Non-contiguous Processor Allocation Algorithms for Distributed Memory Multicomputers

Wanqian Liu, Virginia Lo,
Kurt Windisch & Bill Nitzberg

CIS-TR-94-11
April 1994

Department of Computer and Information Science
University of Oregon

Non-contiguous Processor Allocation Algorithms for Mesh-connected Multicomputers[1]

Virginia Lo, Kurt Windisch, and Wanqian Liu
Department of Computer and Information Science
University of Oregon, Eugene, OR 97403
Email: lo, kurtw, wliu@cs.uoregon.edu
Tel: (503) 346-4408
Fax: (503) 346-5373

Bill Nitzberg[2]
NAS Systems Division,
NASA Ames Research Center, Moffett Field, CA 94035
Email: nitzberg@nas.nasa.gov
Tel: (415) 604-4513

Abstract

Current processor allocation techniques for highly parallel systems are typically restricted to contiguous allocation strategies for which performance suffers significantly due to the inherent problem of fragmentation. As a result, message passing systems have yet to achieve the high utilization levels exhibited by traditional vector supercomputers. We are investigating processor allocation algorithms which lift the restriction on contiguity of processors in order to address the problem of fragmentation.

Three non-contiguous processor allocation strategies: Paging allocation, Random allocation and the Multiple Buddy Strategy (MBS) are proposed and studied in this paper. Simulations compare the performance of the non-contiguous strategies with that of several well-known contiguous algorithms. We show that non-contiguous allocation algorithms perform better overall than the contiguous ones, even when message-passing contention is considered. We also present the results of experiments on an Intel Paragon XP/S-15 with 208 nodes that show non-contiguous allocation is feasible with current technologies.

Keywords: resource management, scheduling, processor allocation, non-contiguous, fragmentation, mesh

1 Introduction

Highly parallel systems have the promise of outperforming traditional vector supercomputers in terms of price/performance on a wide range of individual applications. However, mainstream computing does not run "individual applications," but instead, supports a workload that is a diverse mix of large and small jobs. Workload price/performance is not only dependent on individual applications, but also on overall

[1]This research is sponsored by NSF grant MIP-9108528 and the Oregon Advanced Computing Institute
[2]Computer Sciences Corporation, NASA contract NAS 2-12961, Moffett Field, CA 94035-1000

system utilization. In this respect, traditional supercomputers are far ahead of parallel systems, the former usually achieving 98-99% utilization. The major price/performance obstacle to highly parallel systems is efficiently supporting workloads.

Processor allocation involves the selection of a set of processors for a given parallel job, with the goal of maximizing throughput over a stream of many jobs. Allocation techniques used in the majority of commercial parallel machines, as well as in the research community, have been restricted to **contiguous allocation** in which the processors are constrained to be physically adjacent. In addition, many systems also require that the allocated processors form a subgraph of the original architecture, specifically, subcube allocation in hypercubes and submesh allocation in meshes. The recurring theme found in all these studies is that performance suffers significantly due to internal and external fragmentation, or due to high overheads for allocation and deallocation. **Internal fragmentation** occurs when more processors are allocated to a job than it requests. **External fragmentation** exists when a sufficient number of processors are available to satisfy a request, but they cannot be allocated contiguously. Experimental evidence has shown that little improvement in performance can be realized by refinements of contiguous allocation algorithms [15]. As a result, recent research efforts have focused on the choice of job scheduling policies and their impact on contiguous allocation schemes.

Our research takes a different approach to overcoming the limitations of contiguous allocation. We are investigating processor allocation algorithms which lift the restriction of contiguity of processors in order to address the problem of fragmentation. As we shall show, **non-contiguous allocation** offers several significant advantages over contiguous schemes, including the elimination of internal and external fragmentation and low allocation and deallocation overheads. Furthermore, non-contiguous allocation is compatible with adaptive processor allocation schemes [23], in which a job may increase or decrease its allocation at runtime and supports straightforward extensions for fault tolerance.

We focus on the class of parallel systems that use variable partitioning to allocate jobs [10]. Independent user jobs arrive in the system, requesting a particular sized partition of the system's processors. If an arriving job can not be run immediately, due to a lack of free processors or other waiting jobs, the job is diverted to the system waiting queue. Before a waiting job can be executed, the **scheduler** must place the job at the head of the queue, and the **allocator** must determine that a partition can be constructed for the job from free processors in the system. When a job is ready to be run, the allocator assigns the job the needed processors, which may be contiguous or non-contiguous, depending on the allocation strategy used. Although messages from other jobs may pass through this new partition, the

new job holds these processors exclusively until it finishes running. At this time, it departs the system and its processors are freed for use by other incoming jobs.

Current communication technologies like wormhole routing [25] enable us to consider non-contiguous allocation, since the number of hops between processors is not the dominant factor determining message latency. However, we also note that non-contiguous allocation introduces potential problems due to message contention because the messages occupy more links, yielding potential communication interference with other jobs. Therefore, the most successful allocation scheme may be a hybrid between contiguous and non-contiguous approaches.

We compare the performance of three non-contiguous processor allocation strategies: Random, Paging, and the Multiple Buddy Strategy (MBS), with three well-known contiguous allocation schemes: Frame Sliding, First Fit, and Best Fit. These strategies represent a continuum with respect to degree of contiguity. As we have shown in [29], these strategies are also directly applicable to processor allocation in k-ary n-cubes which include the hypercube and torus.

One of these strategies, the Multiple Buddy Strategy (MBS), is an elegant algorithm we have developed that exploits the advantages of non-contiguity with respect to fragmentation while addressing the potential contention that may be introduced. MBS maintains contiguity within a small number of blocks to reduce interference between jobs. This strategy completely eliminates both internal and external fragmentation.

Our study shows that:

- Non-contiguous allocation strategies dramatically outperform contiguous allocation strategies with respect to fragmentation.

- The non-contiguous allocation algorithms perform better overall than the contiguous ones, even when message-passing contention is considered.

- Non-contiguous allocation strategies that take advantage of non-contiguity while providing some degree of contiguity exhibit the best performance.

- Non-contiguous allocation is feasible on present day multicomputers with wormhole routing.

Section 2 gives a brief summary of previous work in the area of processor allocation with emphasis on mesh topologies. Section 3 describes the three non-contiguous schemes Random, Paging, and the Multiple Buddy Strategy. Section 4 analyzes the performance of both contiguous and non-contiguous

strategies through simulation results. Section 5 presents the results of preliminary experiments on an Intel Paragon XP/S-15 with 208 compute nodes. Section 6 summarizes our results and discusses future work.

2 Previous Research Work

Our work on the problem of processor allocation takes into account a very rich body of related work by other researchers in the field. The Multiple Buddy Strategy proposed in this paper is an extension of the 2-D Buddy Strategy. Our simulations compare the performance of MBS and other new algorithms with the performance of the well-known contiguous strategies, Frame Sliding, First Fit and Best Fit, which were proposed by other researchers. Variations of these conventional algorithms, as well as an interesting algorithm with similarities to MBS, have been implemented on real mesh and hypercube systems. Work by Phillip Krueger, et al. [15], describes the performance limitations of all contiguous allocation schemes and thus motivates our investigation of non-contiguous approaches.

2-D Buddy

The *two-dimensional buddy strategy*, a generalization of the one-dimensional binary buddy system for memory management [26] [14], is proposed by Li and Cheng [17] as an allocation strategy for a mesh connected system. This strategy is only applicable to the situation when all incoming jobs are given square submeshes of size $n' \times n'$ and when the mesh itself is a square mesh of size $n \times n$, where both n' and n are exact powers of 2. The allocation and deallocation overheads of this strategy are both $O(\log n)$, which is relatively low compared to other strategies. However, it has several apparent drawbacks: (1) it can only be applied to square meshes; (2) it suffers from severe internal fragmentation, because a square submesh with side length of 2^i is always required; (3) it has significant external fragmentation.

Frame Sliding

Chuang and Tzeng proposed an improved strategy called the *frame sliding* strategy [7]. It is applicable to any mesh system and any shape of submesh request, thus it has no internal fragmentation. The frame sliding strategy examines the first candidate "frame" from the lowest leftmost available processor and slides the candidate frame horizontally or vertically by the stride of width or height of the requested submesh, respectively, until an available frame is found, or all candidate frames are checked. This

strategy has better performance than the 2-D buddy strategy. However, (1) it has higher allocation overhead, $O(n)$; (2) it suffers from large external fragmentation; and (3) it cannot recognize all possible free submeshes.

First Fit and Best Fit

In [31], Zhu proposed the *first fit* and *best fit* strategies, which can be applied for contiguous submesh requests of arbitrary sizes and have the ability to recognize all free submeshes in a system. These algorithms locate submeshes by constructing bit arrays indicating which processors have enough free neighbors to host the *base* node, the lower-left processor. These bit arrays can then be searched for the first available submesh (first fit) or submesh that best fits the request (best fit). Despite the attempt of best fit to reduce the probability of a fragmented system, both strategies suffer from significant external fragmentation. These algorithms both have allocation and deallocation overhead of $O(n)$.

Contiguous Allocation in Hypercube-based Machines

Many researchers have been involved in the design of allocation algorithms for machines whose interconnection network is a hypercube. Each job requests a subcube of size sufficient to accommodate the job's needs. The techniques that have been proposed include the Buddy and Gray Code Strategies [5] and later refinements for greater recognition [1][9] [13], These schemes were adapted for use in commercial hypercube-based systems such as the Intel iPSC/860 and nCUBE machines.

Contiguous and Non-contiguous Allocation in the Intel Paragon

The effectiveness of non-contiguous processor allocation has been demonstrated on a 400 node Paragon housed at the San Diego Supercomputing Center. The SDSC allocation algorithm [12][22] uses a modified 2-D Buddy Strategy that is similar to the Multiple Buddy Strategy described in this paper. Under the SDSC scheme, the mesh is configured into disjoint *node sets*; *node groups* are then constructed as groups of these node sets, where two or more groups may share node sets. A scheduling queue is associated with each node group along with the maximum size and duration of jobs that can be submitted to each queue. Within a given queue, scheduling is priority-based with an aging factor.

