF in it;
        return;
    while not at the end of the free list
        let F_i be the current entry;
        if NF's shorter edge < F_i's shorter edge
          or NF's shorter edge = F_i's shorter edge
            and NF's longer edge < F_i's longer edge
        then
            for each submesh F_j of the remaining submeshes of the free list
              if NF is covered by F_j
                  discard NF and return;
            put NF before F_i;
            return;
        else
            if F_i is covered by NF
              delete F_i from the free list;
    end while
    put NF at the end of the free list;
end
```

Figure 4: The Insertion Algorithm

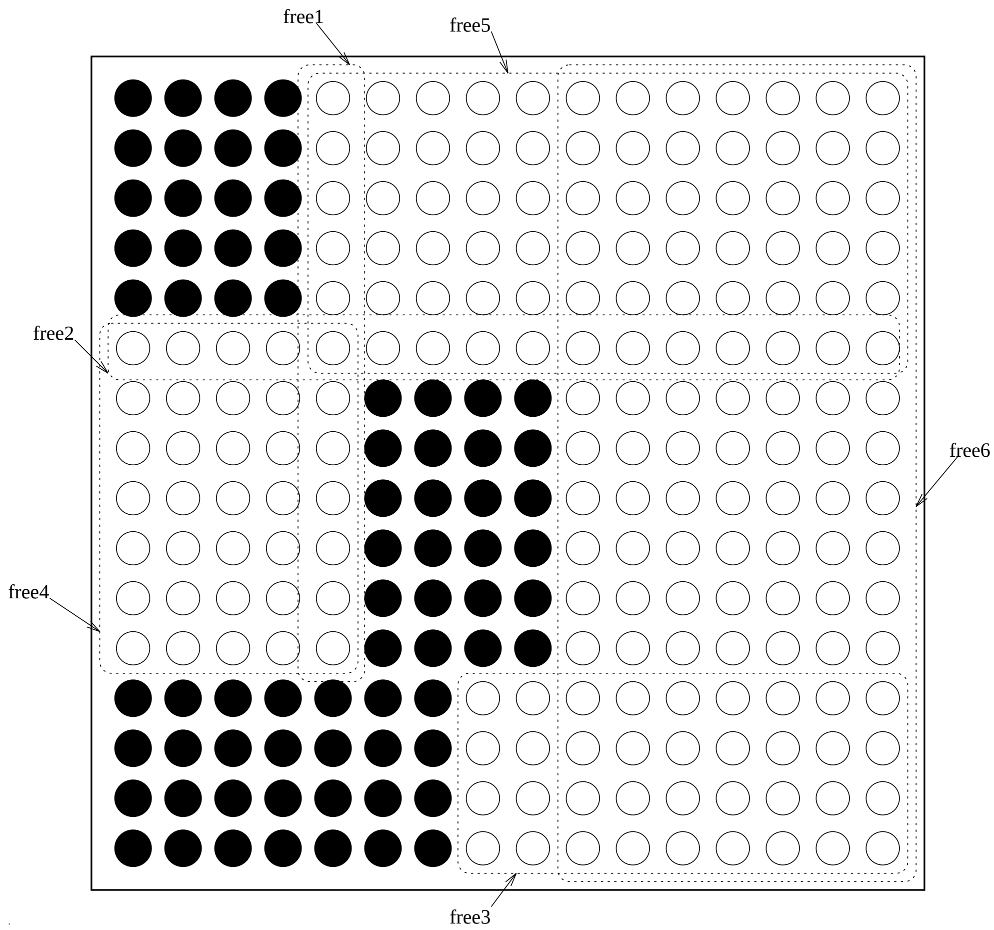

Figure 5: Example of a M(16,16).

The Insertion algorithm, which is presented in Figure 4, is used in the Allocation and Deallocation algorithms to insert a submesh into the free list. The insertion keeps the list by increasing order of shorter edges of submeshes; for those with same shorter edges, the order is by increasing order of longer edges. Also, it is discarded if it is covered by any submesh in the list. This algorithm also deletes all submeshes that are covered by the new submesh. The following two examples will describe these operations in detail.

