Allocating Precise Submeshes in Mesh Connected Systems

Po-Jen Chuang and Nian-Feng Tzeng

***Abstract*—We propose a new processor allocation strategy that applies to any mesh system and recognizes submeshes of arbitrary sizes at any locations in a mesh system. The proposed strategy allocates a submesh of exactly the size requested by an incoming task, completely avoiding internal fragmentation. Because of its efficient allocation, this strategy exhibits better performance than an earlier allocation strategy based on the buddy principle. An efficient implementation of this strategy is presented. Extensive simulation runs are carried out to collect experimental cost and performance measures of interest under different allocation schemes.**

***Index Terms*—Centralized control, distributed approaches, mesh connected systems, cost and performance measures, simulation, submesh allocation.**

I. INTRODUCTION

Various parallel architectures have become economically feasible as a result of advanced very large scale integration (VLSI) technology. The mesh topology, because of its simple and regular structure suitable for VLSI implementation, has been drawing considerable attention from researchers of different fields in recent years. Based on this topology, several prototypes and commercial systems have been built or are under construction, such as CLIP [1], the Tera Computer System [2], and the Touchstone Delta System [3]. They exhibit a high potential for the parallel execution of various algorithms, e.g., image processing, matrix multiplication, as well as for solving partial differential equations [4], and may deliver performance as good as typical supercomputers offer, and at a much lower cost.

Considered in this short note is a mesh connected system where the resources are the processor nodes forming submeshes of various sizes. An incoming task incident on the system is analyzed for decomposability, and the number of processors required for the task is determined. An appropriate submesh is then assigned to the task. The job of a processor allocator is to find an available submesh with a size just sufficient to meet the need of the task. In this short note, we limit our attention to the processor allocation of two-dimensional (rectangular) mesh systems. An incoming task is assumed to need a rectangular submesh. Allocation in a high-dimensional mesh system can be investigated similarly.

The recognition of various submeshes in a mesh connected system using the two-dimensional buddy strategy has been proposed recently by Li and Cheng [5]. The strategy is a generalization of the traditional one-dimensional binary buddy system developed for memory management [6]. This strategy, however, is applicable only to the square mesh system and allocates only square submeshes, both with certain predefined sizes. More specifically, the side lengths of the system and its allowable submeshes are limited to exact powers of 2. Consequently, this strategy can neither be applied to square mesh systems whose dimensions are not powers of 2 nor be incorporated in a *non-square* mesh system like the Touchstone Delta System [3]. Besides, an incoming task is always assigned to a $2^i \times 2^i$ square submesh such that 2^i is no less than the maximum of the two side lengths of the actually required submesh. As a result, this strategy often allocates a submesh with its sizes "dilated" and is thus termed a *dilated* submesh allocation scheme. Because it cannot recognize rectangular submeshes of arbitrary sizes, this strategy tends to result in low processor utilization due to large internal fragmentation. Another allocation scheme, proposed by Livingston and Stout [7], requires all available (nonfaulty and nonbusy) processors to participate in the allocation job in parallel. The correctness of their scheme is assured by intensively *sending* and *receiving* messages among processors operating in a synchronized fashion. As a consequence, it is readily suited for an *SIMD* mesh system, but not for an *MIMD* mesh. Besides, a common way of synchronization requires the broadcast operation that is not supported by certain mesh implementations, such as the wormhole routing meshes [10].

In light of the facts that processors in a mesh system are not always arranged in a square fashion and a dilated submesh allocation scheme often wastes a significant amount of processor resources (due to large internal fragmentation), and hence imposes unnecessary delay on an incoming task, it is desirable to consider a more efficient allocation technique that can apply to any mesh system, recognize submeshes with arbitrary sizes, and assign a submesh of the precise size to an incoming task, totally eliminating internal fragmentation. In this short note, we propose a new strategy that achieves precise submesh allocation, and thereby achieves high processor utilization, as well as short task waiting time in *any* mesh system, even those with unequal sides. Note that a non–square mesh system could come from a square mesh after failures arise and several rows or columns of processors are removed (i.e., bypassed) during reconfiguration. The proposed strategy searches an available submesh capable of accommodating an incoming task by examining a sequence of candidate "frames," each with the same size in both dimensions as the requested submesh, until the processors within a "frame" are found all available and are assigned to the incoming task. Determining whether all processors within a frame are available can be done quickly by managing three simple sets appropriately (without checking every processor individually). This strategy can be implemented in a centralized or distributed manner. It is an efficient strategy for allocating precise submeshes on demand. Simulation results indicate that the proposed strategy gives rise to significantly improved performance over a dilated submesh allocation scheme while maintaining reasonably low time complexity.

This short note is organized as follows. Section II presents useful notation and related work on mesh processor allocation. Section III introduces the proposed allocation strategy and addresses its implementation. Section IV provides the experimental results on the cost and performance measures of interest.