Once a job has been selected for allocation, the allocator selects free processors by scanning the set list associated with that queue. Node sets are examined in the order specified in the definition of the group. Processors are allocated from each set using a modification of the 2-D Buddy Strategy. The

first and largest block to be allocated is the anchor block. Subsequent blocks are chosen to be as close as possible to the anchor block, where *closeness* is the sum of the distances between the corresponding corners of the anchor block and the new candidate block.

Another allocation scheme developed for the Intel Paragon accepts three types of job requests: the number of processors needed, a contiguous rectangle, or a list of specified nodes and rectangles which are not necessarily contiguous. Rectangles are found using the Line-Sweep Strategy [4] which is adapted from an algorithm for the *Largest Rectangle Problem* in computational geometry.

Limits of Contiguous Allocation

Krueger et al. have shown in [15] that increasingly sophisticated contiguous processor allocation algorithms do not significantly influence the performance of hypercube systems. Their simulations of four well-known hypercube allocation strategies, which are equally significant for mesh systems, realized limited improvements despite the differing abilities of these algorithms to reduce fragmentation and recognize available subcubes. The barriers observed by Krueger et. al. are primarily a direct result of external fragmentation, which arises from the contiguity constraint. Although no statistics have been compiled for mesh systems, we believe the same trend will be exhibited under the assumption of contiguity. Thus, improved performance requires exploration of other alternatives, including scheduling policies [3] [21] [24] and the approach we propose: non-contiguous allocation.

3 Non-contiguous Allocation Strategies

3.1 Random Allocation Strategy

One of the most straightforward non-contiguous allocation strategies is the **Random allocation strategy**, under which a request for k processors is satisfied with k randomly selected processors. Both internal fragmentation and external fragmentation are eliminated since all jobs are assigned exactly the requested number of processors, if available. Since no contiguity is enforced under this strategy, we would expect much communication interference between jobs. The complexity of both allocation and deallocation is $O(k)$. We present Random allocation as a baseline against which to compare the performance of more sophisticated schemes described below.

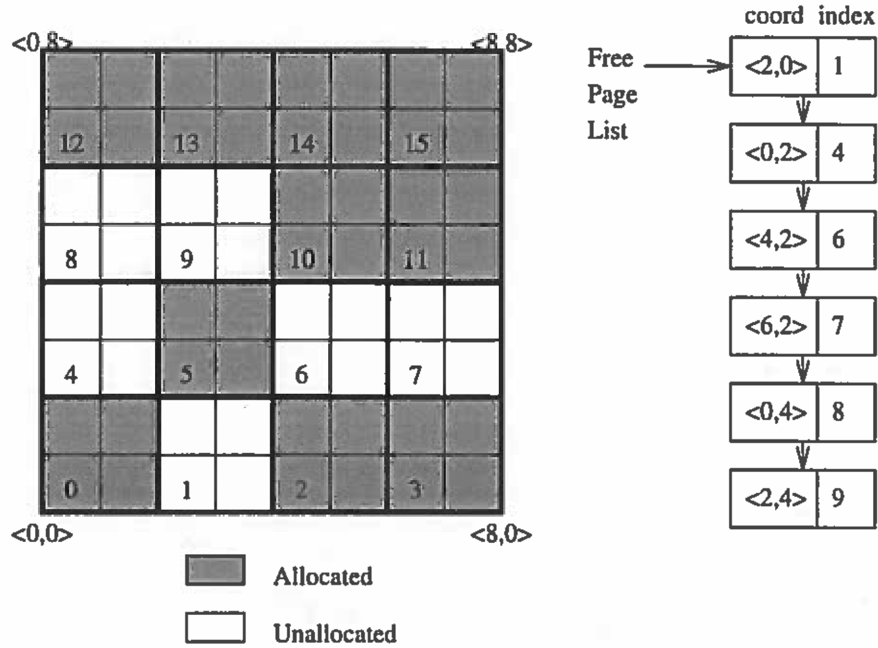

Figure 1: Example mesh and free page list for the $\text{Paging}_{rowmajor}(1)$ allocation strategy. Page blocks are labeled with the index given by the row-major indexing scheme.

3.2 Paging Allocation Strategy

Another simple non-contiguous allocation strategy is the **Paging allocation strategy**. Denoted as $Paging_{indexing_scheme}(page_size)$, it initially divides the entire mesh into *pages*, square blocks having side lengths of 2^{page_size}, which are the basic unit of allocation. A request for k processors is satisfied by allocating free pages until at least the number of processors requested have been allocated. The order in which pages are scanned is determined by the *indexing_schemes*, described below.

An ordered list is used to keep track of all unallocated pages (Figure 1). Each page entry contains its row and column indices and a unique order index, assigned by the indexing scheme. We consider the four indexing schemes shown in Figure 2: (a) *row-major* indexing, (b) *shuffled row-major* indexing, (c) *snake-like* indexing, and (d) *shuffled snake-like* indexing. Upon receiving a job request for k processors, $\lceil \frac{k}{2^{page_size} \cdot 2^{page_size}} \rceil$ entries are simply removed from the list and the corresponding pages are allocated to the job. The complexity of this allocation procedure is simply $O(k)$. When a job is deallocated, its ordered list of the $O(k)$ pages must be merged back into the ordered list of the $O(n)$ free pages, yielding a deallocation complexity of $O(n)$.

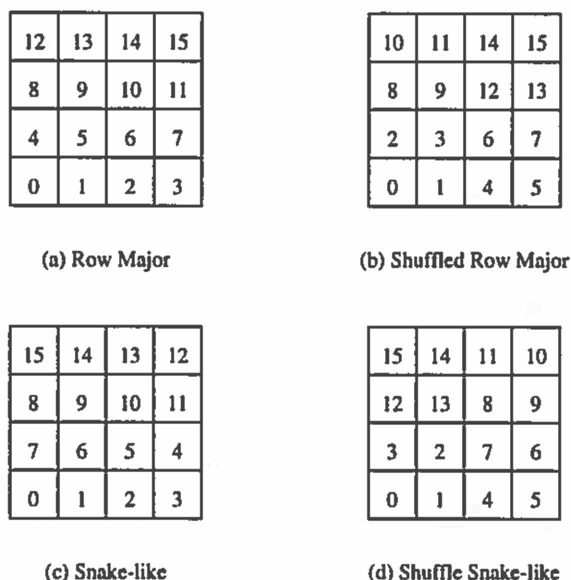

Figure 2: Meshes showing the order in which pages are selected by the four different indexing schemes used by the Paging algorithm.

Figure 1 shows an example of the Paging allocation, which uses the row-major indexing scheme and a page sizes of one (2×2 blocks). The *Free Page List* (FPL), is an ordered list that keeps track of all the unallocated pages. Assume a request for 6 processors arrives. The first two items in the FPL, the 1st page and 4th page, will be removed, and the corresponding eight processors will be allocated, resulting in 25% internal fragmentation.

Some degree of contiguity is maintained through the nature of the indexing schemes, since consecutive blocks will always be allocated adjacently if there are no other jobs allocated nearby. There will be neither internal nor external fragmentation when $page_size = 0$, but for $page_size \geq 1$, internal fragmentation may be introduced as a tradeoff for a greater degree of contiguity.

3.3 Multiple Buddy Strategy

We now introduce another non-contiguous allocation strategy, which we call the *Multiple Buddy Strategy* (MBS). It is an extension of the 2-D buddy strategy[17], which has both internal and external fragmentation problems. MBS eliminates fragmentation by applying the non-contiguous model to the mesh system, while still maintaining contiguity within individual blocks. The following scenarios show the problems exhibited by the 2-D buddy strategy and how they are resolved by the MBS strategy.

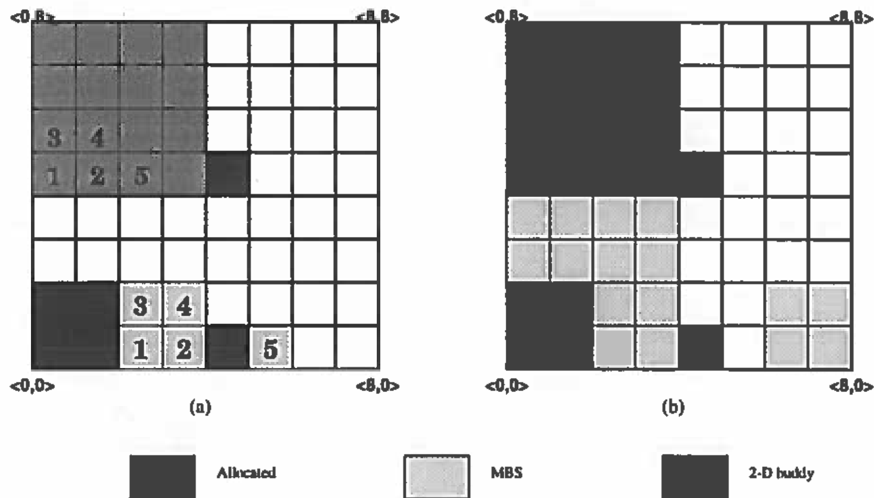

Figure 3: Eliminating system fragmentation using MBS: (a) allocation of a five processor job by 2D Buddy and MBS, and (b) allocation of a 16 processor job by MBS.

Square submeshes are represented by $< x, y, s >$, in which $< x, y >$ is the location of the lower leftmost processor, and s is the side length of the submesh of size $s \times s$.

