Methods for Precise Submesh Allocation

CRAIG MORGENSTERN
Department of Computer Science, Texas Christian University, Fort Worth, TX 76129

ABSTRACT

In this article we describe and compare several recently proposed algorithms for precise submesh allocation in a two-dimensional mesh connected system. The methods surveyed include various frame sliding strategies, the maximum boundary value heuristic, and interval set scan techniques. In addition, a new enhancement to the interval set scan method is described. This enhancement results in an algorithm that has better allocation and run-time performance under a FCFS scheduling policy than any of the other proposed methods. We present results drawn from an extensive simulation study to illustrate the relative efficiency of the various methods. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

This article is concerned with the design and analysis of submesh allocation methods to manage the processors of a two-dimensional mesh connected system. The problem of submesh allocation is similar to that of "two-dimensional" memory management. In a multiuser mesh system, several tasks may be running simultaneously. Each task is exclusively allocated a rectangular submesh of exactly the required size, and a submesh remains allocated to a task until the task completes. That is, we are concerned with precise rectangular submesh allocation in which submesh migration is not allowed. When a task requests a rectangular submesh of processors, the job of the allocation method is to attempt to locate such a submesh and to do so as efficiently as possible in both of the following categories:

1. Run-time efficiency measures the time required to locate a submesh of the requested dimensions and consisting entirely of unallocated processors, or to determine that such a submesh does not currently exist in the system.
2. Allocation efficiency measures the effectiveness with which an allocation algorithm manages the processors of the mesh system.

Several submesh allocation methods have been proposed in the recent literature [1–7]. With the single exception of the binary buddy method [4], these methods all differ from their one-dimensional memory management counterparts in that they utilize data structures to keep track of what has been allocated rather than what is available. This is because the available "space" in a mesh becomes fragmented into shapes that can be represented by numerous differing collections of rectangular submeshes. A method that proceeds by managing the unallocated processors must either maintain several representations in the face of allocations and deallocations or select and enforce a single representation at each allocation and deallocation. The first approach would seem to result in excessive algorithmic complexity and run-time performance whereas the second approach would seem to result in poor allocation performance (as exhibited by the binary buddy [1, 2, 4]). Thus,

Received April 1994
Revised May 1994

the general scheme for submesh allocation has been to identify candidate submesh locations and to check the candidates for conflicts against what is already allocated, rather than to search some representation of what is unallocated.

The submesh allocation methods that have appeared in the literature fall into three main categories:

1. Frame sliding: These methods proceed by identifying candidate submeshes that could satisfy a task request. The state of the mesh is maintained by keeping a list of currently allocated submeshes. This list is scanned once for each candidate until either a candidate is found that does not conflict with any currently allocated submeshes, or until all candidates have been examined. The frame sliding methods differ in how efficiently they identify candidates. Recently proposed frame sliding methods include the original method by Chuang and Tzeng [1], Ding and Bhuyan's [3] adaptive scan (AS) method, Sharma and Pradhan's [6] maximum boundary value (MBV) method, and the fast frame sliding (FS_f) method described by Morgenstern and Fouque [5].
2. Interval set scan: These methods, first introduced by Morgenstern and Fouque [5], maintain a compressed representation of the state of the mesh using sets of integer intervals. Interval set scan (ISS) methods have the same or better allocation performance as the best frame sliding methods but exhibit better run-time performance.
3. 0/1 Array: Zhu [7] describes a scheme in which the state of the mesh is mirrored in a 0/1 two-dimensional array, one array element per mesh processor. If a processor is unallocated/allocated then the corresponding array element contains a 0/1. Allocation algorithms then search this array instead of a list of allocated submeshes. All of the frame sliding strategies can be implemented to use a 0/1 array scheme and vice versa.

Additionally, Ding and Bhuyan [3] propose that mesh systems provide hardware or software support for processor address translation that would allow a request to be rotated 90°; i.e., a $x \times y$ submesh could possibly be satisfied by a $y \times x$ submesh. Allowing rotations to be performed results in a significant increase in allocation efficiency.

This article is organized as follows. Section 2 describes the frame sliding strategy and its numerous variants. Section 3 contains the presentation of the ISS method and enhancements. In Section 4 the run-time efficiency of the methods is analyzed and the results of a simulation study are described and presented. The article concludes with Section 5.

The following notation and definitions are used throughout the article. A mesh connected system, $M(X, Y)$, consists of XY processors arranged in a $X \times Y$ two-dimensional grid. Processors are addressed as (a, b) pairs where $0 \leq a < X$ and $0 \leq b < Y$. A $x \times y$ rectangular submesh has a quadruple address (a, b, c, d) where

$$x = c - a + 1,$$
$$y = d - b + 1,$$

(a, b) specifies the upper leftmost processor and (c, d) specifies the lower rightmost processor. The (a, b) processor is called the base of the submesh and (c, d) is called the reverse base processor.