Example 1. This example illustrates the execution of the Allocation and Insertion Algorithms. Figure 5 gives the current status of a 16x16 mesh system. Three jobs are allocated: Job1 with size 7x4 at (0,0), Job2 with size 4x5 at (0,11), and Job3 with size 4x6 at (5,4). In the free list, there are six submeshes, as shown in Figure 5; they are in the order of free1, free2, free3, free4, free5, free6. Assume that the incoming new job requires a submesh of size 3x4. The allocation algorithm begins by searching the free list from its head free1. The

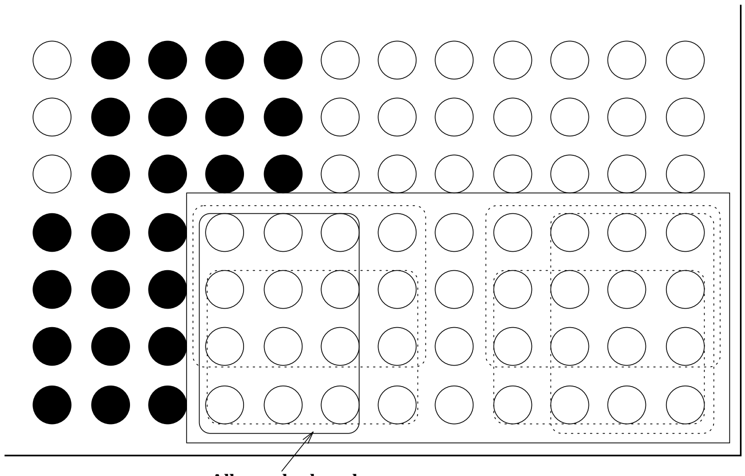

Figure 6: Example of Allocation.

first free submesh that can be allocated to the new job, is free3 with size 9x4 at (7,0). Next, the new job is analyzed in free3 at one of the four corners vertically and horizontally with maximum of corner boundary value. This is detailed in Figure 6. There are six selections in this case, because the length of the new job is 4 and this is exactly the height of free3, and only two possible locations vertically. The position with solid rectangle at the left side vertically corresponds to the maximum boundary value among all six possible selections, and thus it is the final position for the incoming job. After allocating the new job, there is one fragment free submesh 6x4 at (10,0) in free3. This is put in the stack and all submesh in the free list are checked for adjustment. Free6 overlaps with the allocated new job. Free6 is deleted from the free list and two fragments 6x16 at (10,0) and 7x12 at (9,4) are put into the stack. No other submesh needs adjustment. Last, all submeshes in the stack are inserted into the free list by the Insertion Algorithm. In the insertion, 6x16 at (10,0) and 7x12 at (9,4) are inserted into the list, while 6x4 at (10,0) is covered by 6x16 at (10,0) and is discarded. Now, there are four jobs in the system, and six submeshes in the free list.