II. NOMENCLATURE AND RELATED WORK

A two-dimensional (rectangular) mesh, denoted by $M_2\ (w, h)$, consists of $w \times h$ nodes arranged in a $w \times h$ two-dimensional grid. (Notice that a node refers to a processor; nodes and processors are used interchangeably.) The node in row i and column j is identified by address $\langle i, j \rangle$ and is connected through direct communication links to $\langle i \pm 1, j \rangle$ and $\langle i, j \pm 1 \rangle$, for $0 \leq i < w$ and $0 \leq j < h$ (a boundary node has fewer neighbors). A two-dimensional (rectangular) submesh in $M_2(w, h)$, denoted by $S_2(w', h')$, is a subgrid $M_2(w', h')$ such that $1 \leq w' \leq w$ and $1 \leq h' \leq h$. The address of a submesh

Manuscript received May 20, 1991; revised December 9, 1991, and October 27, 1992. N.-F. Tzeng was supported in part by the National Science Foundation under Grants MIP-8807761 and MIP-9201308, and in part by the State of Louisiana under Contract LEQSF (1992–94)–RD–A–32. A preliminary version of this work was presented at the 11th International Conference on Distributed Computing Systems, May 1991.

P.-J. Chuang is with the Department of Electrical Engineering, TamKang University, Tamsui, Taipei Hsien, Taiwan, Republic of China.

N.-F. Tzeng is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

IEEE Log Number 9214470.

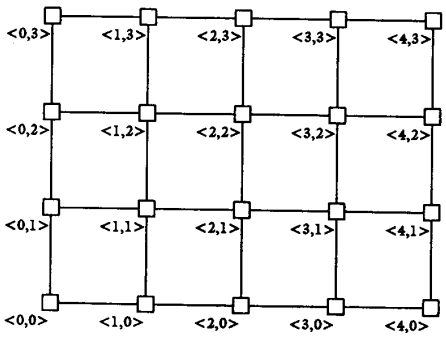

Fig. 1. A rectangular mesh $M_2(5,4)$.

is denoted by a quadruple (x, y, x', y'), where $\langle x, y\rangle$ indicates the lower-left corner coordinate of the submesh, and $\langle x', y'\rangle$ is the upper-right corner. Fig. 1 shows a rectangular mesh $M_2(5,4)$. Nodes $\langle 2,1\rangle$, $\langle 3,1\rangle$, $\langle 4,1\rangle$, $\langle 2,2\rangle$, $\langle 3,2\rangle$, and $\langle 4,2\rangle$, for example, constitute a submesh $S_2(3,2)$ with address $(2,1,4,2)$.

The processors in $M_2(w,h)$ must be allocated to incoming tasks efficiently to reduce fragmentation and obtain high processor utilization. An incoming task T requesting a rectangular submesh $S_2(w',h')$, with $1 \leq w' \leq w$ and $1 \leq h' \leq h$, is specified by $T = (w', h')$. In what follows, we briefly review prior allocation strategies.

A. Two-Dimensional Buddy Strategy

The two-dimensional buddy strategy [5], being a generalization of the conventional one-dimensional binary buddy system [6], is applicable only to the situation where all incoming tasks need square submeshes $S_2(w',w')$'s, and the system itself is a square mesh $M_2(w,w)$, with both w' and w being exactly powers of 2. Assuming $w = 2^r$, this strategy maintains a set of lists of available square submeshes with side length $w' = 2^{r'}$ such that one list is for an r' value, where $0 \leq r' \leq r$. An incoming request for a submesh of side q $(= 2^k)$ is allocated the first element in the list of free square submeshes $S_2(2^k,2^k)$'s, if the list is not empty; otherwise, an available square submesh with $r' > k$ is decomposed before being allocated to the request. Although its allocation procedure is straightforward, this strategy involves a complicated deallocation procedure, whenever a submesh is released, in order to guarantee its correctness. Specifically, the deallocation procedure first has to merge the released submesh with other submeshes of identical size, if any, to form a bigger submesh, and then remove the merged submeshes from the list. This process will be repeated for the newly formed submesh until nothing further can be done. To this end, the strategy is implemented [15] by assigning an *order* to each submesh in the list, through which each submesh can identify its three buddies. A bigger submesh is created from a submesh, together with its three buddies if the three buddies are all in the list.

This strategy cannot be incorporated in square mesh systems with side lengths that are not powers of 2, not to mention in a non–square mesh system like the Touchstone Delta System [3], where 528 processors are arranged in the two-dimensional mesh $M_2(33,16)$. Under the strategy in question, an incoming request for $S_2(w',h')$ is assigned a square submesh $S_2(w,w)$, with w being the maximum of w' and h' rounded up to the nearest power of 2, i.e., $w = 2^{\lceil \log_2(\max(w',h')) \rceil}$ where *max* is a function that returns the bigger value of w' and h'. Consequently, a significant amount of processor resources could be wasted because of large internal fragmentation, which may otherwise form certain submeshes able to accommodate the subsequent incoming tasks. For example, this strategy allocates an $S_2(8,8)$ to a request that actually needs a submesh $S_2(2,5)$, wasting 54 processors. A task may be unnecessarily deferred to enter the system as a result of this inefficient submesh allocation.

