I/O Processor Allocation for Mesh Cluster Computers

Pangfeng Liu
Department of Computer Science
and Information Engineering
National Taiwan University
pangfeng@csie.ntu.edu.tw

Chun-Chen Hsu Jan-Jan Wu
Institute of Information Science
Academia Sinica
wuj@iis.sinica.edu.tw

Abstract

As cluster systems become increasingly popular, more and more parallel applications require need not only computing power but also significant I/O performance. However, the I/O subsystem has become the bottleneck of the overall system performance for years due to slower improvement of the second storage devices. In recent years parallel I/O has drawn an increasing attention as a promising approach to eliminate this bottleneck. To improve I/O efficiency of a cluster system computation tasks must be carefully assigned to processors, so that the communication overheads within the group the processors of the task, and those I/O traffics that connect processors of the task to I/O system are both optimized. Earlier processor allocation strategies considered the optimization of communication traffic or I/O traffic only. Since both the communication and I/O traffic can cause network contention, we develop a set of binary tree *based algorithms to address the issues of both communication and I/O traffics simultaneously. The experimental results indicate that for tasks that have different mixture of communication and I/O traffics, our algorithms have very good performance in terms of overall parallel I/O efficiency. We also developed two mathematical evaluating criteria – "compactness" and "spatial compactness", to determine the fitness of allocation algorithms in terms of geometrical adjacency of processors.*

1. Introduction

Parallel I/O subsystems in a distributed-memory parallel system are typically configured as follows. The compute nodes and I/O nodes are connected by an internal network. Only the I/O nodes have links to disks. For a read request, the I/O nodes reads the data from disks and then send them over the network to the clients (compute nodes). For a write request, the compute nodes send the data over the network to the I/O nodes, and then the I/O nodes write them to the disks. As the data and I/O demands of applications in many fields increase, vast amount of data must be moved – both between compute nodes and I/O nodes, and among compute nodes, hence inducing two kinds of network traffic: the *communication-based* traffic induced by data exchange among compute nodes, and the *I/O-based* traffic induced by data exchange between compute nodes and I/O nodes. As the traffic on the links in the network increases, *network contention* becomes a critical problem, causing significant delay in communication and I/O.

We define the *processor allocation* problem as selection of available processors in the system for incoming parallel jobs, with the goal of minimizing certain cost measurements. A number of research works have demonstrated that the spatial layout of jobs can significantly affect network contention and ultimately overall performance. Spatial layout refers to the location of a job's compute nodes relative to each other and relative to the location of the I/O nodes. There are two types of allocation algorithm in the literature – contiguous and non-contiguous allocations. Processors allocated by a contiguous allocation method are physically adjacent in the layout of cluster topology, and non-contiguous method does not have to following this constraint.

The performance of traditional contiguous allocations is affected significantly by internal and external fragmentation. Internal fragmentation occurs when more processors are allocated than a task requested. External fragmentation occurs when there are sufficient processors but they could not be allocated to a task because they are not all physically adjacent. Fragmentation lowers system utilization. Non-contiguous allocation could eliminate both internal and external fragmentation and improve the system utilization. The non-contiguous allocation, however, also introduces the potential problem due to network contention caused by inter-task communication interference, i.e., communications from several tasks may want to use the same network link at the same time. Obviously there is a trade-off between system utilization due to fragmentation and job

Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)
0-7695-2281-5/05 $20.00 © 2005 IEEE

response time due to network contention. Many researchers have studied various efficient contention-free processor allocation strategies. In these works, the only cause of network contention is communication between compute nodes.

The *Multiple Buddy Strategy* (MBS) is an allocation algorithm that takes advantages both of contiguous and non-contiguous allocations. MBS exploits the advantage of non-contiguous allocation by dividing the request into a small number of sub-requests. The sub-requests are not necessarily allocated adjacently, so that the flexibility of non-contiguous allocation is maintained. On the other hand, each sub-request is allocated contiguously so that communication overhead is reduced.