2 FRAME SLIDING STRATEGIES

Chuang and Tzeng [1, 2] propose an allocation method called frame sliding, which maintains a list, L, of addresses of submeshes that are currently allocated in M. When a task requests a submesh of dimensions $x \times y$, a set of candidate base locations is checked for allocation suitability. Checking the suitability of a candidate base location, (a, b), requires that the list of resident submesh addresses be scanned to determine if any of the processors in candidate submesh $(a, b, a + x - 1, b + y - 1)$ are already allocated. Thus, if an allocation is possible at (a, b), then L must be completely scanned to determine this fact. If an allocation at (a, b) is not possible, then the scan terminates when the first conflict is encountered.

2.1 Restricted Frame Sliding

At one extreme, the set of candidate base locations contains all possible processor locations and is given by

$$C_1 = \{(a, b) \mid 0 \leq a \leq X - x, 0 \leq b \leq Y - y\}, \quad (1)$$

Because the use of C_1 results in excessively high run-time complexity, Chuang and Tzeng [1, 2] suggest that a smaller candidate base set be used to restrict the search space and increase the speed of the search. For example, their suggested frame sliding method uses the set

$$C_n = \{(a, b) \mid 0 \le a \le X - x, a \bmod x = 0 \quad 0 \le b \le Y - y, b \bmod y = 0\}, \qquad (2)$$

and so it is possible for several feasible allocation bases to be skipped and for a request to fail even though an allocation is possible. The frame sliding method that uses C_1 is denoted by FS_1 and the restricted version that uses C_n by FS_n. Use of C_n is a compromise that trades a loss of allocation performance (such as processor utilization and task waiting time) for a gain in run-time efficiency. However, Chuang and Tzeng [1, 2] give the results of a simulation study that compares frame sliding to a two-dimensional buddy allocation strategy [4]. Their study shows that FS_n results in much better allocation performance than is possible with a buddy allocation strategy.

To determine if a request for a $n \times y$ submesh can be satisfied, Chuang and Tzeng [1, 2] suggest that the candidate base sets C_1 and C_n be defined as

$$C_1 = \{(a, b) \mid 0 \le a < X, 0 \le b < Y\}$$

and

$$C_n = \{(a, b) \mid 0 \le a < X, a \bmod x = 0 \quad 0 \le b < Y, b \bmod y = 0\}, \qquad (3)$$

When processing a request, this definition requires that $(X - x, 0, X - 1, Y - 1)$ and $(0, Y - y, X - 1, Y - 1)$ be temporarily added to L because none of the processors in these temporary submeshes can serve as a base of a submesh that satisfies the request. Ding and Bhuyan [3] and Morgenstern and Fouque [5] noted that it is more efficient to implement C_1 and C_n exactly as defined in Equations 1 and 2 to shrink the number of candidate base locations that have to be checked and to avoid increasing the size of L.

2.2 FS_f and AS

Major improvements in time efficiency can be made when searching candidate base set C_1. Suppose that candidate (i, j) is found to be infeasible because one or more of the processors in $(i, j, i + x - 1, j + y - 1)$ are already allocated. Let (a, b, c, d) be a submesh on the allocated submesh list, L, which has a processor in $(i, j, i + x - 1, j + y - 1)$. Then $i \le c$ and candidate base locations $(i + 1, j), \ldots, (c, j)$ can be safely skipped; i.e., base location $(c + 1, j)$ can be safely tried. If $c + 1 > X - x$, then the search begins anew at $(0, j + 1)$. Ding and Bhuyan [3] propose to completely scan L to find the conflicting submesh with the largest c value. Their heuristic, AS, proceeds as shown in Figure 1.

```
for $j := 0$ to $Y - y$ do
  begin
    $i := 0$;
    while $i \le X - x$ do
      begin
        frame := $(i, j, i + x - 1, j + y - 1)$;
        $S := \{s \in L \mid s$ and frame conflict$\}$;
        if $S = \emptyset$ then
          allocate frame and return(success)
        else
          $i := \max\{c + 1 \mid (a, j, c, d) \in S\}$
      end
  end;
return(failure);
```

FIGURE 1 Determining if a request can be satisfied using AS.

Morgenstern and Fouque [5] also noted that candidate base locations could be eliminated on the fly. Their approach, FS_f, prunes more candidates from C_1 than AS and does so more efficiently. Rather than scanning all of L, FS_f stops at the first submesh, (a, b, c, d), encountered in L that has a processor in $(i, j, i + x - 1, j + y - 1)$. Then as before, the next candidate base location tried is $(c + 1, j)$. Our simulation results show that terminating the scan of L at the first conflict is much more efficient than the AS strategy of doing a complete scan to find the conflicting submesh with the largest c value. Further, if all candidate base locations in row j have been eliminated, then it is possible to safely restart the search at candidate $(0, h)$ for some $h \ge j + 1$, where h is the smallest reverse base row of all conflicting submeshes on L that were encountered when searching row j (see Fig. 2).