B. Parallel Allocation Scheme

A parallel allocation scheme is proposed by Livingston and Stout [7] for a d-dimensional mesh $M_d(n_1, n_2, \cdots, n_d)$, with $n_1 = n_2 = \cdots = n_d$. In this scheme, all available (nonfaulty and nonbusy) processors participate in performing the allocation job. In a one-dimensional mesh $M_1(n)$, a processor with address $\langle i\rangle$ is considered to be the *leader* of an $S_1(t)$, for some $t < n$, provided that each of the processors with addresses $\langle i\rangle, \langle i+1\rangle, \cdots, \langle i+t-1\rangle$ is available, but processor $\langle i+t\rangle$ is not, meaning that $\langle i\rangle$ is the starting point of t consecutive free processors. Processor $\langle 0\rangle$ is the leader of $S_1(n)$ if all of the processors are available. An available submesh is positioned by a leader. Similarly, in a two-dimensional mesh $M_2(n,n)$, a processor is the leader of $S_2(t,t)$, provided that the node itself is a leader of $S_1(t')$, $t' \geqslant t$, along one dimension, and that each of its $t-1$ successors along the second dimension is also such a leader. Each available processor determines the size of S_1 of which it is the leader along one dimension by repeatedly communicating with its two immediate neighbors along the dimension in a synchronized fashion. After this, each available processor can likewise determine if it is the leader of a two-dimensional submesh of the required size by appropriately communicating with its two neighbors along the second dimension.

To guarantee the correctness of this scheme, all participant processors must send and receive messages in a strictly synchronized manner, which is possible only in an $SIMD$ mesh. Besides, a common way of synchronization requires the broadcast operation that is not supported by certain mesh implementations, such as the wormhole routing meshes [10].

III. PROPOSED ALLOCATION STRATEGY

We introduce a new submesh allocation strategy based on frame sliding, termed the *FS strategy*, which can recognize submeshes of actual sizes needed by incoming tasks. The FS strategy completely eliminates internal fragmentation and exhibits better performance than the buddy strategy.

A. Basic Idea

All processors within a "frame" of a given size can be told immediately when the location of a particular frame point, say the lower left corner of the frame, is specified. Hereafter, a *frame* refers to those processors within it, and a frame is positioned by its lower left corner address, i.e., the lower left corner address serves as the location of the frame.

We observe that in response to a request, the buddy strategy checks frames at certain specific locations dictated by the size of the requested submesh. The mesh system $M_2(16,16)$ illustrated in Fig. 2(a), where each square represents the location of a processor, gives an example. When an incoming task $T = (8,8)$ arrives, the buddy strategy checks only four locations, namely, A, C, I, and K, for availability. On the other hand, it examines locations A through P (but not other locations) in response to a task $T = (4,4)$. Because all possible locations selected by the buddy strategy are fixed and implicit according to the size of a requested submesh, the strategy can be implemented in a way that a free list is designated for keeping track of available square submeshes of a certain size, and there is no need to explicitly record the locations of all allocated submeshes. Although simple at the allocation step, this scheme is fairly restricted in identifying the possible locations and sizes of recognizable submeshes, making it potentially inefficient. Consider

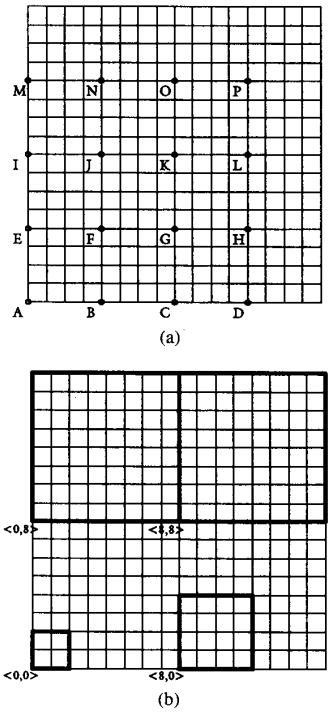

Fig. 2. (a) Recognizable submeshes in $M_2(16,16)$ under the buddy strategy. (b) Four allocated square submeshes present in $M_2(16,16)$, indicated by bold boxes.

Fig. 2(b), for instance: An incoming task of $T = (5,4)$ cannot enter the mesh system in the presence of four allocated square submeshes with addresses (0, 0, 1, 1), (8, 0, 11, 3), (0, 8, 7, 15), and (8, 8, 15, 15), as shown in the figure, despite the fact that a frame with size 5×4 at node $\langle 2,0\rangle$ involves only free processors and would be assigned to the task if it were recognizable. The deallocation step of this strategy also tends to be fairly complex.

To remedy this inadequacy and to improve flexibility in identifying available submeshes (in both their sizes and locations), we propose using a frame whose size is identical to that of the requested submesh (rather than fixed and square) and whose possible location depends on the available resources at the time of allocation (rather than implicit and predefined). When processors in the currently examined frame are not all available, the frame is slid over the "plane" of the mesh system in search of next candidates, with the horizontal and vertical strides equivalent to, respectively, the width and height of the requested submesh. The search process starts with the frame at the lowes-leftmost available processor, called *start_loc*. When the same incoming task $T = (5,4)$ reaches Fig. 2(b), for example, a 5×4 frame at $\langle 2,0\rangle$, the start_loc, is examined first. In this case, the first attempt succeeds, and the processors in the examined frame will be assigned to the incoming task.