Lo. et al. introduced non-contiguous with the goal of maximizing system throughput over a stream of many tasks [4]. They proposed several non-contiguous allocation algorithms which yield high system utilization and better performance comparing with contiguous allocation algorithms [4].

Leung uses a one dimensional non-contiguous allocation strategy to achieve general processor locality, so that [3]. First, a linear ordering on the processors is established via a space-filling curve. If processors are close to each other in the space-filling curve order, the processors that are close to each other in the cluster topology. A *Hilbert curves* is used when the cluster topology is 2-D or 3-D mesh [3]. The algorithm tries to find a contiguous interval of free processors large enough for the task. If several contiguous intervals can be allocated to the task, the algorithm chose an interval according to First-Fit Allocation, Best-Fit Allocation and Sun-of-Squares Allocation. If a contiguous interval cannot be found, the task is allocated across multiple intervals. The algorithm chooses the allocation that minimizes the *span* of the task – namely the distance in the one-dimensional ordering of processors, which servers as a measurement of the locality of the allocated processors. Experimental results indicate that space-filling curves with one-dimensional allocation strategy improve processor locality. These allocation strategies were later implemented in the release of the Cplant System Software at Sandia National Laboratory.

Allocations algorithms described above have good performance when we consider only communication traffic. However, the efficiency of parallel I/O is also a critical problem for parallel computing. As a result it may not be sufficient to consider only communication traffic in the treatment of communication optimization.

There are several research issues in the improvement of parallel I/O efficiency, including computer architecture, system software, parallel file systems, parallel I/O scheduling and so forth. One of the research direction on parallel I/O is to study how parallel I/O performance is affected by the *spatial layout* of tasks on a mesh cluster [5, 6, 1, 2], much like the processor allocation problem described earlier in the context of optimizing communication traffic.

Mache et. al. [6] showed that on a mesh-connected cluster the spatial layout of a task directly affects network contention due to I/O traffic and thus has significant impact on parallel I/O performance. There is serious contention for write traffic when the I/O nodes lie vertically on the left or right side of the mesh, and serious read traffic when the I/O nodes lie horizontally, due to the use of XY-routing in the mesh cluster topology. As a result allocation algorithms should take into account the relative percentage of read/write operations to improve parallel I/O efficiency. Considering I/O traffic only, Mache et. al proposed a Parallel Layout Allocation Strategy (PLAS) to assign compute nodes so that they lie parallel to the I/O nodes. It is concluded in [6] that the *parallel* spatial layout is superior to block-based and orthogonal layouts, and it is desirable that one half of the compute nodes are placed above the link in the middle of the I/O nodes and one half are placed below.

From the discussion above we conclude that it is vital to "spread out" the compute nodes allocated to a task so that messages going I/O nodes do not interfere with each other, and on the other hand, it is equally important to "compact" the compute nodes so that the communication traffic among processors allocated to a task is limited within themselves. However, those algorithms described in the literature either consider either only communication traffic or I/O traffic. In this paper we consider both communication-based and I/O-based contention, and propose allocation strategies that aim to finding a good balance between these two types of communication overhead. We develop a group of *binary tree* based allocation algorithms and a *Hilbert curve* based algorithm to deal with both kinds of contention at the same time. We also introduce the concept of the compact allocation and spatially compact allocation, which serve as formal criteria for evaluating the "compactness" of allocation algorithms..

The rest paper is organized as follows. Section 2 describes our communication model for a mesh cluster. Section 3 presents the algorithms that we developed. Section 4 theoretical results about compact and spatially compact algorithms. Section 5 presents our experiments results. Section 6 concludes the paper.

2. Communication Model

We consider a parallel system where the compute nodes are interconnected with bi-directional network links and are configured as a mesh. We assume that all I/O nodes are located at one side of the mesh. Compute nodes are responsible for computing and they may read/write data from/to I/O nodes. I/O nodes have disks attached to them and are re-

sponsible for writing data from computer nodes into disks and sending requested data to compute nodes.