Both AS and FS_f have the same allocation performance as FS_1 but have much better run-time efficiency. In fact, FS_f is actually faster on some simulation runs than the original FS_n frame sliding version that uses the restricted candidate set C_n.

```
j := 0;
while $j \leq Y - y$ do
  begin
    i := 0;   h := Y;
    while $i \leq X - x$ do
      begin
        frame := $(i, j, i + x - 1, j + y - 1)$;
        okay := true;
        for $(a, b, c, d) \in L$ st okay do
          if $(a, b, c, d)$ and frame conflict then
            begin
              i := c + 1;   okay := false;
              if $h > d + 1$ then $h := d + 1$
            end;
        if okay then allocate frame and return(success)
      end;
    j := h;
  end;
return(failure);
```

FIGURE 2 Determining if a request can be satisfied using FS_f.

2.3 Frame Placement and Best-Fit Heuristics

In addition to the completeness of the candidate base set, another issue is the order in which candidate bases are checked. A first-fit policy has been implicitly assumed; the first candidate base encountered that does not result in a conflict with an already allocated submesh is the one that is used to satisfy a request. With this assumption, is it possible to obtain better allocation efficiency by checking candidate bases in an order than the implicitly assumed first-fit row-major ordering? Morgenstern and Fouque [5] report that frame placement heuristics can make a substantial difference. In particular, they note that the following heuristic results in a noticeable improvement of allocation efficiency over numerous other orderings (including row major):

> Order candidate set C_1 so that the requested submesh is allocated with its largest side as close to an edge of the mesh as possible.

This and other frame placement heuristics are difficult to implement efficiently using the frame sliding strategy because they present an obstacle to "on the fly" pruning of C_1. In the following section the ISS search framework is described, which does allow for efficient implementation of this heuristic.

The frame sliding strategy can also be used to support various best-fit heuristics. However, even though C_1 can be pruned using FS_f while searching for the frame with the best fit, simulation results show that an excessive number of frames are still examined, which leads to excessive run-time. Both Zhu [7] and Sharma and Pradhan [6] have devised heuristics to support an intuitive notion of two-dimensional best fit. Sharma and Pradham report that their heuristic, MBV, has better allocation performance than Zhu's heuristic. The boundary value of an unallocated node (a, b), in the mesh is given by the sum of the number of allocated neighbors of (a, b) and the number of mesh boundary points on which (a, b) lies. Corner nodes (e.g., $(0, 0)$) lie on two boundary points, horizontal and vertical. The boundary value of a frame is the sum of all boundary values in the frame's periphery. The MBV heuristic proceeds by selecting the frame of the requested size that has the largest boundary value and consists entirely of unallocated processors.

3 ISS METHODS

A new submesh allocation algorithm called the ISS method is described that has the same allocation performance as the frame sliding methods FS_1, FS_f, and AS that use candidate base set C_1 [5]; i.e., if an allocation is possible then ISS will find it. However, our simulation studies show that the run-time efficiency of ISS is much better than the frame sliding methods (even when compared to frame sliding using the restricted candidate base set C_n). Thus, the ISS method has both better run-time efficiency and better allocation efficiency than is possible using a row-major, first-fit version of the frame sliding strategy. In this section the basic ISS method is described as well as an enhancement to efficiently support the frame placement heuristic described earlier.

3.1 Basic ISS

The state of the mesh is maintained by the use of a data structure called an interval set. Let $[a, b]$ denote an interval of contiguous integers from a to b where $a \leq b$. Two disjoint intervals, $[a, b]$ and $[c, d]$, are said to be adjacent if $b + 1 = c$ or $d + 1 = a$. An interval set is a representation of a set of disjoint nonadjacent intervals on a range of contiguous integers, together with the following operations:

1. `create_interval_set()`: Return a new empty interval set.

2. `add_interval(s, i)`: Add interval i to set s, where i is assumed to be disjoint from intervals already in s. If i is adjacent any interval j in s, then i and j are merged into a single interval; i.e., all intervals in s are nonadjacent after the inclusion of i.
3. `delete_interval(s, i)`: Delete the interval i from s, where i is assumed to belong to s or to be a subinterval of an interval in s.

An interval set can be easily implemented using an ordered linked list structure in which each node contains the endpoints of an interval. Two interval sets, B_i and R_i, are associated with each row i of M. Whenever submesh (a, b, c, d) is allocated, interval $[a, c]$ is added to B_b and to R_d. When (a, b, c, d) is deallocated, $[a, c]$ is deleted from both sets B_b and R_d. Thus, set B_b contains a representation of the "top" row of processor addresses of all allocated submeshes whose base processor is in row b. R_d contains a representation of the "bottom" row of processor addresses of all allocated submeshes whose reverse base processor is in row d. For example, the nonempty base and reverse base interval sets corresponding to the mesh given in Figure 4 are:

$$B_0 = \{[0, 8], [10, 13]\} \quad R_0 = \{[0, 8]\}$$
$$B_1 = \{[7, 8]\} \quad R_5 = \{[7, 8]\}$$
$$B_3 = \{[2, 3]\} \quad R_6 = \{[2, 3]\}$$
$$R_8 = \{[10, 13]\}$$

To determine if a request for a $x \times y$ submesh can be satisfied, a data structure similar to an interval set is used, called a scan set. A scan set is used to efficiently maintain a representation of a collection of (possibly overlapping) intervals obtained from several interval sets. Each element, $([a, b], p)$, of a scan set consists of an interval $[a, b]$ and a count p. This element represents the fact that p intervals, whose intersection is $[a, b]$, have been added to the scan set. The intervals belonging to a scan set are all disjoint and no two adjacent intervals can have the same count—adjacent intervals with the same count are merged. A scan set can also be easily implemented using an ordered list in which each node contains a scan set element. The operations supported by a scan set include:

1. `initialize_scan()`: Initialize the scan set to contain the single empty interval, $([0, X - 1], 0)$, where X is the width of the entire mesh.