In Fig 3(a), a mesh of size $2^3 \times 2^3$ has three allocated submeshes (represented by the black squares): $< 0, 0, 2 >$, $< 4, 0, 1 >$, and $< 4, 4, 1 >$. Assume that a job which needs 5 processors is submitted to the system. Since all submeshes have to be $2^i \times 2^i$ under the 2-D buddy strategy, a 4×4 submesh ($< 0, 4, 4 >$) will be allocated. In this case 11 processors in the submesh will be wasted during the lifetime of the job. This wastes 69% of the whole submesh, which is unacceptable in a real system. Under the MBS strategy, the exact number of processors will be assigned to the job. In the above case, two *blocks* will be assigned to the job: $< 2, 0, 2 >$ and $< 5, 0, 1 >$. Therefore, the internal fragmentation is eliminated.

Assume that the mesh shown in Fig 3(b) receives a request for 16 processors. Since a 4×4 block cannot be found in the mesh using the 2-D buddy strategy, the request will be put into a waiting queue. Because there are more processors in the mesh system than what was requested, external fragmentation results. The MBS strategy resolves this problem by breaking a large request into smaller blocks. In the above case, 4 blocks of size 2×2 will be assigned to the job. Since larger requests can always be broken down to 1×1 blocks, there will not be any external fragmentation.

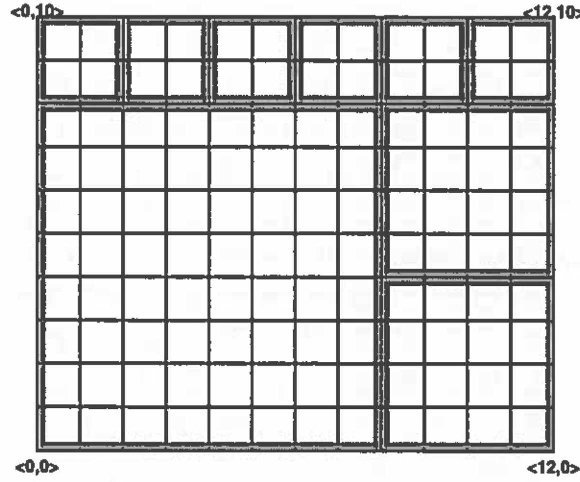

Figure 4: Initial blocks of a 12×10 mesh system

In general, a request for k processors is represented as a base 4 number of the form $k = d_m \times 2^m \times 2^m + ... + d_0 \times 2^0 \times 2^0$. MBS attempts to satisfy this request using blocks of sizes $2^m \times 2^m, \cdots, 2^0 \times 2^0$. If a block of a desired size is unavailable, MBS searches for a bigger block which it repeatedly breaks down into buddies until it produces blocks of the desired size. If that fails, MBS breaks a request for a block of size $2^i \times 2^i$ into four requests for blocks of sizes $2^{i-1} \times 2^{i-1}$. This process continues until the original request for k blocks is satisfied. The proposed MBS strategy can eliminate the fragmentation problems and be implemented efficiently. It is composed of the following 5 parts: system initialization, request factoring algorithm, buddy generating algorithm, allocation algorithm, deallocation algorithm.

3.3.1 System Initialization

System initialization is done only once at system startup time. At this time, the whole mesh system is divided into *initial blocks*, which are non-overlapped square submeshes with side lengths that are exactly powers of 2. The initialization process allows the strategy to be applicable to any size mesh system.

As shown in Fig 4, the 12×10 mesh system is divided into 9 initial blocks $< 0,0,8 >$, $< 8,0,4 >$, $< 8,4,4 >$, $< 0,8,2 >$, $< 2,8,2 >$, $< 4,8,2 >$, $< 6,8,2 >$, $< 8,8,2 >$, and $< 10,8,2 >$.

Definition 1 *A* **block** *and its* **buddies** *are defined recursively as follows:*

- *Any initial block is a* **block***;*

- *Each* **block** *is a square mesh and represented by* $< x, y, p >$, *where* $< x, y >$ *is the location of the lower leftmost processor, and p is the side length of the submesh;*
- *If* $< x, y, p >$ *is a* **block** *and* $p > 1$, *then* $< x, y, \frac{p}{2} >$, $< x + \frac{p}{2}, y, \frac{p}{2} >$, $< x, y + \frac{p}{2}, \frac{p}{2} >$, *and* $< x + \frac{p}{2}, y + \frac{p}{2}, \frac{p}{2} >$ *are* **blocks**, *and they are* **buddies** *of each other.*

The concept of *free block records* (FBR) extends the notion of the free block lists in the 2-D buddy strategy. $FBR[i]$ records the number ($FBR[i]$.*block_num*) of available blocks of size $2^i \times 2^i$ and an ordered list ($FBR[i]$.*block_list*) of the locations of such blocks. At startup time, information about the initial blocks will be kept in FBRs, where $FBR[i]$.*block_num* keeps the number of $2^i \times 2^i$ initial blocks and $FBR[i]$.*block_list* keeps a location list of such initial blocks. Another global variable $AVAIL$, the current number of available processors in the system, is initialized to the number of processors in the mesh system during the startup.

3.3.2 Request Factoring Algorithm

Any integer, n, has a base 4 representation, expressible as a sum $\sum_{i=0}^{\lfloor \log_4 n \rfloor} d_i \times (2^i \times 2^i)$ where $0 \leq d_i \leq 3$. Thus any legal job request can be accommodated by d_i blocks of size $2^i \times 2^i$. At most $\lceil \log_4 n \rceil$ distinct blocks are needed with a maximum of 3 blocks of a given size.

We define the *maximum distinct blocks* (MaxDB) of a given mesh system as $\lceil \log_4 n \rceil$, where n is the number of processors in the system. The factoring algorithm needs to take as input the job size and produces as output a request array (*Request_Array*[0..$MaxDB$]), where *Request_Array*[i] stands for the number of size $2^i \times 2^i$ blocks that the job needs. The algorithm essentially is an integer conversion algorithm, where *Request_Array*[i] is the ith digit in the base 4 integer representation of the job size.

3.3.3 Buddy Generating Algorithm

The buddy generation algorithm breaks a large block into several smaller blocks to satisfy the $2^i \times 2^i$ requests. It contains two phases. In the first phase, an available block is sought by examining the FBRs in increasing order of block size from $2^{i+1} \times 2^{i+1}$ to $2^{max} \times 2^{max}$. During the second phase, the block is repetitively broken down into smaller buddies until the desired size blocks are found. If no block is found in the search phase, as we shall see, the allocation algorithm will break the request down into smaller requests.

1. Search j from i+1 to search_level until a non-zero FBR[j].block_num is found. If such j can not be found, return Fail and quit.

2. Remove first block entry $< x, y, j >$ from FBR[j].

3. While $j > i$, add three blocks $< x + 2^{j-1}, y, j - 1 >$, $< x, y + 2^{j-1}, j - 1 >$, $< x + 2^{j-1}, y + 2^{j-1}, j - 1 >$ into FBR[j-1]. Set $j = j - 1$.

4. Add $< x, y, i >$ into FBR[i], return Succeed.

Algorithm 2: Buddy Generating Algorithm
(search for free block of size $2^i \times 2^i$)

3.3.4 Allocation and Deallocation Algorithm

1. If requested job size is greater than AVAIL, put the request into waiting queue and quit.

2. Factor job size into Request_Array. Set search_level to MaxDB.

3. Set i=MaxDB.
   While i $\geq$ 0 satisfy Request_Array[i] with steps 4 through 7.

4. If Request_Array[i] $\leq$ FBR[i].block_number, allocate the first Request_Array[i] blocks of FBR[i]. Jump to step 7.

5. Try to make more $2^i \times 2^i$ blocks using generate_buddy(i) and set search_level = i. If generate_buddy succeeds, allocate the first Request_Array[i] blocks of FBR[i], jump to step 7. Otherwise do step 6.

6. Give all available blocks of FBR[i] to the job.
   Add 4 $\times$ (Request_Array[i] - FBR[i].block_number) to Request_Array[i-1].

7. i=i-1

8. Decrease AVAIL by job size

Algorithm 3: Multiple Buddy Allocation Algorithm

The allocation algorithm includes two main parts. First, the request is factored and stored in Request_Array[i]. If possible, each request for a block of size i is allocated immediatedly from FBR[i]. Otherwise, an attempt is made to satisfy this request from a larger block by breaking it into smaller buddies. If that fails, the request will be broken down into 4 smaller requests, which are stored in Request_Array[i-1]. By the above algorithm, job requests are satisfied with the exact number of processors, and large requests can be accommodated by available smaller blocks; hence we can conclude that the

Multiple Buddy Strategy suffers from neither internal nor external fragmentation.

The deallocation procedure is essentially the same as that of the 2-D buddy strategy. Instead of returning just one block to the system, the MBS strategy needs to return all blocks owned by the job to the mesh system, and merge the buddies up to restore the larger blocks.

In a system with n processors, the total accumulated overhead of all calls to the generate_buddy procedure during job allocation is bounded by $MaxDB = \lceil \log_4 n \rceil$, and thus is $O(\log n)$. Since at most $O(n)$ blocks will be allocated, the total worst case complexity for allocation is $O(n) + O(\log n) = O(n)$. For deallocation, since the maximum number of buddy merges is $\frac{n}{4} + \frac{n}{16} + \cdots + 1 = \frac{1}{3}n = O(n)$, the overhead in the worst case will also not exceed $O(n)$.

3.4 Characteristics of Non-contiguous Strategies

3.4.1 Dispersal

The *dispersal ratio* reflects the degree to which non-contiguity of a given job allocation may increase contention in the whole system when the job's messages pass through nodes other than those allocated to it.

Definition 2 *For an allocation of k processors, let w and h be the width and the height of the minimal enclosing rectangle of the allocation,*

$$dispersal\ ratio = \frac{w \times h - k}{w \times h}$$

Weighted dispersal is defined as ($dispersal\ ratio \times k$).