If the processors in an examined frame are not all available, the next candidate frame is checked until a frame involving only free processors is found, or until all candidate frames are exhausted. The latter case indicates that no suitable submesh exists to accommodate the incoming task and that the task cannot enter the mesh system at this time. Consider an incoming task $T = (8,4)$ incident to the mesh system shown in Fig. 2(b). The first examined 8×4 frame, which is again at start_loc $\langle 2,0\rangle$, is unavailable, and the second candidate frame at node $\langle 10,0\rangle$ (determined by sliding the frame rightward along the horizontal direction with a stride 8, the width of the requested submesh) is then checked. This candidate is apparently unavailable, because node $\langle 10,0\rangle$ itself is not free. When the frame is further slid rightward, the location of the next candidate is found outside the boundary of the system. Therefore, the frame is slid along the vertical direction with a stride 4, the height of the requested submesh, and then moved leftward such that the right end of the frame is at the right boundary of the mesh system, reaching the candidate frame at node $\langle 8,4\rangle$. Now this frame is available, and all processors in it are assigned to the incoming task. Note that after making a vertical stride, the frame is slid repeatedly along the horizontal direction until it exceeds the boundary in the opposite side. In general, the search process is carried out by examining a sequence of frames that altogether cover the entire mesh "plane."

B. Implementing the FS Strategy

The proposed strategy can be implemented efficiently, without examining the availability of every processor in a candidate frame individually during search, by appropriately generating two sets according to the requested submesh size and the current occupancy situation in the mesh system. The *base* of a (sub)mesh is the processor at the lower-left corner of the (sub)mesh. In Fig. 1, for example, the bases of mesh $M_2(5,4)$ and the submesh with address (2, 1, 4, 2) are processors $\langle 0,0\rangle$ and $\langle 2,1\rangle$, respectively. Some definitions are given before an efficient implementation of the FS strategy is presented.

Definition 1: The busy set of a mesh system is the collection of all current allocated submeshes in the system.

The busy set of the mesh shown in Fig. 2(b) consists of four elements corresponding to the four allocated square submeshes, i.e., $\{(0, 0, 1, 1), (8, 0, 11, 3), (0, 8, 7, 15), (8, 8, 15, 15)\}$.

Definition 2: The coverage of an allocated submesh α with respect to an incoming task T, denoted by $\xi_{\alpha,T}$, is a collection of processors, each of which cannot serve as the base of any available submesh for accommodating T. The coverage set is the union of the coverages of all allocated submeshes, i.e., the elements in the busy set.

As an example, in Fig. 1, if the submesh with address (2, 1, 4, 2) is allocated and $T = (2,1)$ is requested by an incoming task, the coverage of the allocated submesh comprises all processors in the submesh (1, 1, 4, 2). In general, let (x, y, x', y') be the address of an allocated submesh α for an incoming task $T = (i,j)$. It is easy to obtain that coverage $\xi_{\alpha,T}$ comprises all of the processors of the submesh $(x - i + 1, y - j + 1, x', y')$.

Definition 3: A reject submesh with respect to an incoming task is a submesh involving processors that can never be the base of any available submesh for accommodating the incoming task. The reject set is the collection of all such reject submeshes.

To give an example, the two submeshes (3, 0, 4, 3) and (0, 3, 4, 3) in Fig. 1 form the reject set with respect to an incoming task $T = (3,2)$. In general, for system $M_2(w,h)$, the reject set with respect to an incoming task $T = (i,j)$ comprises two submeshes with addresses $(w-i+1, 0, w-1, h-1)$ and $(0, h-j+1, w-1, h-1)$.

The next theorem serves as the basis of an efficient implementation of our strategy.

Theorem 1: Let Ψ, Δ, and x be the coverage set, the reject set with respect to an incoming task, and the location of a frame, respectively. Then the processors within the frame are all available only if x does not belong to any element in Ψ or Δ.

This theorem follows directly from the fact that 1) if x belongs to any element in Ψ, the frame would be overlapped with some allocated submesh(es); and 2) if x belongs to any element in Δ, then (a portion of) the frame would be outside the boundary of the mesh system. When an incoming task arrives, all candidate frames can be determined immediately and are examined in sequence, starting with the one positioned at start_loc. Whether an examined frame is

available can be determined quickly by checking its lower-left corner node against the coverage set and the reject set, which are produced on receiving the incoming task.

Variable start_loc, which dictates the starting location of each search process, is initialized to $\langle 0,0 \rangle$ and may be updated at an allocation/deallocation step, depending upon whether such a step leads to a change of the position of the lowest-leftmost available processor in the system. Specifically, start_loc is updated at an allocation step if the first examined frame is assigned to the incoming task, making the lowest-leftmost available processor changed to the first subsequent available processor. Likewise, if a released frame is positioned at a row no higher than the row specified by start_loc (when they are at the same row, the released frame is to the left of start_loc), this deallocation step causes start_loc to shift to the location of the released frame, reflecting the current position of the lowest-leftmost available processor. The proposed FS allocation strategy is formally described as follows.

(The busy set is empty and start_loc is $\langle 0,0 \rangle$ initially.)

Processor Allocation:

Step 1) Set $i := w'$ and $j := h'$, where $T = (w', h')$ is the current incoming task.

Step 2) Generate the coverage set (from the busy set) and the reject set according to i and j.

Step 3) Compare the lower-left corner node of every candidate frame in sequence, starting with start_loc, against the coverage set and the reject set. If it belongs to any element of the two sets, proceed to the next candidate; otherwise, go to the next step. If all candidates are exhausted, go to Step 5.

Step 4) Include the selected submesh (i.e., the current frame) in the busy set, update start_loc if necessary, and then allocate the submesh to T. Stop.