Example 2. This example shows the deallocation procedure. We also begin from the state shown in Figure 5. Assume that Job3 is departing from the system. The job list is searched twice to get west=0, east=15, north=15, south=4, west-up=10, west-down=4, east-up=15, east-down=4, north-left=4, north-right=15, south-left=0, south-right=15. As shown in Figure 7, by Free_Submesh1, we have FREE1: 16x7 at (0,4), by Free_Submesh2, we have FREE2: 12x12 at (4,4), by Free_Submesh3, same as FREE2, by Free_Submesh4, same as FREE1. i.e. FREE1 and FREE2 are two new free submeshes in the stack from

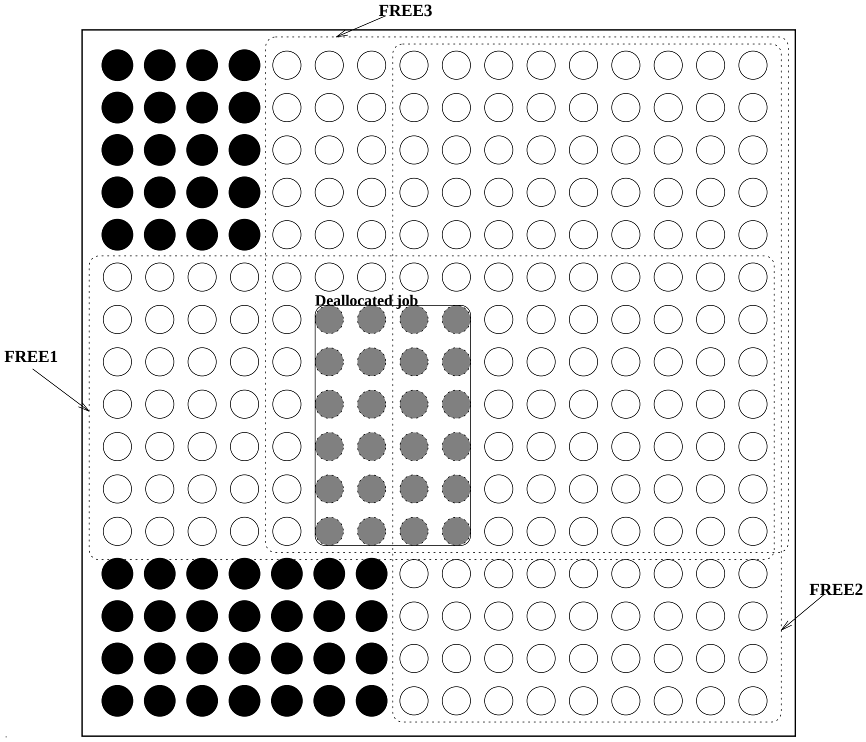

Figure 7: Example of Deallocation.

Step 1. Next the free list is adjusted, free2 and free4 are covered by FREE1, free1 and free5 are covered by FREE2, thus free1, free2, free4 and free5 are deleted from the free list. free3 is extended through FREE2 to north, and is changed to FREE3: 9x16 at (7,0). FREE3 is put into the stack and free3 is deleted from the free list. After this step, only free6 is in the free list, and FREE1, FREE2 and FREE3 are in the stack. Last, submeshes in the stack are inserted into the free list. In the insertion of FREE3, free6 in the free list is covered by FREE3 and is discarded from the list. After Job3 has left, FREE1, FREE2, and FREE3 are in the free list.

5 Complexity Analysis and Simulation Results

In this section, we present the time complexity of the proposed algorithms and their average performance through extensive simulation. In both cases, we compare our results with the existing results [7, 8, 9, 10, 11] to show the superiority of the new algorithms.

5.1 Complexity Analysis of the Proposed Algorithms

Since it has been already shown that the busy list approach [11] has the least time complexity compared to all the previous algorithms [11], we will compare our results with [11]. For the complexity analysis of the proposed algorithms, we first give a lemma showing that the lengths of the busy list (N_B) and the free list (N_F) are in the same order. As our approach is based on the free list strategy as opposed to the busy list in [11], the lemma makes this comparison at the same level.

Lemma 1. N_B and N_F are in the same order.

Proof. Let $s_1, s_2, s_3, ..., s_{N_B}$ be the N_B jobs currently in the system, and f_1, f_2, f_3, ..., f_{N_F} be the N_F free submeshes in the free list. Let us consider a reverse situation, i.e. if currently $[i,j]$ is busy, we set it to free, while if $[i,j]$ is free, we set it to busy. The number of free submeshes after the reverse operation is equal to or less than N_B, because two or more previous busy submeshes may be merged into one free submesh after the reverse operation. Similarly, by putting the overlapped free nodes into only one busy submesh in the reverse operation, the number of busy submeshes is equal to or less than N_F, because all nodes in some free submeshes may be overlapped and put into other submeshes by the reverse operation. Dynamically, a node changes from busy/free to free/busy, then N_B and N_F do not have fixed values, in some case N_B/N_F equals to N'_F/N'_B of some other case. So, we say they are in the same order. This conclusion is applicable to both the worst case and the average case. $\square$

First, we do the analysis of the Insertion Algorithm, as it is used both in the Allocation and Deallocation Algorithms. The while-loop requires N_F iterations in the worst case, each of these iterations (except at most once) takes constant time. The for-loop runs at most once when the corresponding if-condition is valid. This for-loop is in the worst case $O(N_F)$. The while-loop and the for-loop together have a time complexity of $O(N_F)$. So, the time complexity of the Insertion Algorithm is $O(N_F)$.