For a non-contiguous allocation, the larger the weighted dispersal ratio is, the more the other jobs may interfere with it. As we show later in the paper, dispersal ratio is an important factor affecting system communication contention. For contiguous allocation, since the minimal enclosing rectangle of the allocation is the allocation itself, the dispersal ratio will always be 0, which represents no contention with other jobs.

3.4.2 Dynamic Optimality

An allocation strategy is *statically optimal*, as defined by Chen and Shin [5], if and only if the strategy can accommodate any valid sequence of job requests, in which each individual job requests no more

processors than remain free after allocating all preceding jobs. Thus, the total number of processors requested by the entire job stream must be less than or equal to the number of processors in the system. Static optimality is concerned only with job allocations, not with deallocations.

We extend this concept to define *Dynamic optimality*, which considers both allocations *and* deallocations of jobs. Further, we prove that several of the non-contiguous strategies that we have presented are dynamically optimal.

Definition 3 *A (legal)* allocation and deallocation sequence *in a system containing n processors is defined as a sequence of integers* $< r_1, r_2, \cdots, r_m >$, *where*

1. $|r_i| \leq n$ *for* $1 \leq i \leq m$, *and*

2. *for* j, $1 \leq j \leq m$, $0 \leq \sum_{i=1}^{j} r_i \leq n$.

Each positive integer r_i *represents an allocation of processors and each negative integer* r_i *represents a deallocation of processors. Condition (2) ensures that no allocation receives more processors than are available in the system, and that no deallocation releases more processors than have been allocated.*

Definition 4 *A processor allocation strategy is* dynamically optimal *when it can accommodate any allocation and deallocation sequence. If* r_i *in the sequence is limited to be non-negative, i.e. no deallocation can be involved, static optimality as defined in [6] is achieved.*

Theorem 1 *A processor allocation strategy is dynamically optimal if and only if it suffers from neither internal nor external fragmentation.*

Proof:

1. Soundness: Assume that an allocation strategy is optimal but suffers from either internal or external fragmentation, or both. Let $< r_1, r_2, \cdots, r_m >$ be an allocation and deallocation sequence. Suppose a fragmentation problem occurs when r_m is allocated. Then we can construct a contradiction by submitting a request for $n - (r_1 + r_2 + \cdots + r_m)$ processors to the system. The sum of these requests is n. The system cannot accommodate the last request because of fragmentation. Contradiction.

2. Completeness: Because it does not suffer from any fragmentation problem, the allocation strategy will always allocate the exact number of requested processors. Trivial to prove by induction.

Corollary 1 *Random allocation, $Paging_{indexing_scheme}(0)$ for any indexing_scheme, and the Multiple Buddy Strategy are dynamically optimal.*

4 Performance Analysis

We conducted two distinct sets of simulation experiments to analyze the performance of non-contiguous allocation strategies compared to contiguous ones: (1) fragmentation experiments and (2) message-passing experiments. Our discrete event simulator, ProcSimity [30], is a multicomputer simulator supporting experimentation with selected allocation and scheduling algorithms on architectures with a range of network topologies and several current routing and flow control mechanisms.

The fragmentation experiments model the arrival, service, and departure of a stream of jobs in a mesh-connected system using first-come, first-serve scheduling (FCFS). These high-level experiments focus on the effects of system fragmentation (both internal and external). Thus, the overhead of allocation and deallocation is ignored in the simulation, and the message-passing behavior of the algorithms is not modeled.

The message-passing experiments model the same stream of jobs, but at a much finer-grained level. The detailed message-passing behavior in a mesh with XY, wormhole routing is simulated down to the level of individual flits and message-passing buffers. The purpose of these experiments is to carefully examine the message contention introduced by non-contiguity.

In both sets of experiments, our focus is on the performance of contiguous versus non-contiguous allocation. Thus, we fixed the choice of scheduling algorithm, using the straightforward FCFS strategy throughout. We are conducting a separate set of experiments which analyze the performance of a range of scheduling algorithms in conjunction with both contiguous and non-contiguous allocation.

4.1 Fragmentation Experiments

The first set of experiments, studying the effects of fragmentation on system utilization and job response time, are modeled after the simulation experiments conducted in previous allocation strategy research [31] [7] [15]. In these experiments, jobs arrive, delay for an amount of time taken from an exponential distribution, and then depart. Message-passing is not modeled.

The contiguous allocation strategies simulated in these experiments are First Fit, Best Fit [31], and Frame Sliding [7].

From the non-contiguous strategies, we only present the results for the Paging algorithms. With respect to the fragmentation measured in this experiment, $Paging_{indexing_scheme}(0)$ performs identically, for all indexing schemes, to the Random and the Multiple Buddy strategies. The job request streams were modeled taking the submesh request sizes from the uniform, exponential, increasing, and decreasing distributions (See Appendix). The independent variable in these experiments was the system load, defined as the ratio of the mean service time to mean interarrival time of jobs. Higher system loads reflect the greater demands when jobs arrive faster than they can be processed. For example, under a system load of 1.0, jobs arrive as fast as they are serviced, on the average, and under a system load of 2.0, jobs arrive twice as fast as they can be serviced.

For each job size distribution in these experiments, we measure:

- *Finish Time* - the time required for completion of all the jobs.
- *System Utilization* - the percentage of processors that are utilized over time.
- *Job Response Time* - the time from when a job arrives in the waiting queue until the time it completes.

All simulations model a 32×32 mesh and run until 1000 jobs have been completed. Results reported for the fragmentation experiments represent the statistical mean after 10 simulation runs with identical parameters, and given 95% confidence level, mean results have less than 5% error. Additional details about the simulator are included in the Appendix.

Table 1 shows how well contiguous and non-contiguous algorithms handle a system saturated by job requests with job sizes taken from each distribution. Simulation results for a heavy system load of 10.0 are presented since, at this load, the system waiting queue is filled very early in the simulation, allowing each allocation strategy to reach its upper limits of performance.

In all cases, the non-contiguous Paging strategy, also representing the equivalent performance of Random and MBS, performs much better than First Fit, Best Fit, and Frame Sliding. With uniform, exponential, and decreasing distributions, simulations using non-contiguous allocation finish at least 57% faster than any of the contiguous algorithms. Further, Paging attains system utilizations from 69% to 77%, while none of the contiguous strategies are able to surpass 46%. Thus, in some cases, the non-contiguous strategies are able to utilize almost twice the number of available processors than the contiguous strategies. Improvement is less dramatic, though still significant, under the increasing

Algorithm	Job Size Distribution Uniform	Expon.	Incr.[a]	Decr.[b]
	Finish Time (simulation time units)			
$\text{Paging}_{rowmajor}(0)$	365.32	258.68	753.66	119.89
First Fit	582.01	429.57	882.94	237.90
Best Fit	573.79	428.72	883.08	231.92
Frame Sliding	608.02	457.88	885.56	267.40
	System Utilization (percent)			
$\text{Paging}_{rowmajor}(0)$	72.39	69.36	70.18	77.32
First Fit	45.96	41.68	60.15	39.15
Best Fit	45.70	41.64	60.30	39.28
Frame Sliding	43.39	38.47	59.84	34.30

[a] $P_{[1,16]} = 0.2, P_{[17,24]} = 0.2, P_{[25,28]} = 0.2, P_{[29,32]} = 0.4$

[b] $P_{[1,4]} = 0.4, P_{[5,8]} = 0.2, P_{[9,16]} = 0.2, P_{[16,32]} = 0.2$

Table 1: Fragmentation experiment results: Finish Time and System Utilization of each algorithm under different job size distributions for a heavy system load (10.0).

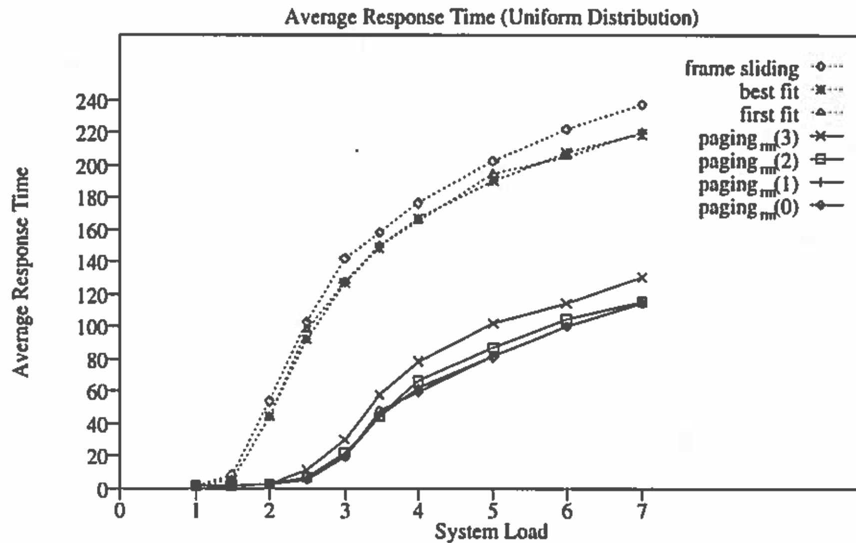

Figure 5: Average Job Response Time vs. System Load for the uniform distribution of job sizes.

distribution because the large job sizes tend to degrade the system towards the point where it can only service one job at a time.