2. `open_interval(x)`: Returns the leftmost (lowest) value of an interval of size at least x with a count of 0; i.e., if $([a, b], p)$ is in the scan set with $p = 0$ and $b - a + 1 \geq x$, then a is returned. If such an interval does not exist, then -1 is returned.
3. `add_interval_set(s)`: Add all intervals belonging to the interval set s to the scan set.
4. `delete_interval_set(s)`: Delete all intervals belonging to the interval set s from the scan set.

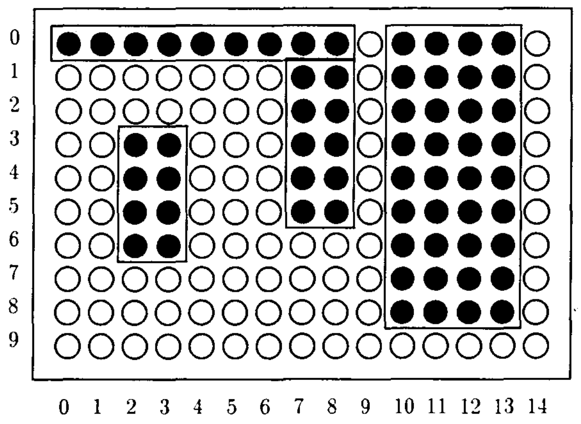
The routine given in Figure 3 shows how a scan set is used to determine if a request for a $x \times y$ submesh can be satisfied. We maintain a "compressed" representation of y contiguous mesh rows, $r_0, \ldots, r_{y-1}$, in the scan set. If (a, b, c, d) is an allocated submesh with $r_0 \leq b \leq r_{y-1}$ or $r_0 \leq d \leq r_{y-1}$, then the interval $[a, c]$ will have been added to the scan set, so no allocation will be made in those columns. If the scan set contains an element $([e, f], p)$ with $p = 0$ and $f - e + 1 \geq x$, then the request can be satisfied. Otherwise, we "drop" row r_0 and "add" row r_y; meaning that all intervals that correspond to a reverse base in r_0 are deleted and all intervals that correspond to a base in r_y are added. This is illustrated in Figure 4.

```
initialize_scan();
$j := 0$;
for $r := 0$ to $y - 1$ do add_interval_set($B_r$);
$r := y$;
while $r < Y$ and open_interval($x$) = $-1$ do
  begin
    delete_interval_set($R_j$);  $j := j + 1$;
    add_interval_set($B_r$);  $r := r + 1$
  end;
$i :=$ open_interval($x$);
if $i > -1$ then
  allocate at $(i, j, i + x - 1, j + y - 1)$ and return(success)
else
  return(failure);
```

FIGURE 3 Determining if a request can be satisfied using ISS.

3.2 Four-Way ISS

The simulation results reported [5] indicate that allocating a submesh with its largest side as close to an edge of the mesh as possible results in near optimal allocation performance under a FCFS



Rows	Scan Set
initial	$\{([0,14],0)\}$
0	$\{([0,8],1),([9,9],0),([10,13],1),([14,14],0)\}$
0,1	$\{([0,6],1),([7,8],2),([9,9],0),([10,13],1),([14,14],0)\}$
1,2	$\{([0,6],0),([7,8],1),([9,9],0),([10,13],1),([14,14],0)\}$
2,3	$\{([0,1],0),([2,3],1),([4,6],0),([7,8],1),([9,9],0),([10,13],1),([14,14],0)\}$
5,6	$\{([0,1],0),([2,3],1),([4,6],0),([7,8],1),([9,9],0),([10,13],1),([14,14],0)\}$
6,7	$\{([0,1],0),([2,3],1),([4,9],0),([10,13],1),([14,14],0)\}$
7,8	$\{([0,9],0),([10,13],1),([14,14],0)\}$
final	allocate at $(0,7,9,8)$

FIGURE 4 Progression of an interval scan to allocate a 10 × 2 request.

scheduling policy. The allocation method used by Morgenstern and Fouque [5] to implement this heuristic was based on frame sliding and had extremely poor run-time performance. A simple enhancement to the basic ISS method is now described. Called the four-way ISS (4ISS), it can be used to efficiently implement this heuristic.