Consider now the complexity of the Allocation Algorithm. Step 1 takes $O(N_F)$ in the worst case. In Step 2, the for-loop takes N_B, the remaining part executes in constant time. Since we know that N_B and N_F are in the same order, then Step 2 takes $O(N_F)$. Step 3 takes N_F. In Step 4, each iteration executes an insertion and it takes $O(N_F)$ (as from above). From Step 2 and Step 3, there are in the worst case $2(N_F+1)$ free submeshes in the stack, thus Step 4 takes $O(N_F^2)$ in the worst case. Overall, the Allocation Algorithm

Table 1: Complexity of Submesh Allocation Strategies.

Strategy	*2DB[7]*	*FS[8]*	*FF[9]*	*BF[9]*	*AS[10]*	*BL[11]*	*FL*
Allocation	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N_B^3)$	$O(N_F^2)$
Deallocation	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	O(1)	$O(N_F^2)$
Total Complexity	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N_B^3)$	$O(N_F^2)$

takes $O(N_F^2)$ in the worst case.

Finally, the analysis for the Deallcation Algorithm is as follows. In Step 1, the two for-loops take $O(N_F)$ each, followed by the generation of at most four free submeshes (this takes constant time). So, Step 1 takes $O(N_F)$. Step 2 takes $4N_F$. Step 3, (as in the analysis of Step 4 of the Allocation Algorithm,) takes $O(N_F^2)$ in the worst case. Overall, the time complexity of the Deallocation Algorithm is $O(N_F^2)$ in the worst case.

Thus, the time complexity of our approach, including allocation and deallocation, is $O(N_F^2)$ in the worst case. As the time complexity of BL strategy is $O(N_B^3)$, and Lemma 1 has shown that $O(N_F)$ and $O(N_B)$ are in the same order, then our new approach has a lower complexity than BL. Table 1 shows the comparison of the time complexity of all the submesh allocation strategies. 2DB, FS, FF, BF, and AS strategies [7, 8, 9, 10] have a time complexity of $O(N)$, where N is the number of processors in the mesh system. In the extreme case, each task utilizes one processor and there are N tasks running in the system, then $N_F = N$. However, in real applications each task requires a set of processors, otherwise, there is no need to use multiprocessors in the first case, and not all the processors are busy. Therefore, N_F is much smaller than N. Although we don't have quantitative comparison between $O(N_F^2)$ and N for the average case, the simulation results below will show that our strategy takes much less time to run compared to those of 2DB, FS, AS, and the BL approaches.