Figures 5 and 6 show the average *job response times* for the uniform and decreasing job size distributions at varying system loads. For the uniform distribution, the system cannot maintain stability with the contiguous FF, BF or FS strategies past a system load of about 1.5. At this point, the job response

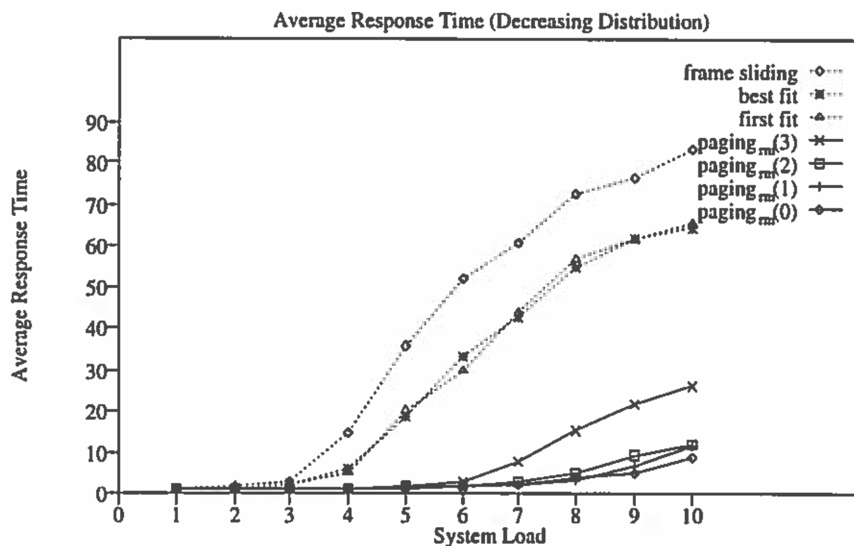

Figure 6: Average Job Response Time vs. System Load for the decreasing distribution of job sizes.

times for these algorithms begin to increase very sharply. However for the non-contiguous strategies (represented by Paging) with the uniform distribution, the system remains stable until a system load of about 2.5, where job response times begin to increase significantly, though not as dramatically as with the other strategies. Notice that as the page size increases, average response times increase for the Paging strategy, due to the internal fragmentation being introduced. The same trends can be seen in the job response times for the decreasing distribution of job sizes. At a system load of 3.0, the FF, BF, and FS strategies begin to lose stability and their job response times increase very rapidly. Yet, with Paging, the system performance does not begin to degrade significantly until a system load of about 8.0, and then job response times increase only slightly faster.

Notice that the curves in these graph begin to reach a plateau at high average response times. This is due to the fact that the simulated job stream is finite in length, and all response times are bounded by the overall finish time of the simulation. For an infinite job stream, the response time curves would continue to increase exponentially, resulting in near-infinite response times at high system loads. Regardless, the point of the initial dramatic increase would still occur at the same system load.

Figures 7 and 8 graph the *system utilization* for these same algorithms and job size distributions at varying system loads. Notice that peak utilization is reached just after the same load where the system was seen to become unstable in its response time graph. For the uniform distribution, the non-contiguous strategies (represented by Paging(0)) attain their peak utilizations at system loads of

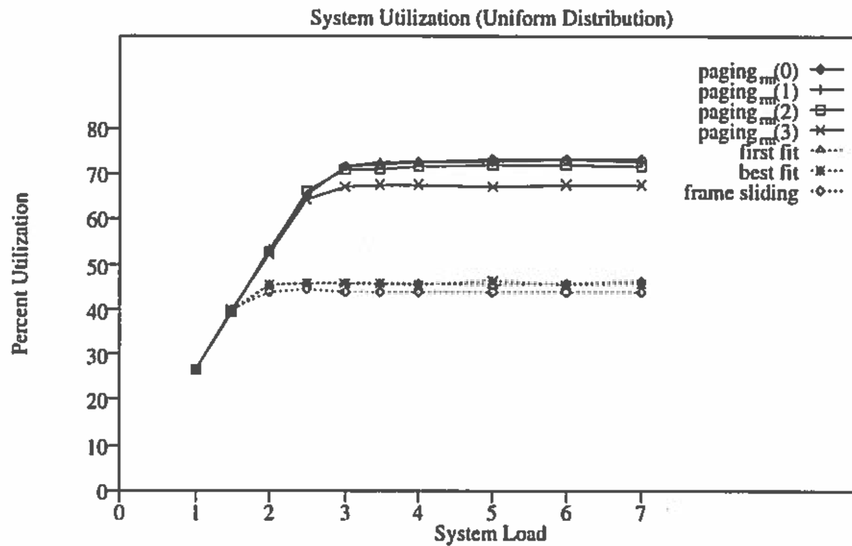

Figure 7: System Utilization vs. System Load for the uniform distribution of job sizes.

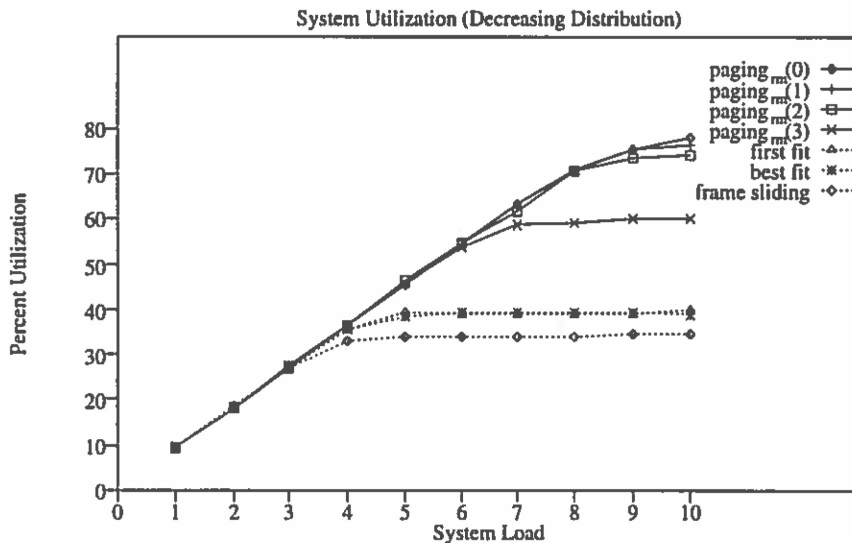

Figure 8: System Utilization vs. System Load for the decreasing distribution of job sizes.

about 3.0 with up to 72% utilization, whereas the contiguous strategies peak at system loads of about 2.0 with only 43 – 46% utilization. We again notice that as the page size increases, increasing internal fragmentation in the Paging strategy begins to decrease system utilization slightly. For the decreasing distribution, the non-contiguous strategies attains their peak utilizations at system loads of about 9.0 with about 77% utilization, whereas the others peak at system loads of about 4.0 with utilizations

around $34 - 40\%$.

The results measured in these experiments are all consistent with those reported by Zhu in [31] for the contiguous Frame Sliding, First Fit and Best Fit strategies.

These fragmentation experiments indicate that non-contiguous allocation is far superior to contiguous in terms of its ability to utilize the processors. Because non-contiguous allocation can always allocate a job if there are enough processors available, eliminating external fragmentation, it is shown to achieve higher system utilization. Thus, non-contiguous allocation allows for greater job throughput. However, these results ignore the increased communication contention that may be introduced as a result of non-contiguous allocation. Therefore, in order to validate non-contiguous allocation as a viable strategy, experiments must be performed to evaluate message-passing performance.

4.2 Message-passing Experiments

The second set of experiments measures message-passing contention and its effects on overall performance. The same simulator used in the fragmentation experiments was extended to model the sending and receiving of messages between the processors allocated to a job. Thus, rather than simply delaying for a given service time, processors allocated to the job communicate with each other according to a given communication pattern modeled from one of several common parallel applications. The communication pattern iterates until the number of messages sent within the job has reached its message quota, a value taken from an exponential distribution. This quota ensures that the job service times are independent of the job sizes. Once communication ceases, the job departs from the system and is deallocated.

The interconnection network is modeled by XY routing switches. These routing switches are connected by two uni-directional channels to neighboring switches in the mesh and to the corresponding processor elements. The flow control mechanism governing flit movement (flits are the smallest unit of data transmission in the network) is wormhole routing [25]. Messages originate from a processor element and their flits traverse the network in pipeline fashion to their destination processor. If the header flit of a packet is routed to a busy channel, that header flit and its trailing flits stop moving and block whichever channels they occupy in the network. This results in packet blocking time, due to contention, which can be measured in the simulation.

The message-passing experiments implement five communication patterns, in which n processors allocated to a job communicate with each other: all-to-all broadcast, one-to-all broadcast, the n-body

computation, fast fourier transform (FFT) [16], and multigrid (MG) from the NAS parallel benchmarks [2]. These cover many communications patterns used very frequently by highly parallel applications and provide a spectrum of message passing complexity ranging from $O(n)$ to $O(n^2)$.

All-to-all broadcast, in which every processor in a job sends to every other processor, loads the network very heavily with $O(n^2)$ simultaneous messages per iteration. In the one-to-all broadcast pattern, a randomly selected processor sends a message to every other processor in the job, loading the network with $O(n)$ simultaneous messages per iteration. The n-body communication pattern, described in [27], loads the network with $O(n)$ simultaneous messages per iteration, that are directed in a ring pattern. The 2D FFT pattern sends $O(n \log n)$ messages, and the NAS Multigrid sends $O(n)$ messages.

For simplicity and consistency, the internal mapping of the processes within each job is a row-major ordering of processors in each contiguously allocated block. This makes the latter three patterns very interesting cases, since the row-major mapping of these patterns is well-suited to contiguous allocations. These cases will be examined in more detail below.

Using this simulation model, experiments were conducted for each communication pattern using job request sizes generated from the uniform distribution. The network communication delay parameters were chosen such that the average job service times were great enough to result in high system loads, and thus, minimal system fragmentation. Experimental results are presented for Multiple Buddy Strategy, Random, Paging, and First Fit allocation. First Fit was chosen as a representative of contiguous allocation strategies since it has been shown to perform as well as or better than the others. For the Paging algorithm, we experimented with the four indexing schemes described in Section 3.2, but noticed only very slight variations in performance as a result. Therefore, we only present results from the row-major indexing scheme here. See the Appendix for more simulation details.

From each simulation run, we measure:

- *Finish Time* - the time required for completion of all the jobs. Finish time is a good measure of overall performance.

- *Packet Blocking Time* - the time that a packet is blocked in the network waiting for a channel to become free. Packet blocking time is a measure of the contention.