Step 5) Attach T to the task queue and wait until a submesh is released.

Processor Relinquishment: Remove the released submesh from the busy set and update start_loc if necessary.

Our allocation strategy can be carried out either by a host processor centrally or by multiple free processors in a distributed manner. In a centralized approach, the host processor keeps the busy set and start_loc, and only the host processor is responsible for allocation. When a request arrives at the mesh system, the host processor executes the allocation steps provided above.

In a distributed approach, all of the free processors are involved in the job of processor allocation. Each free processor checks in parallel whether it can be the base of a submesh suitable for the incoming task, but the busy set is managed uniquely by the host processor. Like the parallel allocation scheme proposed by Livingston and Stout [7], the distributed approach also requires the support of broadcast operations, which are not implemented in certain mesh systems. During the entire course of performing the allocation job, however, the involved processors need not communicate with one another; instead, they communicate only with the host processor. This makes our strategy more efficient and suitable for an *MIMD* mesh system.

IV. Experimental Performance Evaluation

Extensive simulation runs are conducted to collect important cost and performance measures of different strategies for mesh processor allocation. Because of difficulty in quantifying the communication time and cost, we simulated only our allocation strategy under centralized control and the buddy strategy,[1] leaving out our strategy operating in a distributed manner, as well as the parallel allocation scheme by Livingston and Stout [7]. Simulation studies on various mesh sizes ranging from 16×16 to 1024×1024 were carried out, with all of the results following a trend similar to those reported here for $M_2(256, 256)$. The entire simulation is performed on a Sun 4 System.

The simulation model is given as follows. Task allocation is carried out centrally by a dispatching processor outside the simulated mesh system so that a request for all of the mesh processors is allowed. Initially, the entire simulated mesh is free, and 1000 tasks are generated and queued at the dispatcher. The dimensions of submeshes, and the residence times requested by these 1000 tasks, are assumed to follow given distributions. The dispatcher attempts to allocate the task at the top of the queue first, and if it fails to identify an available submesh of the desired size for the task, it reattempts when a submesh is released until an available one is eventually obtained; i.e., the FCFS scheduling discipline is followed. When a task gets allocated, it is removed from the queue, and the next task is served at the following time unit. No new task is generated during the course of simulation.

Under this simulation model, we collected such performance measures as the completion time (Ξ—the time taken to finish all 1000 tasks), processor utilization (the percentage of a processor being used per unit time) over the period Ξ, internal fragmentation (F_{int}), external fragmentation (F_{ext}), and total fragmentation (F_{tot}). F_{int} is similar to what is defined in [5]: the percentage of overallocated mesh processors (i.e., those exceeding what are actually needed) over allocated mesh processors. F_{ext} refers to the percentage of total processors in the requested submesh over total processors at each feasible allocation failure (in which the number of processors available is no less than the number of processors requested). F_{tot} indicates the percentage of total processors wasted in allocation, and it can be calculated according to (*hit ratio*) $\times$ F_{int} + (*miss ratio*) $\times$ F_{ext}, where *miss ratio* is the rate of total feasible allocation failures to total feasible requests and *hit ratio* $= 1 -$ *miss ratio*. Note that F_{ext} and F_{tot} are defined differently from those given in [5], because our preceding definitions appear to be more appropriate. The simulation results are averaged over five independent runs.

The results for processor allocation in $M_2(256, 256)$ are shown in Table I. The side lengths of submeshes requested by the 1000 tasks are governed by four distributions—uniform, normal, decreasing, and increasing distributions. It should be noted that the two side lengths of each requested submesh are under the same distribution, with two different random number streams, one for each side, employed to generate the length value. A rectangular submesh so generated can be of any arbitrary size. The uniform distribution indicates that each side length is uniformly distributed between 1 and 256, which seems to reflect well the situations where the nature of tasks to be executed is unknown. By contrast, the decreasing distribution indicates that the probability of requesting a larger submesh is lower, reflecting better the cases where most tasks have reasonable parallelism, but are not suitable for fine-grained partition. The probabilities of side lengths of requested submeshes under the decreasing and increasing distributions are shown below the table. $P_{[a,b]}$ denotes the probability of side length to fall within $Q = [a, b]$, and all possible side lengths in Q are uniformly distributed. The mean of the normal distribution is one-half of the mesh system side size, 128, and the standard deviation is 43. The residence times of the 1000 tasks are governed by two distributions: uniform and normal distributions. Simulation studies on various ranges of residence time were carried out, and all of the results followed a trend similar to those presented here; i.e., for the

[1] The simulation program for the buddy strategy was provided by K. Li and K. H. Cheng.