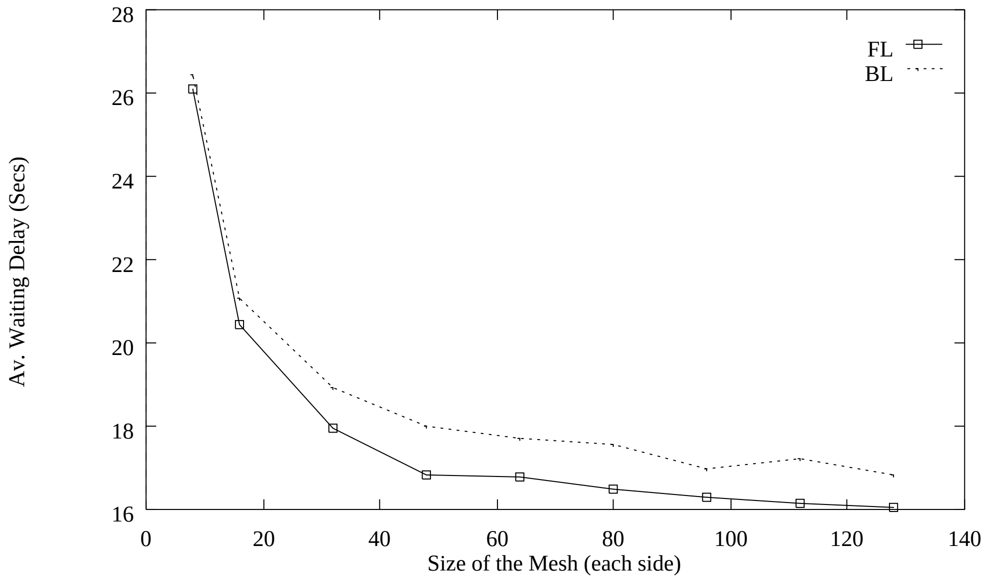

Figure 8: Submesh size for a Pseudo-Normal Distribution.

5.2 Simulation Results

Simulations were performed for comparing the performance of the proposed approach with previous approaches. Since the BL strategy [11] has better performance than other previous strategies, most comparison here is with the BL. The size of the mesh system we use in our simulation varies from 8x8 to 128x128, the job interarrival time is 6 to 12 seconds, and the service time is fixed at 10 seconds per job. For simplicity, without losing the generality, equal length and width are used on the mesh system like in others [10, 11].

For comparison consistency, we use the same simulation model as in [10, 11]. The First-Come-First-Serve (FCFS) policy is used for the job dispatcher. In the simulation, the new job is generated only until its previous one has been allocated successfully. The sizes (length and width) of incoming jobs are generated under three different distributions: uniform, pseudo-normal and linear on the size of the mesh system; the length and width of each incoming job are generated independently.

First, we run the simulation by fixing the interarrival time rate at 7 seconds and service time rate at 10 seconds, but varying the mesh size from 8x8 to 128x128. The performance is measured in terms of the average waiting delay for a job before it is allocated a submesh. Figures 8 and 9 show the average waiting delay of BL [11] and FL under a Pseudo-Normal and a Uniform Distribution respectively. Figure 10 is for AS [10], BL, and and FL ap-

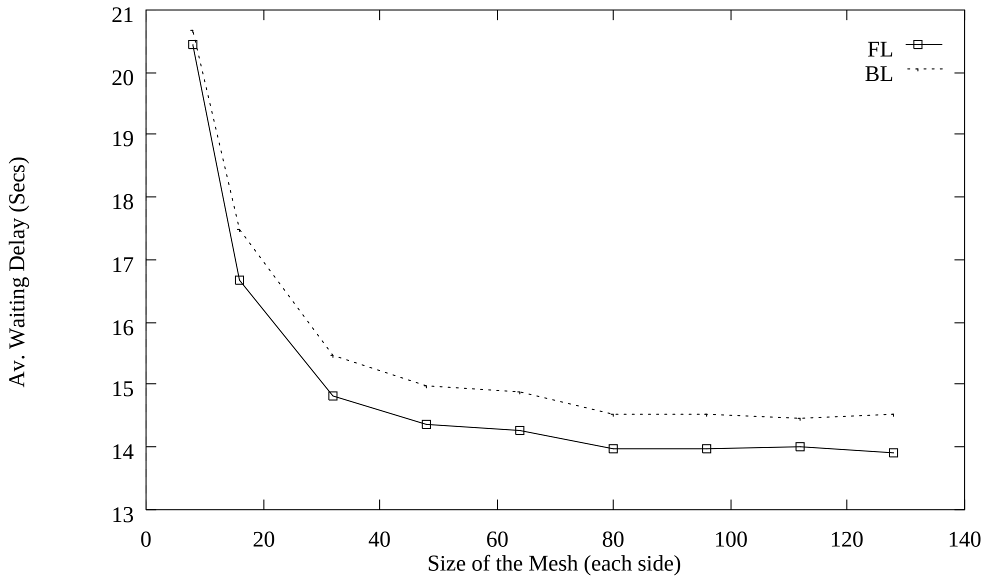

Figure 9: Submesh size for an Uniform Distribution.