- *Message Latency* - the time that a message takes to be delivered to its destination from the time it is injected into the network. This includes its blocking time.

- *Weighted Dispersal* - the degree of non-contiguity for an allocation, approximating the percentage

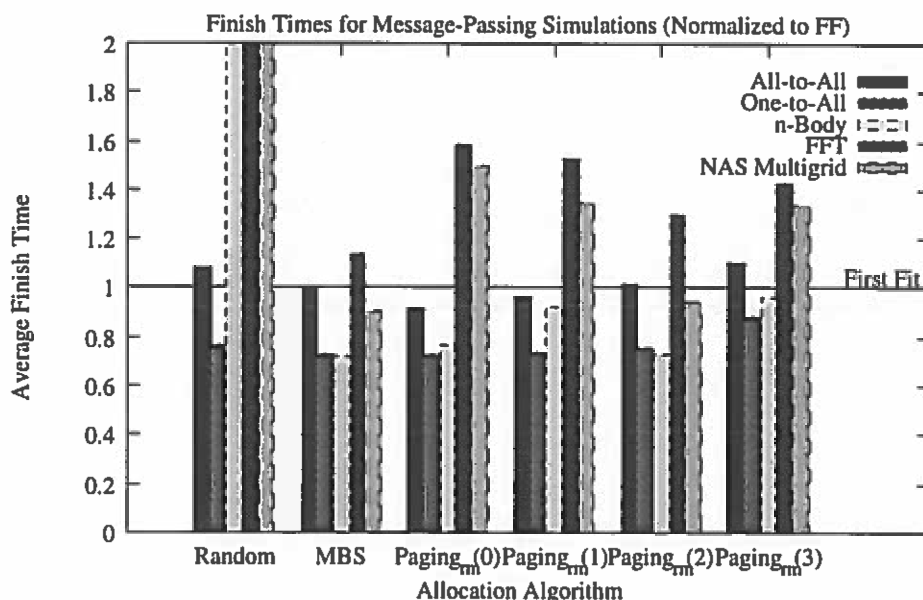

Figure 9: Chart showing the finish times of the six non-contiguous allocation strategies (along horizontal axis) and contiguous First Fit, for the five communication patterns tested. Results have been normalized to those of First Fit (finish time = 1.0) to show relative performance.

of links that are potential sources of contention. Dispersal is defined in Section 3.4.1 as the number of unallocated processors divided by the total number of processors in the smallest rectangle circumscribing all processors allocated to a specific job. The weighted dispersal, then, is the job's dispersal multiplied by the number of processors allocated to the job.

All message-passing simulations model a 16×16 mesh and run until 1000 jobs have been completed. Results reported represent the statistical mean after 10 simulation runs with identical parameters and, given 95% confidence level, the mean results have less than 5% error. Additional details about the ProcSimity simulator are included in the Appendix.

Figure 9 graphically summarizes the relative overall performance for the five allocation strategies and five communication patterns.

Table 2(a) shows the results of simulations for jobs executing the heavy All-to-All communication pattern. As expected, contiguous First Fit allocation shows much less contention (as seen in the packet blocking times and latencies) than most other non-contiguous strategies. However, even with this extremely heavy network traffic, the non-contiguous $\text{Paging}_{rowmajor}(0)$ strategy significantly outperforms any of the others, including First Fit. The overall performance of Paging degrades due to internal

(a) All-To-All Broadcast

Algorithm	Finish Time	Average Packet Blocking Time	Average Latency	Weighted Dispersal
Random	17228598.3	280.9944	351.9981	42.06
Multiple Buddy	15719664.7	282.7248	338.4383	28.17
$\text{Paging}_{rowmajor}(0)$	14486701.9	249.2457	307.2660	16.84
$\text{Paging}_{rowmajor}(1)$	15273589.4	285.5060	342.6990	20.14
$\text{Paging}_{rowmajor}(2)$	16142675.5	251.5254	305.8958	20.98
$\text{Paging}_{rowmajor}(3)$	17530161.1	179.2186	227.9724	11.60
First Fit	15848946.0	181.5983	230.1223	0.00

(b) One-To-All Broadcast

Algorithm	Finish Time	Average Packet Blocking Time	Average Latency	Weighted Dispersal
Random	1531265.6	2.7747	77.9199	42.07
Multiple Buddy	1443778.5	1.5189	61.4029	26.85
$\text{Paging}_{rowmajor}(0)$	1449696.8	1.2108	63.2294	14.72
$\text{Paging}_{rowmajor}(1)$	1458501.6	1.4242	62.6420	18.93
$\text{Paging}_{rowmajor}(2)$	1514414.0	1.4104	60.1752	20.25
$\text{Paging}_{rowmajor}(3)$	1755462.5	0.7292	54.8235	11.61
First Fit	1984068.8	0.3311	53.2524	0.00

(c) n-Body

Algorithm	Finish Time	Average Packet Blocking Time	Average Latency	Weighted Dispersal
Random	7332717.4	63.5616	136.7622	42.06
Multiple Buddy	2639727.9	3.6565	40.7655	29.90
$\text{Paging}_{rowmajor}(0)$	2824293.5	3.7822	34.6664	18.00
$\text{Paging}_{rowmajor}(1)$	3384394.3	13.3489	50.0536	21.11
$\text{Paging}_{rowmajor}(2)$	2654762.0	2.8538	39.6320	22.32
$\text{Paging}_{rowmajor}(3)$	3508907.9	1.4446	37.3785	12.11
First Fit	3652541.4	1.1394	36.8638	0.00

(d) 2D FFT

Algorithm	Finish Time	Average Packet Blocking Time	Average Latency	Weighted Dispersal
Random	707661.6	60.7329	133.5139	32.17
Multiple Buddy	268829.2	41.4761	80.8650	11.94
$\text{Paging}_{rowmajor}(0)$	372750.7	52.1801	95.4658	14.59
$\text{Paging}_{rowmajor}(1)$	359071.1	56.4589	100.1233	16.32
$\text{Paging}_{rowmajor}(2)$	304651.8	50.0448	91.3185	14.70
$\text{Paging}_{rowmajor}(3)$	334017.9	24.4993	59.2510	1.46
First Fit	234583.1	20.6071	54.7013	0.00

(e) NAS Multigrid Benchmark

Algorithm	Finish Time	Average Packet Blocking Time	Average Latency	Weighted Dispersal
Random	897196.5	60.5322	133.3742	32.17
Multiple Buddy	306114.5	22.1061	56.9394	12.02
$\text{Paging}_{rowmajor}(0)$	504543.9	65.5707	108.1267	14.89
$\text{Paging}_{rowmajor}(1)$	452825.5	47.8665	87.1907	16.30
$\text{Paging}_{rowmajor}(2)$	317950.8	18.6757	52.4804	14.58
$\text{Paging}_{rowmajor}(3)$	451686.6	26.6397	62.8348	1.62
First Fit	335899.5	25.1352	60.6687	0.00

Table 2: Message-passing experiment results for the five communication patterns.

fragmentation as the page size increases, despite a corresponding decrease in dispersal and contention. Although the performance of MBS is not significantly better than that of First Fit, its non-contiguity does not appear to degrade performance even under this communication intensive application. The reason the non-contiguous allocation algorithms do relatively well is that, although they suffer slightly more contention than contiguous allocation, the improvements in system utilization generally offset the increased communication overhead. It is also interesting to note that MBS and Paging allocation result in only moderate dispersal when compared to Random allocation.

Table 2(b) shows the results of simulations for jobs executing the One-to-All communication pattern. Under the lighter traffic load induced by this pattern, the contention effects seen in the experiments with All-to-All, are reduced. MBS and $\text{Paging}_{rowmajor}(0)$ perform extremely well, showing only moderate dispersal and contention. Contiguous allocation finishes last, performing even worse than non-contiguous Random allocation and taking 37% more time than MBS.

Table 2(c) shows the results of simulations for jobs executing the n-body communication pattern. The packet blocking times show that contiguous strategies have very little contention for this pattern, in which almost all communication occurs between adjacent neighbors when mapped by a row-major ordering. For MBS and Paging, contention increases somewhat, but still remains relatively low due to the fact that some degree of contiguity is maintained. This allows the ring communication to still be executed efficiently. Furthermore, it is interesting that the performance of Paging is best for 4×4 page sizes. The increased contention for MBS and Paging allocation is not significant enough to outweigh the improvements in system utilization. Random performs much worse than any of the others since it cannot take advantage of the regular ring communication in the n-body. Overall, MBS, $\text{Paging}_{rowmajor}(0)$, and $\text{Paging}_{rowmajor}(2)$ still finish much faster than either Random or contiguous allocation.

Tables 2(d) and 2(e) show the results of simulations for jobs executing two other communication patterns that are well matched to the mesh topology of the target machine: 2D FFT and NAS Multigrid. Due to restrictions imposed by the communication pattern, all job request sizes were rounded to the nearest power of two in these experiments. However, this biases the results since it has the side effect of decreasing system fragmentation for the contiguous strategies, improving their relative performance. Because both communication patterns are optimized to perform best in a mesh allocation whose side lengths are powers of two, they perform efficiently with contiguous allocation. However, since MBS allocates multiple such submesh blocks to each job, message passing is also surprisingly efficient under this allocation strategy. Therefore, with these highly mapping-sensitive applications, MBS performs

nearly as well or better than the contiguous strategies. Paging and Random allocation tend to perform very poorly since these mesh-optimized communication patterns do not usually map efficiently to their haphazard allocation patterns.

From these message-passing experiments, it appears that the MBS and Paging$_{rowmajor}(0)$ allocation strategies outperform both contiguous and Random allocation, with higher system utilization and increased job throughput, reflected in their faster finishing times. As hybrid strategies, they take advantage of the greater flexibility offered by non-contiguous allocation while still maintaining a degree of contiguity, as reflected in their moderate dispersal values. The packet blocking times indicate that this pays off in performance because contention is reduced in comparison to Random allocation. We would expect contention effects to be even less significant in real parallel applications, where only a portion of the total execution time is spent in communication.