TABLE I
SIMULATION RESULTS OF PERFORMANCE MEASURES FOR $M_2(256, 256)$

Side-length distribution		Uniform		Normal		Decreasing*		Increasing**	
Residence Time Distribution		Uniform	Normal	Uniform	Normal	Uniform	Normal	Uniform	Normal
Completion time	Buddy	16642.9	18443.0	16641.2	18433.6	11785.5	12833.8	17478.3	19492.2
	FS	9922.1	10766.8	11050.4	12084.0	4176.6	4520.0	15234.5	16718.6
Processor utilization	Buddy	25.8	26.2	26.0	26.2	12.0	12.5	51.2	50.8
	FS	43.2	44.9	39.1	39.9	33.9	35.5	58.7	59.2
Internal fragmentation	Buddy	68.9	68.9	69.3	69.3	81.9	81.9	47.6	47.6
	FS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
External fragmentation	Buddy	28.1	28.1	28.6	28.5	14.9	15.0	42.7	42.4
	FS	33.5	33.1	29.1	29.1	25.2	24.8	40.4	39.9
Total fragmentation	Buddy	61.1	61.1	61.4	61.5	57.2	58.2	47.5	47.5
	FS	14.6	14.0	14.1	13.9	11.1	10.2	10.4	10.1
Memory requirement	SC***	3.5	3.5	3.2	3.2	6.5	6.4	2.7	2.7
	SC****	8.8	8.4	6.2	6.4	16.0	16.0	6.0	6.2
FS time complexity	NC†	16.4	16.3	4.0	4.0	47.0	50.8	6.9	6.8
	NC††	13.7	13.8	5.8	5.7	39.3	41.7	6.8	6.8
	NC†††	15.3	14.7	4.1	4.0	44.4	43.8	7.9	7.8
	NC††††	13.1	13.0	5.9	5.9	38.5	40.5	7.4	7.4

*$P_{[1,32]} = 0.4$, $P_{[33,64]} = 0.2$, $P_{[65,128]} = 0.2$, $P_{[129,256]} = 0.2$.
**$P_{[1,128]} = 0.2$, $P_{[129,192]} = 0.2$, $P_{[193,224]} = 0.2$, $P_{[225,256]} = 0.4$.
***Size of union of coverage and reject sets (average case).
****Size of union of coverage and reject sets (worst case).
† Number of comparisons per successful attempt (first 1000 tasks).
†† Number of comparisons per attempt (first 1000 tasks).
††† Number of comparisons per successful attempt (second 1000 tasks).
†††† Number of comparisons per attempt (second 1000 tasks).

uniform distribution, it ranges from 5 to 30 time units, and for the normal distribution, its mean and standard deviation are 20 and 5, respectively.

The results shown in Table I are reasonably accurate. Given 95% confidence, for instance, the first value 16642.9 (in time unit) provided in the table has the calculated confidence interval half-width over the five replications equal to 456.7, meaning that we are 95% confident that the true result would fall into the interval of 16642.9 ± 456.7, or equivalently, $16642.9 \pm 2.7\%$. This simulated value involves only less than 3% error.

As can be observed from Table I, the FS strategy has significantly shorter completion time and notably larger processor utilization than the buddy strategy, particularly under the uniform, normal, and decreasing side-length distributions. Simulation results are almost the same for both uniform and normal residence time distributions. Shorter completion time is a direct result of more efficient processor utilization for the FS strategy, which totally eliminates internal fragmentation (whose value ranges from roughly 69% to 82%, as experienced by the buddy strategy shown in Table I). This clearly demonstrates the advantage of our strategy over a dilated submesh allocation scheme, like the buddy strategy. Note that both strategies hold almost the same external fragmentation under the normal and increasing side-length distributions, whereas for the uniform and decreasing distributions, the FS strategy exhibits larger external fragmentation. The total fragmentation, nevertheless, is shown to be much less for the FS strategy, meaning that fewer processors are wasted in allocation.

The next two rows give the average-case and worst-case memory requirement using the FS strategy, measured by the number of elements in the union of the coverage set and the reject set (which comprise the major memory space required by the FS strategy). As we can see, the memory requirement is no more than 4 in the average case and no more than 9 in the worst case under all distributions except the decreasing distribution. Note that one element in the coverage set or the reject set consists of four coordinates that take no more than four words. The largest memory requirement happens under the decreasing distribution, because tasks then are more likely to request smaller submeshes, and, at any point of time, more allocated submeshes (most of which are small) exist simultaneously in the mesh system. The required size of memory remains reasonable, however: No more than 7 in the average case, and no more than 16 in the worst case.

We then studied the situation where the number of tasks generated and queued at the dispatcher is 2000 initially. The last four rows of the table give time complexity using the FS strategy, measured by the expected number of comparisons per (successful) allocation attempt. The results in the first two rows are accumulated from the first 1000 tasks, and the next two rows are from the second 1000 tasks, that is, when there are many allocated submeshes present at the time when the first task under consideration (i.e., task 1001) enters. By a comparison, we mean taking one processor to see if it is in one element of the coverage set or the reject set. A comparison thus involves checking the two coordinates of the processor address against the two ranges specified by the element. Since every attempt often requires a small number of comparisons, and since each comparison includes only simple operations, the mean search time for finding an available submesh is very short, yielding low time complexity. As can be noticed from the first two rows, it takes about 42 comparisons for

TABLE II
SIMULATION RESULTS OF TIME COMPLEXITY

System	Strategy	Complexity	Submesh Side-Length Distribution			
			Uniform	Normal	Decreasing*	Increasing**
$M_2(256,256)$	Buddy	# operations	2636	2170	4554	1295
		# orders	7641	5861	16880	2142
		CPU time	95	94	106	93
	FS***	#operations	2386	2346	5649	2042
		#coverages	5140	5135	6636	5001
		CPU time	82	81	82	84
$M_2(1024,1024)$	Buddy	# operations	2636	2154	4554	1295
		# orders	7641	5775	16880	2142
		CPU time	96	95	108	91
	FS***	#operations	2386	2327	5649	2042
		#coverages	5140	5120	6636	5001
		CPU time	83	86	83	82

* For $M_2(256,256)$, see Table I; for $M_2(1024,1024)$: $P_{[1,128]} = 0.4$, $P_{[129,256]} = 0.2$, $P_{[257,512]} = 0.2$, $P_{[513,1024]} = 0.2$.