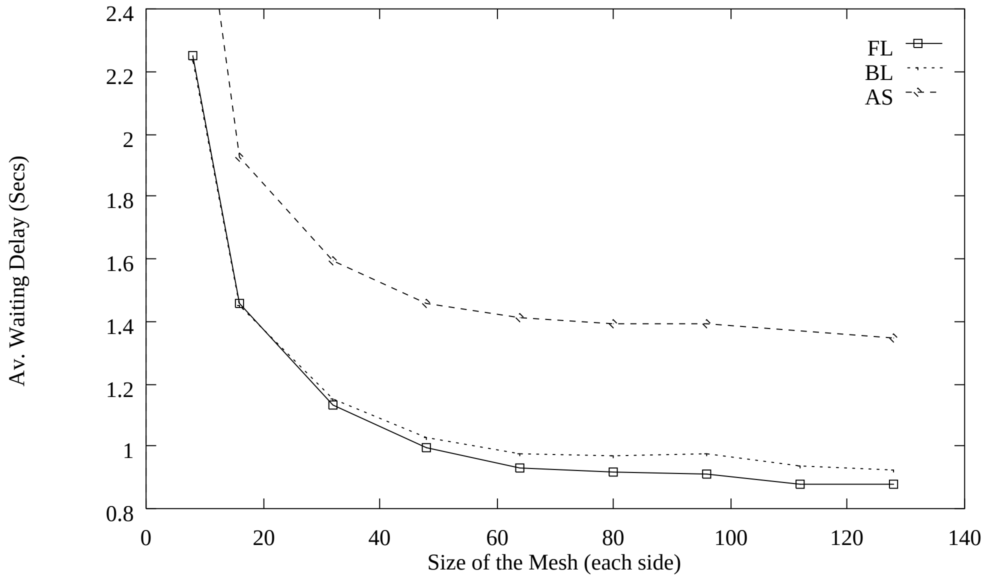

Figure 10: Submesh size for a Linear Distribution.

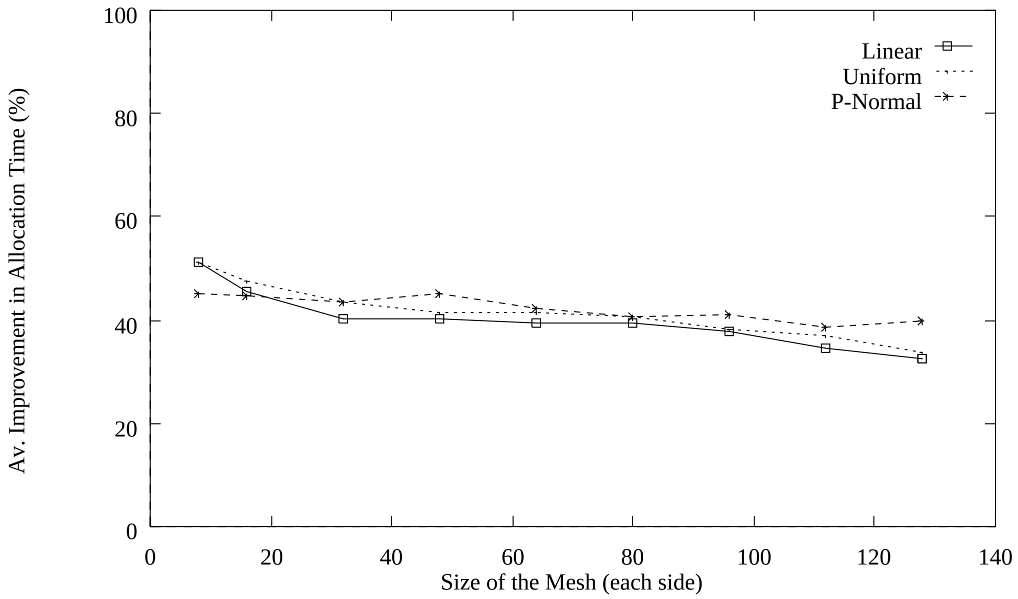

Figure 11: Submesh size for the Three Distributions.