In addition to maintaining base row and reverse base row interval sets, 4ISS also maintains base column and reverse base column interval sets. Each row j of M is associated with interval sets B_j and R_j as before, and additionally each column i of M has interval sets B'_i and R'_i associated with it. Whenever submesh (a, b, c, d) is allocated, interval $[a, c]$ is added to B_b and to R_d, and interval $[b, d]$ is added to B'_a and R'_c. When (a, b, c, d) is deallocated, $[a, c]$ is deleted from both sets B_b and R_d, and $[b, d]$ is deleted from B'_a and R'_c. Set B'_a contains a representation of the "left" column of processor addresses of all allocated submeshes whose base processor is in column a. R'_c contains a representation of the "right" column of processor addresses of all allocated submeshes whose reverse base processor is in column c. For example, the nonempty base and reverse base column sets corresponding to the mesh given in Figure 4 are:

$$
\begin{array}{llll}
B'_0 = \{[0, 0]\} & R'_3 = \{[3, 6]\} \\
B'_2 = \{[3, 6]\} & R'_8 = \{[0, 5]\} \\
B'_7 = \{[1, 5]\} & R'_{13} = \{[0, 8]\} \\
B'_{10} = \{[0, 8]\} &
\end{array}
$$

To determine if a $x \times y$ submesh request can be satisfied, 4ISS does alternating top and bottom row scans if $x \geq y$; otherwise, alternating left and right column scans are performed (using the base and reverse base column interval sets). A top row scan is just the ISS method whereas a bottom row scan starts at the bottom of the mesh, adds reverse base row interval sets to the scan set, and deletes base row intervals sets from the scan set. Similarly, a left column scan starts at the left side of the mesh, adds base column interval sets to the scan set, and deletes reverse base column interval sets from the scan set. A right column scan starts at the right side of the mesh, adds reverse base column interval sets to the scan set, and deletes base column interval sets from the scan set. These alternatives are illustrated in Figure 5. Thus, at each allocation attempt two scan sets are active (top and bottom, or left and right), and the search

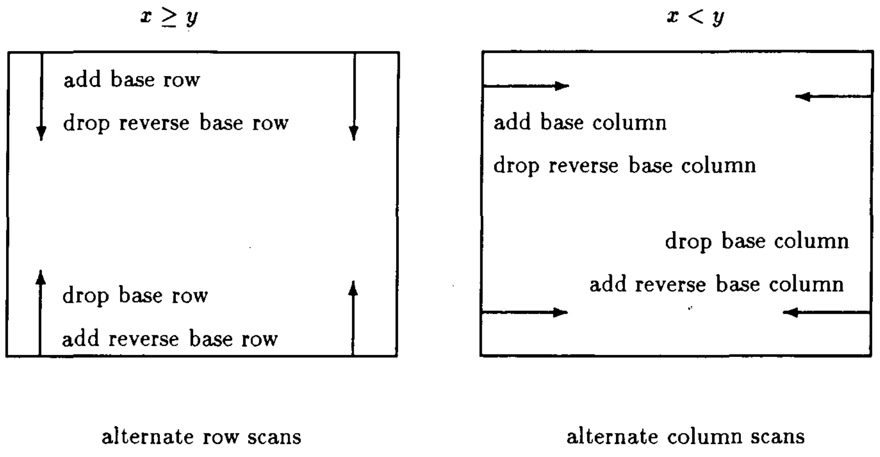


FIGURE 5 Scan direction alternatives for 4ISS.

alternates between the two scan sets one row or column at a time. If the two scans completely cross each other (by x columns or y rows), then no allocation is possible. Otherwise, as in ISS, the first free $x \times y$ submesh encountered is allocated. This free submesh will have its largest side as close to the edge of the mesh as is possible, and the amount of work required to find the submesh is no more than the worst case under ISS (when the largest side is x and the available submesh is near the bottom).

4 ANALYSIS AND EXPERIMENTAL RESULTS

The worst case running-times of the allocation methods are fairly easy to derive. For the analysis, we will assume that the mesh is square with $X = Y = N$.

1. All of the frame sliding methods require a scan of the list of allocated submeshes, L, for each candidate frame at each allocation attempt. For a $x \times y$ request, there can be as many as $(X - x)(Y - y)$ frames and so these methods are bounded by $O((X - x)(Y - y)\,|L|)$. This gives an $O(X^2Y^2)$ or $O(N^4)$ absolute worst case of when the mesh is full of 1×1 submeshes and a 1×1 request is being serviced. The time required for deallocation is constant because the pointer into L can be associated with the allocated submesh during allocation and passed back to the deallocation routine during deallocation.
2. The ISS methods do at most N traversals of interval sets and N traversals of scan sets. Each set can have no more than N elements, resulting in a $O(N^2)$ worst case allocation time. Deallocation requires two or four traversals of interval sets and so has a worst case time of $O(N)$.
3. Methods based on Zhu's [7] 0/1 array scheme will require $\Theta(N^2)$ allocation time because the $N \times N$ array must be copied and traversed at each allocation. Deallocation of a $x \times y$ submesh would require $\Theta(xy)$ time.
4. Sharma and Pradhan [6] describe a (complex) alternative frame sliding implementation to support their MBV heuristic. Their implementation identifies candidate frames by scanning L and allocation requires $O(|L|^3)$ time or $O(N^6)$ absolute worst case time. As for frame sliding, deallocation takes constant time.