**For $M_2(256,256)$, see Table I; for $M_2(1024,1024)$: $P_{[1,512]} = 0.2$, $P_{[513,768]} = 0.2$, $P_{[769,896]} = 0.2$, $P_{[897,1024]} = 0.4$.

*** FS strategy with restricted allowable submesh sizes.

an allocation attempt in the worst case, which is still favorable when compared with the buddy strategy, because the latter tends to involve high overhead in the deallocation procedure. If a desired submesh is available (a successful attempt), it takes a little more mean search time to identify such a submesh, because all elements in the coverage and reject sets must be compared exhaustively in order for the frame to be assigned. The worst case happens under the decreasing side-length distribution, because as mentioned earlier, more small allocated submeshes exist simultaneously under this distribution at any time, leading to more elements in the coverage set to be compared at every allocation attempt. It is also observed that time complexities are almost the same for the first 1000 and the second 1000 tasks, indicating that the expected number of comparisons per (successful) allocation attempt remains the same when there are many allocated submeshes present initially.

The search spaces (composed of recognizable submeshes) are different for precise (like the FS strategy) and dilated (like the buddy strategy) submesh allocation strategies. To compare the costs meaningfully, the allowable submesh sizes of the FS strategy are limited to those permitted by the buddy strategy so that the two strategies would have the same search space. The total number of operations performed and the CPU time (in seconds) required to finish the allocations and deallocations of the 1000 tasks in $M_2(256,256)$ and $M_2(1024,1024)$ are accumulated and listed in Table II. By an operation, we mean a comparison (described earlier) for the FS strategy, whereas for the buddy strategy, we mean decomposing a bigger available submesh, inserting a submesh in the appropriate position of a particular list, or merging and removing a submesh together with its three buddies from the list. We concern only the major operations involved in the strategies, excluding trivial operations such as calculating the order in the buddy strategy and generating the coverage for a particular submesh in the FS strategy, which are collected and shown separately. Since all of the simulation results under the normal residence time distribution follow the same trend as they do under the uniform residence time distribution, only those under the uniform residence time distribution are presented. The side lengths of submeshes requested by the 1000 tasks in $M_2(256,256)$ are under the same four distributions as those given in Table I. The side length of requested submeshes in $M_2(1024,1024)$ is uniformly distributed between 1 and 1024 under the uniform distribution, and has a mean value of 512 and a standard deviation value of 171 under the normal distribution. The probabilities of side lengths under the decreasing and the increasing distributions are provided below the table. The FS strategy is found to involve more operations under the normal, decreasing, and increasing distributions (by 8%, 24%, and 58%, respectively), but requires slightly fewer operations under the uniform distributions. It should be remembered, however, that an operation of the FS strategy is far simpler than that of the buddy strategy; thus, the total time spent could be more for the buddy strategy than for the FS strategy with restricted allowable submesh sizes. The total number of orders calculated in the buddy strategy and the total number of coverages generated in the FS strategy are also listed in the table for reference. Note that generating a coverage is much simpler than calculating an order, and that coverages involved are fewer than orders, except for the increasing distribution. As shown in the table, the FS strategy always requires less CPU time to finish allocating the 1000 tasks. It can also be observed that the numbers of operations and CPU times involved are almost identical in $M_2(256,256)$ and in $M_2(1024,1024)$ for both strategies, implying that the FS strategy, like the buddy strategy, involves no higher complexity for a larger system, ensuring its suitability for a system of any size.

Instead of making strides according to the width and height of a requested submesh, we investigate other choices of stride values. Shortening strides, intuitively, could have more candidate frames examined, and thus could lead to a higher probability of successfully finding an available candidate frame for an incoming request. For example, making the horizontal stride one-half of the submesh width gives rise to twice as many candidate frames being examined. We simulated the FS strategy with various strides and observed that with shorter strides, the performance gains are always negligible, though substantially higher time complexities often result. This point is illustrated by the simulation outcomes of the FS strategy, with both horizontal and vertical strides being one-half of the submesh width

TABLE III
PERFORMANCE COMPARISON OF DIFFERENT STRIDES IN $M_2(256, 256)$

Performance measure	Completion time	Processor utilization	External fragment	NC†	NC††
$FS_{1,1}$	89830.3	47.7	33.7	16958.7	56937.3
$FS_{half,half}$	9716.8	44.1	34.0	44.4	38.8
FS	9922.1	43.2	33.5	16.4	13.7

† Number of comparisons per successful attempt.

†† Number of comparisons per attempt.

and height, listed in row $FS_{half,half}$ of Table III, where both side-length and residence time distributions are uniform. In the extreme case, when both horizontal and vertical strides equal 1 (the results are shown in row $FS_{1,1}$), time complexity becomes excessively high, but performance does not improve significantly. It is thus concluded that making strides equal to the side lengths of a requested submesh appears to be a good choice.