proaches under the Linear Distribution, We can see that the performance by our approach is significantly better when the mesh size is larger than 16x16.

We measure the real allocation time in our simulations, as shown in Figure 11. The allocation time by our approach is 32 to 51 percent better than that of BL. This is mostly due to the low complexity of our allocation algorithm, as proved in section 5.1. Next, we simulate by varying the interarrival time from 6 to 12 seconds at 10 seconds service rate, but fixing the mesh size at 64x64, as well as all other parameters. The submesh sizes are generated under those three distributions for the given size of the mesh system. In this case, results show that our approach is consistently superior to all others. Figure 12 gives the comparison of BL and FL for the average delay versus interarrival time for a Linear Distribution. The results for Pseudo-Normal and Uniform Distributions are similar to the Linear Distribution and hence are not shown. The improvement of FL over BL for the allocation time in this case is demonstrated in Figure 13 for the three distributions. The improvement is significant in all three cases, and higher the system load, larger is the improvement.

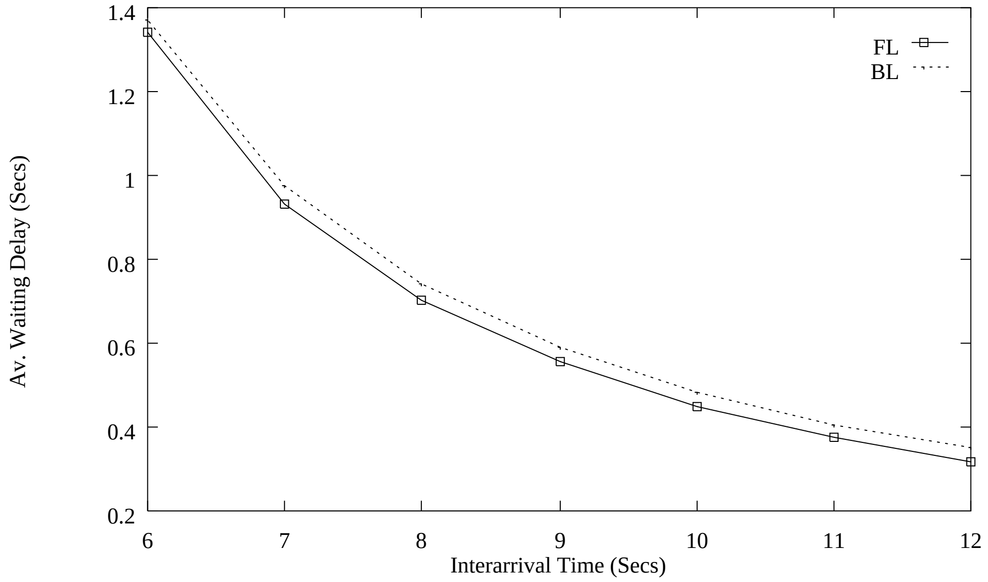

Figure 12: Submesh size for a Linear Distribution and M(64,64).