These experiments also show that the performance of different allocation algorithms varies significantly with the type of applications executed. The best choice of allocation algorithms might vary from site to site, depending on the communication characteristics of the applications most commonly executed. In this respect, the MBS strategy seems to offer the greatest flexibility, greatly improving the performance of many types of applications while not degrading performance significantly for the rest.

Investigation of non-contiguous allocation requires much more exhaustive experimentation. We are currently conducting simulation experiments that more closely model real parallel applications, looking at the effects of message size, computation/communication ratio, and emerging routing technologies. In addition to these simulation studies, which allow us to address future architectural developments, we are conducting experiments with real production parallel computers.

5 Contention on Real Systems

As a first step in evaluating the feasibility of non-contiguous allocation on real parallel machines, we measured worst-case contention and ran benchmarks on the Intel Paragon XP/S-15 at the Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center. The NAS Paragon is a distributed memory multicomputer with 208 compute nodes connected by a 175 megabyte per second bi-directional mesh, with wormhole, XY routing. In addition to using the operating system supplied by Intel, Paragon OS release 1.1, we ran worst-case contention tests under SUNMOS, a minimal operating system developed by Sandia National Labs and the University of New Mexico [19].

One would expect message contention to have a noticeable impact on performance; however, we were unable to measure any performance degradation on real applications due to contention. We ran two, four, and six copies of selected NAS Parallel Benchmarks [2] simultaneously and found no measurable performance difference compared with running the benchmarks one at a time on a dedicated system. For example, we partitioned the compute nodes based on diagonals, splitting the machine into four partitions (64, 64, 32, and 32 nodes) in a checkerboard pattern—this pattern was designed to encourage message contention between applications. We ran the best Intel-supplied implementations of selected NAS Parallel Benchmarks (FFT, MG, and CG) in a loop on each partition. The largest performance variation was for FFT, which varied by less than 2% over 13 runs. We saw similar results for other partitioning schemes.

In order to better quantify the effects of message contention on the Paragon, we developed a simple worst-case contention generating program, `contend`. To force contention on the XY routed Paragon, multiple messages must be sent over the same physical network link. It suffices to allocate a row of nodes, pairing nodes from the middle outward, and have each pair exchange messages. With this configuration, all messages must traverse the network link in the center. Since the Paragon hardware supports bi-directional communication, only one direction was used to measure contention; zero length replies were sent in the other direction. This also allowed us to extend the row of nodes around a corner, forming an inverted "L", allowing more pairs to be simultaneously active while still sharing a single network link in the center. We ran `contend` on up to nine pairs of communicating nodes, with message sizes ranging from 0 to 64 kilobytes.

Under the native Paragon operating system (Figure 10), virtually no contention is noticeable (RPC times are flat) through six pairs of communicating nodes. Starting with seven pairs, contention begins to slow message-passing performance, but only for messages larger than 16 kilobytes. This surprising result is an artifact of the current release of the operating system, and explains our inability to measure contention while running multiple simultaneous copies of the NAS benchmarks. Although the Paragon hardware supports 175 megabytes per second bandwidth, the current release of the operating system (R1.1) delivers only about 30 megabytes per second. The hardware has more than enough excess bandwidth to support about six pairs of communicating nodes without any noticeable contention ($6 \times 30 = 180$).

We then ran `contend` on the Paragon under the SUNMOS operating system (Figure 11), which delivers 170 megabytes per second bandwidth, nearly peak speed. With the anomalous operating system

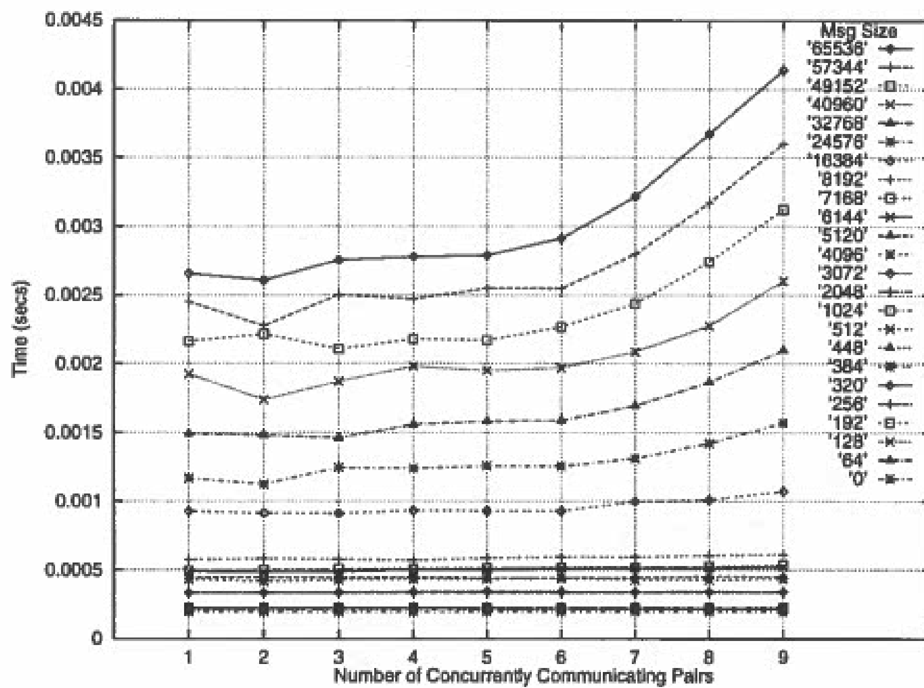

Figure 10: Worst Case Contention on the Intel Paragon (Paragon OS R1.1)

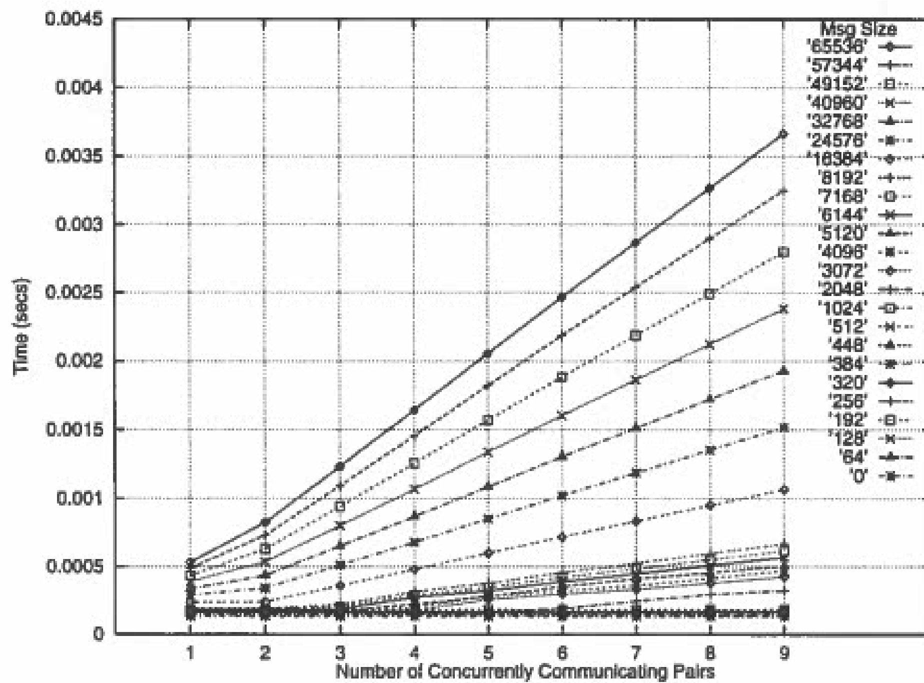

Figure 11: Worst Case Contention on the Intel Paragon (SUNMOS S1.0.94)

behavior eliminated, the effects of contention are significant with only two pairs of communicating nodes, and increase linearly with the number of pairs. However, small messages (less than one kilobyte) appear to be little effected by contention, even with nine pairs of communicating nodes.

Although preliminary, and by no means exhaustive, our experiments show two interesting facts about contention on the Paragon: current operating system overhead subsumes contention effects, and small messages do not cause contention. The poor operating system performance will likely be corrected in future releases of the Paragon OS, but the level of contention for of small messages will likely remain low. VanVoorst, et. al. [28], measured the workload of the Intel iPSC/860 system at NAS for ten days, and found that 87% of all messages are, in fact, one kilobyte or less. So, at least for a class of scientific applications, large messages may not be a significant issue. This empirical data is encouraging, and supports the notion that a purely non-contiguous allocation strategy may run into contention effects with large messages, but a purely contiguous strategy is also unnecessary.

6 Conclusions

This paper investigates non-contiguous processor allocation strategies as a method for improving performance in message-passing multicomputers. Contiguous allocation schemes suffer from low utilization due to serious fragmentation problems, and experiments have demonstrated that there is a limit to the amount of improvement that can be achieved for contiguous allocation.

We study three non-contiguous processor allocation strategies for mesh-based multicomputers: the Multiple Buddy Strategy, Paging, and Random allocation. We compare their performance with that of several well-known contiguous allocation schemes: First Fit, Best Fit, and Frame Sliding. Two sets of simulations experiments were conducted to analyze the performance of non-contiguous allocation compared to contiguous allocation. In addition, empirical experiments on a 208 node Intel Paragon XP/S-15 were conducted to investigate the performance of three of the NAS parallel benchmarks under non-contiguous allocation.

To summarize our results:

- **Non-contiguous allocation strategies dramatically outperform contiguous allocation strategies with respect to fragmentation.** As a result system utilizations for non-contiguous schemes reach as high as 77% compared to utilizations of 34% to 46% for contiguous schemes when message-passing contention is not considered.