From this analysis, it can be seen that when the mesh is sparsely populated, several of the methods will exhibit equally good run-time performance. In this case, the pruning heuristics of FS_f and AS will be effective, the interval and scan sets of the ISS methods will be small, and the size of L will be small (allowing Sharma and Pradhan's alternative MBV implementation to be competitive). One objective of our simulation study is to stress the methods by running them on densely populated meshes.

4.1 Simulation Model

Numerous simulation runs were performed using the same simulation model described previously [1–3, 5]. In this model, allocation is performed centrally by a dispatcher that is independent of the mesh system. A fixed number of submesh requests are queued at the dispatcher with predetermined side lengths and resident times that follow selected distributions. All of the requests are immediately pending, simulating a worse case task interarrival time of zero. An FCFS scheduling order is maintained; the dispatcher attempts to satisfy the request at the head of the queue. If the request cannot be satisfied, then the dispatcher reattempts after an allocated submesh has been released. This continues until the request at the head of the queue has been satisfied, at which time the head request is dequeued and the next request serviced.

The main parameters needed to perform a simulation run are: the allocation method, the mesh dimensions, the submesh side range and distribution, the submesh resident time range and distribution, the number of submesh requests to be queued at the dispatcher, and whether or not to allow submesh requests to be rotated. When comparing methods, we used the same queue of submesh requests; i.e., we used the same random number stream seeds and only altered the allocation method parameter. In all the simulation runs that we report, the submesh resident times (task service times) were real values uniformly distributed in the range of 5 to 30. The precise values used for the other simulation parameters are described later. With this simulation model, we determined allocation efficiency of the various allocation schemes by making the following measures:

1. The simulated time required to completely service all requests. This is the simulated time at which the final allocated submesh gets released; called completion time.
2. The percentage of a processor in the mesh that is utilized per unit of simulated time; called processor utilization.
3. The percentage of total processors of a requested submesh over the total number of processors in the mesh at each allocation failure, in which the number of available processors at the time of failure is at least as large as the number of requested processors; called external fragmentation.
4. The percentage of total processors of a requested submesh over the total number of processors in the mesh at each "feasible" allocation failure; called migration fragmentation. An allocation failure is feasible if an allocation is possible by rearranging or migrating the currently allocated submeshes. Because deciding whether or not an allocation failure is feasible requires an exponential search, this measure could not be taken on all of the simulation runs—the decision algorithm was intractable when the number of allocated submeshes was larger than 10.

4.2 Methods Tested

The allocation methods that we tested are summarized below:

1. ISS: interval set scan (first fit).
2. 4ISS: four-way interval set scan (first fit, allocates a submesh with its largest side as close to the edge of the mesh as possible).
3. FS_n: restricted frame sliding (first fit, uses candidate set C_n).
4. FS_f: fast frame sliding (first fit, prunes candidate base set C_1).
5. AS: adaptive scan (first fit, prunes candidate base set C_1).
6. MBV: maximum boundary value (best fit, uses FS_f to allocate a submesh with maximum boundary value).
7. MIG: This method attempts to allocate by first using FS_f. If FS_f fails to allocate, then attempt to allocate by rearranging or migrating the currently allocated submeshes so as to make room for the pending request. Although this is not a practical allocation method, it is included for comparison purposes. Its completion time and processor utilization measures are the best possible under an FCFS scheduling order. By definition, this method does not have migration fragmentation.

In addition, we performed two sets of simulations runs for each method, one in which rotations were not allowed and one in which the methods could rotate a submesh request. Both Ding and Bhuyan [3] and Sharma and Pradhan [6] incorporated rotations in the implementations of their methods but did not allow the methods that they compared against to do rotations. Rotations result in a substantial increase in allocation performance, and so it is important to isolate the effect of allowing rotations in all methods tested. When running the MIG method and when deriving migration fragmentation values under rotations, we adapted the rule that, once allocated, a submesh could be migrated but not rotated. This results in lower processor utilization and higher completion time values for the MIG method, but was necessary to make the search tractable. Similarly, the migration fragmentation values are higher than they would be without this rule.

Run-time efficiency was measured by counting the number of nodes touched in the linked data structures used by each method. For each method, a single step is performed when a pointer is dereferenced. The actual cpu time of the simulation runs followed the trends indicated by the step counts—the step counts serve to better illustrate the differences in run-time efficiency. In fact, the frame sliding methods exhibited a slightly worse run-time efficiency ratio to other methods if actual cpu time is considered. Three run-time measures are given: the average number of steps needed to make a successful allocation, the average number of steps needed to decide that an allocation is not possible (i.e., a failed allocation), and the average number of steps taken per attempted allocation whether successful or not.