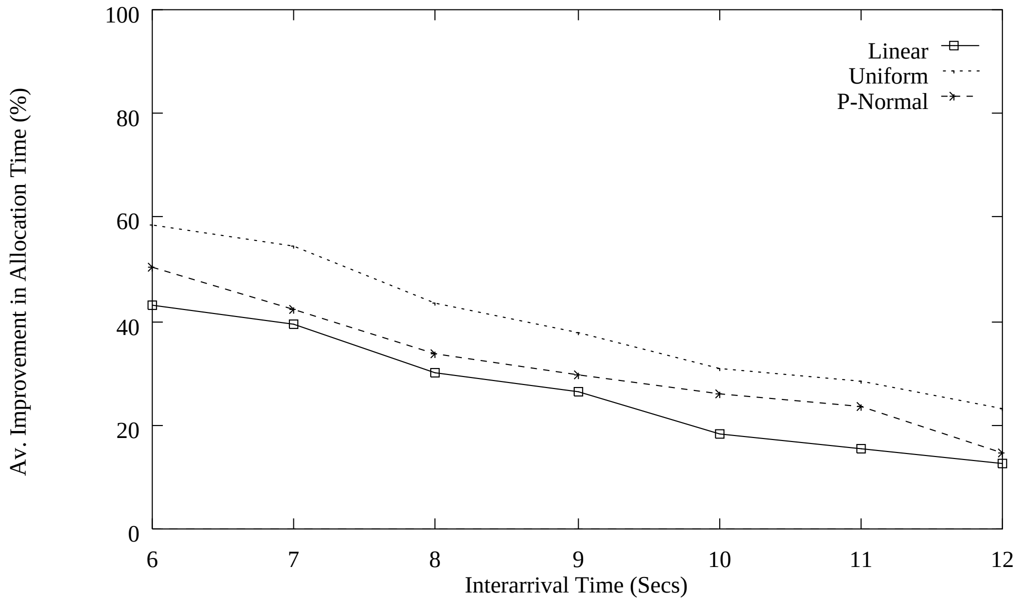

Figure 13: Submesh size for the Three Distributions and M(64,64).

6 Conclusion

In this paper, we have proposed an efficient submesh allocation strategy suitable for distributed shared memory multiprocessors. The proposed approach is based on a free list. Unlike the free list strategy used for hypercubes, the free list here is an overlapped one, where the free submeshes in the list may overlap with each other. We have developed a low complexity allocation algorithm $O(N_F^2)$ compared to $O(N_B^3)$ of the Busy-List strategy. Previously, it was shown that a Busy-List strategy can perform better than the Adaptive Scan, the Frame Sliding, and the Buddy strategies. The extensive simulation results show that the average waiting delay and the allocation time of the proposed strategy are always better than all the other existing strategies. Overall, our approach is superior to the previous strategies.

In general, for a multiprocessor system, the proposed free list strategy for processor allocation may have the advantage of efficiently allocating a group of processors as required in the allocation process. This characteristic can be generalized provided an efficient technique can be deviced for maintaining the free list itself.

References

[1] Intel, *Paragon XP/S: Product Overview*, Intel Corporation, 1991.

[2] D. Lenoski and et.al., "The Directory Based Coherent Protocol for the DASH Multiprocessors," *Proc. International Symposium on Computer Architecture*, May 1990.

[3] Cray Research, Inc., *Cray MPP Technology Preview*, 1992.

[4] nCUBE Corporation, *nCUBE 2 Processor Manual*, December 1990.

[5] BBN Advanced Computer Inc., *Butterfly GP1000 Switch Tutorial*, 1989.

[6] Y. Chang and L. N. Bhuyan, "Extending Multistage Interconnection Networks for Multitasking," In *Proc. International Conference on Parallel Processing*, pp. I–151–158, August 1992.

[7] K. Li and K. H. Cheng, "A Two Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System," In *Proc. ACM Computer Science Conf.*, pp. 22–28, 1990.

[8] P. J. Chuang and N. F. Tzeng, "An Efficient Submesh Allocation for Mesh Computer Systems," In *Proc. International Conference on Distributed Computing Systems*, pp. 256–263, May 1991.

[9] Y. Zhu, "Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 328–337, 1992.

[10] J. Ding and L. N. Bhuyan, "An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems," In *Proc. International Conference on Parallel Processing*, pp. 193–200, Aug. 1993.

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

[12] J. Kim, C. R. Das, and W. Lin, "A Top-Down Processor Allocation Scheme for Hypercube Computers," *IEEE Transactions on Parallel and Distributed Systems*, pp. 20–30, January 1991.

[13] K. C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, pp. 22–23, Jan. 1985.

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