V. CONCLUSION

In this short note, we have introduced an efficient mesh processor allocation strategy based on frame sliding, termed the FS strategy, which can be applied to any mesh system and recognizes submeshes with arbitrary sizes at any location. The FS strategy allocates a submesh of the precise size to an incoming task, completely eliminating internal fragmentation. As a result, when compared with an earlier allocation strategy based on the buddy principle, this strategy leads to far better processor use and substantially reduces total time spent in finishing a batch of tasks. An effective implementation of the strategy that greatly simplifies the search of candidate frames for an available submesh is also presented. Simulation results confirm that our proposed strategy is consistently superior in allocating submeshes whose side lengths follow any of the four distributions, in terms of performance measures of interest. The mean search time per allocation attempt is fairly short, involving at most tens of simple comparisons. The FS strategy tends to have a larger search space than the buddy strategy, but its time complexity is kept low. It is observed from simulation that the CPU time spent in allocating 1000 tasks is less following the FS strategy than it is following the buddy strategy, when both strategies have the same search space. The FS strategy involves no higher complexity for a larger mesh system and is thus efficiently applicable to a system of any size. The performance of the FS strategy under different stride values is also pursued by simulation. Shortening strides gives rise to more candidate frames being examined and incurs higher time complexity. In the extreme case, where both horizontal and vertical strides are equal to 1, time complexity becomes excessively high, but performance improves insignificantly. It appears that making strides the same as side lengths of a requested submesh is a good choice.

We also have explored (but have not included in early sections, because of the space limitation) two approaches for reducing external fragmentation of our strategy: one by limiting the number of allowable submesh sizes and the other by using compaction [8], [9]. It is observed from simulation that our strategy is found to behave virtually the same as the buddy strategy if its allowable submesh sizes are limited to those permitted by the buddy strategy. As for compaction, not to mention its large migration overhead, it contributes no performance improvement, perhaps because of the essential nature of precise submesh allocation: reduced fragmentation, which is not further reducible. Compaction is thus unnecessary for our allocation strategy. We believe that the FS strategy is readily useful for any mesh connected system.

REFERENCES

[1] T. J. Fountain, K. N. Matthews, and M. J. B. Duff, "The CLIP7A image processor," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 10, pp. 310–319, May 1988.
[2] R. Alverson, *et al.*, "The Tera computer system," in *Proc. 1990 Int. Conf. on Supercomputing*, 1990, pp. 1–6.
[3] G. Zorpette, "Technology 1991: Minis and mainframes," *IEEE Spectrum*, pp. 40–43, Jan. 1991.
[4] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms.* Bristol, UK: Adam Hilger, 1981.
[5] K. Li and K.-H. Cheng, "A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system," *Proc. ACM Comput. Sci. Conf.*, Feb. 1990, pp. 22–28; also *J. Parallel Distrib. Computing*, vol. 12, pp. 79–83, May 1991.
[6] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, June 1977.
[7] M. Livingston and Q. F. Stout, "Parallel allocation algorithms for hypercubes and meshes," in *Proc. 4th Conf. Hypercube Concurrent Comput. Applications*, 1989, pp. 59–66.
[8] M.-S. Chen and K. G. Shin, "Task migration in hypercube multiprocessors," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, 1989, pp. 105–111.
[9] C.-H. Huang and J.-Y. Juang, "A partial compaction scheme for processor allocation in hypercube multiprocessors," in *Proc. 1990 Int. Conf. Parallel Processing*, vol. I, 1990, pp. 211–217.
[10] W. J. Dally and C. L. Seitz, "The Torus routing chip," *Distrib. Computing*, vol. 1, no. 3, pp. 187–196, 1986.

Performance Evaluation of an Efficient Multiple Copy Update Algorithm

T. V. Lakshman and Dipak Ghosal

***Abstract*— A well-known algorithm for updating multiple copies is the Thomas majority consensus algorithm. This algorithm, before performing an update, needs to obtain permission from a majority of the nodes in the system. In this short note, we study the response-time behavior of a symmetric (each node seeks permission from the same number of other nodes and each node receives requests for update permission from the same number of other nodes) distributed update-synchronization algorithm where nodes need to obtain permission from only $O(\sqrt{N})$ (N being the number of database copies) other nodes before performing an update. The algorithm we use is an adaptation of Maekawa's $O(\sqrt{N})$ distributed mutual exclusion algorithm to multiple-copy update-synchronization. This increase in the efficiency of the update-synchronization algorithm enhances performance in two ways. First, the reduction in transaction service time reduces the response time. Second, for a given arrival rate of transactions, the decrease in response time reduces the number of waiting transactions in the system. This reduces the probability of conflict between transactions. To capture the interaction between the probability of conflict and the transaction response time, we define a new measure called the *conflict response-time* product. Based on the solution of a queueing model we show that optimizing this measure yields a different and more appropriate choice of system parameters than simply minimizing the mean transaction response time.**

***Index Terms*—Transaction processing, concurrency control, replicated data, multiple copy update, transaction response time, conflict probability, transaction throughput, diameter-two sparse-graphs, finite projective planes.**

Manuscript received September 4, 1991; revised March 17, 1992.

The authors are with Bell Communications Research, 331 Newman Springs Road, Red Bank, NJ 07701.

IEEE Log Number 9214472.