- **The non-contiguous allocation algorithms perform better overall than the contiguous ones, even when message-passing contention is considered.** The increased contention due to non-contiguous allocation is not as serious as the fragmentation effects of contiguous allocation. Although contiguous allocation performs slighly better for some highly mapping-sensitve communication patterns, the non-contiguous algorithms, MBS and Paging(0), exhibited the best across-the-board performance due to their superior utilization of processors.

- **Non-contiguous allocation strategies that take advantage of non-contiguity while providing some degree of contiguity exhibit the best performance.** The fully contiguous First Fit algorithm and the fully non-contiguous Random allocation algorithm exhibited the worst performance, while the best performance was achieved by the two hybrid algorithms, the Multiple Buddy Strategy and Paging allocation.

- **Non-contiguous allocation is feasible on present day multicomputers with wormhole routing.** Current operating system overhead on the Paragon XP/S-15 (Paragon OS R1.1) subsumes the contention effects under non-contiguous allocation. Even under improved operating systems, the contention effects are negligible for small messages (less than one kilobyte). A sample workload at NASA NAS shows 87% of all messages to be one kilobyte or less.

Our study shows that non-contiguous strategies yield dramatic improvements in system performance because they eliminate both internal and external fragmentation. Furthermore, the amount of contention introduced by non-contiguity can be limited so their effects on utilization and throughput are minimized. We conclude that non-contiguous allocation provides a new approach that will help highly parallel systems achieve excellent price/performance ratios in a high demand, multi-user environment.

These results are being further evaluated in the larger framework of the combined effects of scheduling with allocation. As discussed earlier, the choice of scheduling algorithm is also a crucial factor in performance. Our ongoing work involves experimentation with a spectrum of scheduling and non-contiguous allocation algorithms to determine the performance gains attributed to each alone, as well as the best performance achievable through selection of combined scheduling/allocation strategies.

7 Acknowledgments

We would like to thank Yahui Zhu for providing us with his code for the First Fit, Best Fit, and Frame Sliding allocation algorithms, and Reagan Moore and Michael Wan for descriptions of the allocation policies at the San Diego Supercomputing Center.

References

[1] A. Al-Dhelaan and B. Bose. A new strategy for processor allocation in an n-CUBE multiprocessor. In *Proceedings of the International Phoenix Conference on Computers and Communication*, pages 114–118, March 1989.

[2] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results 3-94. Technical Report Technical Report RNR-94-006, NASA Ames Research Center, Moffett Field, CA 94035-1000, March 1994.

[3] S. Bhattacharya, L. M. Ni, and W. Tsai. Lookahead processor allocation in mesh-connected massively parallel computers. Technical report, University of Minnesota (Bhattacharya and Tsai) and Michigan State University (Ni), 1993.

[4] J. Carbajal, D. Cameron, and Y. Zhu. Node allocation for the mesh-connected paragon supercomputer. Personal communication, April 1994.

[5] M. Chen and K. G. Shin. Processor allocation in an n-CUBE multiprocessor using gray codes. *IEEE Transactions on Computers*, C-36(12):1396–1407, December 1987.

[6] M. Chen and K. G. Shin. Processor allocation in an n-CUBE multiprocessor using gray codes. *IEEE Transactions on Computers*, C-36(12):1396–1407, December 1987.

[7] P. Chuang and N. Tzeng. An efficient submesh allocation strategy for mesh computer systems. In *1991 International Conference on Distributed Computer Systems*, pages 256–263, 1991.

[8] W. J. Dally and C. L. Seitz. The torus routing chip. *Distributed Computing*, 1986.

[9] S. Dutt and J. P. Hayes. Subcube allocation in hypercube computers. *IEEE Transactions on Computers*, 40(3):341–352, March 1991.

[10] D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790 (87657), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1994.

[11] M. L. Fulgham and L. Snyder. Performance of chaos and oblivious routers under non-uniform traffic. Technical report, University of Washington, 1993.

[12] Intel Corp. Paragon network queuing system manual. October 1993.

[13] J. Kim, C. R. Das, and W. Lin. A top-down processor allocation scheme for hypercube computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):20–30, January 1991.

[14] D. E. Knuth. *Fundamental Algorithms: Volume I.* Addison Wesley, 1973.

[15] P. Krueger, T. Lai, and V. A. Dixit-Radiya. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):488–497, May 1994.

[16] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin Cummings Publishing Company, Inc., 1994.

[17] K. Li and K. Cheng. A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. *Journal of Parallel and Distributed Computing*, 12:79–83, 1991.

[18] W. Liu, V. M. Lo, K. Windisch, and B. Nitzberg. Non-contiguous processor allocation algorithms for distributed memory multicomputers. In *Proceedings of Supercomputing 1994*, pages 227–236, 1994. Best student paper award.

[19] B. MacCabe and K. McCurley. SUNMOS for the Intel Paragon. unpublished, 1993.

[20] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, 1987.

[21] P. Mohapatra, C. Yu, C. R. Das, and J. Kim. A lazy scheduling scheme for improving hypercube performance. In *International Conference on Parallel Processing*, 1993.

[22] R. Moore and M. Wan. Intel Paragon allocation algorithms. Personal communication, 1995.

[23] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of Supercomputing 1993*, pages 824–833, 1993.

[24] B. Narahari and R. Krishnamurti. Scheduling independent tasks on partitionable hypercube multiprocessors. In *Proceedings of the International Parallel Processing Symposium*, 1993.

[25] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Trans. Computers*, 1993.

[26] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[27] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[28] B. VanVoorst, S. Seidel, and E. Barszcz. Profiling the communication workload of an iPSC/860. In *Scalable High Performance Computing Conference*, May 1994.

[29] K. Windisch, V. M. Lo, and B. Bose. Contiguous and non-contiguous processor allocation algorithms for k-ary n-cubes. In *Proceedings of the International Conference on Parallel Processing*, 1995. To appear.

[30] K. Windisch, J. V. Miller, and V. Lo. Procsimity: an experimental tool for processor allocation and scheduling in highly parallel systems. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 414–421, February 1995.

[31] Y. Zhu. Efficient processor allocation strategies for mesh-connected parallel computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.

A Simulation Details

Scheduling Policy - The system waiting queue implements the traditional first-come, first-serve (FCFS) policy.

Allocation Algorithms - We implemented the three non-contiguous algorithms as described in Section 3. Zhu's implementations were used for three contiguous allocation algorithms, First Fit, Best Fit, and

Frame Sliding [31].

Job Request Sizes - Job request submesh dimensions are randomly generated from one of four different distributions: uniform, exponential, increasing or decreasing. In the uniform distribution, $U[a, b]$, the widths and heights of each job are uniformly distributed over the range $[a, b]$. In the exponential distribution, job side lengths are exponentially distributed with a mean of half the side length of the entire mesh. In this case, there are many small jobs and fewer large ones. Increasing and decreasing use a distribution called uniform on intervals. The range of job side lengths is divided into four intervals and a probability is assigned to each interval. A value is then generated from the uniform distribution on that interval. $P_{[a,b]}$ denotes the probability that the side length will fall within the range [a,b]. The increasing distribution has very few small jobs and many large ones and is described as $P_{[1,16]} = 0.2, P_{[17,24]} = 0.2, P_{[25,28]} = 0.2, P_{[29,32]} = 0.4$. The decreasing distribution has many small jobs and very few large ones and is described as $P_{[1,4]} = 0.4, P_{[5,8]} = 0.2, P_{[9,16]} = 0.2, P_{[16,32]} = 0.2$.

Communication Network - The processor interconnection network is a mesh with no wrap-around edges. Routing is performed through XY, wormhole routing switches modeled roughly after the torus routing chip [8]. The routing switches are connected to each other and their corresponding processor elements via two uni-directional channels. Simulated packets contain 8 flits. As measured in [11], flits take 1 network clock cycle to move between routing switches, and two cycles to be routed to the next channel through the switch. This is sufficiently large to result in high job service times, and thus high system loads ($\gg 10.0$).

Communication Patterns - We model five communication patterns: all-to-all broadcast, one-to-all broadcast, n-body [27], 2D FFT [16], and the NAS Multigrid benchmark [2]. All communications patterns execute until the total number of one packet messages sent equals a message quota drawn from an exponential distribution. The mean of this distribution is set to the number of messages sent in one iteration of the communication pattern for the average sized job. Within each communication pattern barrier synchronization occurs between iterations. A job completes its execution only when its message quota has been reached and all messages are received. Jobs never delay to simulate computation.

Instrumentation - Each simulation run reports average response time, finish time, average allocation dispersal, average service time, average packet blocking time, and system utilization. Measurements are

averaged over multiple replicated runs, and output analysis is performed assuming a 95% confidence level according to the techniques in [20]. In all measurements, the overhead of allocation and deallocation is ignored.

Fragmentation Experiments - In the fragmentation experiments, jobs only delay for an exponentially distributed service time with mean of 1.0 time units. Jobs arrive in the waiting queue with exponentially distributed interarrival times having a mean of the inverse of the system load and simulations are run for system loads from 0.5 to 10.0. The system modeled is a 32×32 mesh. Each simulation consists of 10 replicated runs of 1000 jobs each. This experiment is duplicated for each of the jobs side length distribution: uniform ($U[1, 32]$), exponential (mean 16), increasing, and decreasing.

Message-passing Experiments - In the fragmentation experiments, jobs are serviced for as long as it takes to execute their communication pattern for their message quota. Jobs arrive in the waiting queue with exponentially distributed interarrival times having a mean of 0.1 time units. The average interarrival and service times observed with these parameter yields a very high system load, such that each allocation algorithm should reach its upper limits of performance. The system modeled is an 16×16 mesh, connected as described above. Simulations are run for each communication pattern with differing allocation strategies. Job side length are taken from the uniform distribution ($U[2, 8]$). Each simulation consists of 10 replicated runs of 1000 jobs each.