4.3 Results

Table 1 is reprinted from Morgenstern and Fouque [5] and contains the average of the results of five independent simulation runs per distribution on a 256 × 256 processor mesh in which 1000 submesh requests were queued at the dispatcher for each run. This table is augmented to include the results of running AS, 4ISS, and MBV using the same simulation parameters that were used to produce the results for the other methods. The side lengths of submesh requests were made over uniform and normal distributions, using two random number streams, one stream per side. Under the uniform distribution, submesh side lengths were generated over the range 1 to 256. Under the normal distribution, the mean was 128 and the standard deviation was set to 43 [2, 5]. As can be observed from Table 1, 4ISS and MBV have the best allocation efficiency. Under a uniform side distribution, 4ISS performs so well that little improvement is possible. As might be expected, our implementation of MBV suffers from excessive run-time complexity. It should be noted that Sharma and Pradhan's [6] $O(|L|^3)$ implementation would have a running-time similar to the other fast methods because the mesh is so sparsely populated. Next in allocation efficiency

Table 1. Allocation Performance in a 256 × 256 Mesh (Without Rotations)

Method	Completion Time	Processor Utilization	External Fragmentation	Migration Fragmentation	Steps per Success	Steps per Failure	Steps per Attempt
Uniform distribution							
MIG	8367.9	53.90	32.6	0.0			
4ISS	8637.5	52.27	32.9	24.9	4.10	10.61	7.20
MBV	8755.4	51.56	33.5	30.9	8275.0	2.40	4138.6
ISS	9020.0	50.06	33.7	30.7	3.22	5.34	4.26
FS_f	9020.0	50.06	33.7	30.7	3.82	2.28	3.06
AS	9020.0	50.06	33.7	30.7	100.3	208.8	155.1
FS_1	9020.0	50.06	33.7	30.7	5553.6	9195.2	7371.5
FS_n	10837.5	41.64	33.2	28.9	10.68	6.38	7.56
Normal distribution							
MIG	8575.6	50.58	29.7	0.0			
4ISS	8914.3	48.66	29.6	24.6	3.27	11.77	7.53
MBV	9078.7	47.78	29.9	26.4	3928.5	3.11	1964.2
ISS	9527.9	45.56	29.8	25.8	2.84	5.94	4.38
FS_f	9527.9	45.56	29.8	25.8	3.26	2.72	2.98
AS	9527.9	45.56	29.8	25.8	90.1	252.8	171.5
FS_1	9527.9	45.56	29.8	25.8	5558.7	14438.2	9993.2
FS_n	12265.7	35.36	28.6	24.6	2.84	3.62	3.76

Table 2. Allocation Performance in a 256 × 256 Mesh (With Rotations)

Method	Completion Time	Processor Utilization	External Fragmentation	Migration Fragmentation	Steps per Success	Steps per Failure	Steps per Attempt
Uniform distribution							
MIG	7465.0	60.48	33.4	0.0			
4ISS	7720.5	58.46	33.3	28.1	7.36	17.96	12.68
MBV	7881.5	57.28	34.2	31.9	7194.4	4.44	3600.5
ISS	8104.5	55.72	35.2	33.6	5.00	10.01	7.48
Normal distribution							
MIG	7684.5	55.46	29.7	0.0			
4ISS	7917.9	54.80	29.5	23.7	6.34	23.00	14.66
MBV	8055.3	53.88	29.9	26.4	3334.5	5.94	1669.8
ISS	8495.5	51.06	30.6	26.3	4.66	12.44	8.56

are the four allocation-equivalent methods, ISS, FS_f, AS, and FS_1, that all do a row-major search of the C_1 candidate base set. Of these four, only ISS and FS_f have low time complexity and FS_f does a much more efficient job of pruning C_1 than AS. The results of allowing methods to rotate submesh requests are given in Table 2. Allowing rotations does not change the ranking of the methods and an across-the-board increase in allocation efficiency is observed (the allocation efficiency values for AS, FS_1, and FS_f are the same as those for ISS). Finally, the external and migration fragmentation values are somewhat confusing. For example, the external fragmentation values of MIG are nearly identical to the other methods, even though no allocation is possible without doing preemption

Table 3. Allocation Performance in a 1024 × 1024 Mesh

Method	Completion Time	Processor Utilization	Number Allocated	Steps per Success	Steps per Failure	Steps per Attempt
Submesh sides uniformly distributed between 1 and 1024						
MIG	32224.8	53.0	2.2			
4ISS	33332.7	51.3	2.0	4.3	7.5	7.5
MBV	33763.8	50.6	2.0	139337.7	2.6	69661.4
ISS	35069.5	48.7	1.8	3.4	5.7	4.5
FS_n	41231.5	41.5	1.4	13.9	5.1	9.5
FS_f	35069.5	48.7	1.8	3.9	2.4	3.2
Submesh sides uniformly distributed between 1 and 512						
4ISS	7516.4	57.0	9.3	22.2	66.0	44.1
ISS	8259.0	51.8	8.5	23.7	47.9	36.0
FS_n	10852.4	39.4	6.4	50.2	25.7	38.0
FS_f	8259.0	51.8	8.5	34.0	32.3	33.2
Submesh sides uniformly distributed between 1 and 256						
4ISS	1697.0	63.3	41.6	130.6	441.6	286.3
ISS	1746.4	61.5	40.5	216.5	442.2	328.9
FS_n	2563.6	41.9	27.3	271.3	246.7	259.0
FS_f	1746.4	61.5	40.5	667.9	886.5	776.7
Submesh sides uniformly distributed between 1 and 128						
4ISS	404.6	66.9	180.8	984.2	3549.3	2279.2
ISS	412.0	65.7	176.7	2144.6	4362.0	3234.4
FS_n	573.3	47.2	124.7	2868.4	3319.9	3031.1
FS_f	412.0	65.7	176.7	17216.6	23791.9	20448.2
Submesh sides uniformly distributed between 1 and 64						
4ISS	110.1	62.5	769.1	6764.2	25826.7	16625.0
ISS	109.5	62.8	731.9	12443.6	30317.3	20453.3
FS_n	138.9	49.5	556.2	47663.1	48318.5	47965.6
FS_f	109.5	62.8	731.9	465699.5	627873.2	538373.4

when these measurements are taken. This is evidence that external fragmentation (in two dimensions) is a misleading measure of allocation efficiency under an FCFS scheduling policy.

To further distinguish between the performance of the faster methods, 4ISS, ISS, FS_f, and FS_n, we tested them with another set of simulation runs designed to stress their run-time efficiency. Table 3 contains the results of five simulation runs on a 1024 × 1024 mesh in which 4000 submesh requests were queued at the dispatcher for each run. The side lengths of submesh requests were drawn from a uniform distribution over ranges that varied from 1 to 64 up to 1 to 1024. The smaller ranges resulted in a large number of resident submeshes: the average number of allocated submeshes per allocation attempt is given in the fourth column of the table. The efficiency of the ISS strategy is seen. Its compact representation of the mesh allows it to quickly process allocation requests in a mesh that is already dense with resident submeshes. On the other hand, as the submesh side request range shrinks, both FS_f and FS_n have a run-time performance that approaches that of FS_1. The MBV method, either as implemented here or using Sharma and Pradhan's [6] $O(|L|^3)$ implementation, would also exhibit extremely large allocation times on dense meshes.

Finally, several other submesh side request and resident time distributions were used and mesh sizes of up to 4096 × 4096 were tested. The results included here are representative of numerous other results not reported here due to space limitations. In particular, the ranking of the methods did not change from what is implied by the results in Tables 1, 2, and 3.

5 CONCLUSIONS

This article presented a comprehensive evaluation of several of the submesh allocation methods that have appeared in the literature. A completely new method that uses interval sets was also introduced. From the results, the methods can be ranked as follows:

5.1 Allocation Time Performance

Sparse Mesh

In a sparsely populated mesh, several of the methods have near identical allocation times. 4ISS, ISS, MBV (Sharma and Pradhan's [6] implementation), and FS_f are all efficient. Implementations using Zhu's [7], $\Theta(XY)$, 0/1 array scheme will not be competitive on sparse meshes.

Dense Mesh

Of the methods we tested, 4ISS and ISS dominated allocation time performance on dense meshes. Because the ISS methods approach their $O(XY)$ worse case running-time on these meshes, it is likely that implementations that use Zhu's 0/1 array scheme will also be competitive.

5.2 Allocation Efficiency Performance

Sparse Mesh

4ISS dominates MBV, which clearly dominates the allocation equivalent methods ISS, FS_f, AS, and FS_1. There is no need for the allocation/time efficiency compromise embodied in FS_n.

Dense Mesh

Our results show that 4ISS again dominates on the methods tested. It is not known for certain what the allocation performance of MBV would be relative to 4ISS on very dense meshes, but the trends indicate that 4ISS would be slightly more efficient.

The main conclusions that we can make from our study are that:

1. First-fit methods can outperform best-fit methods provided that the candidate frames are searched in the proper order.
2. Rotation of submesh requests increases the allocation efficiency of all allocation methods by nearly the same amount.
3. Of all the methods tested and analyzed, only 4ISS dominates in both allocation efficiency and allocation time performance on both sparse and densely populated meshes.

However, much work remains to be done in this area—in particular other scheduling policies besides FCFS need to be examined.

REFERENCES

[1] P.-J. Chuang and N.-F. Tzeng, *Proceedings of the 11th International Conference on Distributed Computing Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 256–263.

[2] P.-J. Chuang and N.-F. Tzeng, "Allocating precise

submeshes in mesh connected systems," *IEEE Trans. Parallel Distrib. Systems,* vol. 5, pp. 211–216, 1994.

[3] J. Ding and L. Bhuyan, *Proceedings of the 1993 International Conference on Parallel Processing.* Boca Raton, FL: CRC Press, 1993, vol. II, pp. 193–200.

[4] K. Li and K.-H. Cheng, "A two dimensional buddy system for dynamic allocation in a partitionable mesh connected system," *J. Parallel Distrib. Comput.,* vol. 12, pp. 79–83, 1991.

[5] C. Morgenstern and P. Fouque, *Proceedings of the 27th Hawaii International Conference of System Sciences.* Los Alamitos, CA: IEEE Computer Society Press, 1994, vol. II, pp. 493–501.

[6] D.-D. Sharma and D. Pradhan, *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing.* Los Alamitos, CA: IEEE Computer Society Press, 1993, pp. 682–689.

[7] Y. Zhu, "Efficient processor allocation strategies for mesh-connected parallel computers," *J. Parallel Distrib. Comput.,* vol. 16, pp. 328–337, 1992.