The parallel processing in our model alternates between computation and I/O phases. During the computation phase, processors conduct inter-processor communication to send and receives messages from each other. Processor then can proceed to the actual computation with all the necessary data. After a computation phase the processors perform I/O operation so that the result created from the computation may be transferred to disk for permanent storage, or new data may be transferred from the disks for processing in the incoming computation stage. Note that the I/O phase is essential to permanently safeguarding the computation results, and periodical transferring the computation results, also known as checkpointing, is a safety measure again unexpected computer malfunctioning and crashes. We classify network contention into two categories. Compute nodes may exchange data so that they compete with each other for the communication links, and compute nodes may compete with one another for I/O nodes during I/O phases. We will refer to these two kinds of network contention as *communication-based* contention and *I/O-based* contention.

This paper proposes strategies in how to allocating processors for incoming tasks in order to minimize both kinds of network contentions. A sequence of tasks are given to our cluster system and each one of them must be served in a timely fashion. Each task requests a fixed number of processors for computation, and tasks do not share compute nodes. The goal is an allocation strategy of assigning compute nodes to a incoming task, which provides a spatial layout of allocated compute nodes, so that both communication-based and I/O-based network contention are reduced.

3. Tree-Based Allocation Algorithms

In this section, we propose a binary-tree-based allocation strategy. We first introduce a basic tree-based algorithm which aims to finding a balance between optimizing compactness for communication-based contention and balancing compute nodes relative to I/O nodes for I/O-based contention. We then enhance the tree-based algorithm with non-contiguous allocation capability for better system utilization.

3.1. Basic Binary-Tree Allocation

We assume that the cluster has an M by N mesh topology. We assume that both M and N are powers of 2, and $M \geq N$. Due to the nature of mesh topology we will allocate processors to tasks in "blocks", with is a rectangle in the processor mesh. From now on we will use the term "rectangle" as a synonym of "processor block".

We use a binary tree T to represent the current allocation status. Initially the tree T has only a root representing the available processors as a M by N rectangle. When the first task arrives, the binary tree allocation algorithm looks into the binary tree T, which has only one rectangle of size M by N at this moment, and decides MN processors is too many for the incoming task. If this is the case, the algorithm divides the root rectangle into two half equal sized halves. When BT decides to divide a rectangle, it will "cut" the rectangle in the middle of the longer side to maintain a better aspect ratio, since a rectangle with high aspect ratio tends to incur serious communication overheads. As a result BT divides the root rectangle into two sub-rectangles that both are of size $\frac{M}{2}$ by N, and sets two rectangles to be the children of the root of tree T. This selection process recursively traverses the left sub-rectangle until BT finds a rectangle large enough to accommodate the incoming request. By the nature of recursive binary division the number of processors in the rectangle (denoted by A) and the number of processors request (denoted by R) satisfies the inequality $\log_2(A) - 1 < \log_2(R) \leq \log_2(A)$. BT also "backtracks" in order to find the rectangle for a task. It is possible that a task cannot fit into a rectangle not because the rectangle does not have enough processors for that task, but because the available processors are located in non-contiguous areas. This is called *external fragmentation*. If BT cannot allocate enough processor in one rectangle due to external fragmentation, it backtracks to parent node in the binary tree T and tries the sibling node. Backtracking allows BT to try every possible rectangle in the mesh system in order to allocate processors for a task.

After BT finds the rectangle for a task, it allocate processors within the rectangle in a column-first fashion. The reason is that by maintain a wider "front" facing the I/O nodes on the left-hand side of the mesh, we can utilize all the horizontal communication links to the I/O subsystem. This conform to the principle of good spatial layout in relation to parallel I/O as reported in [6].

Imbalance Tolerance The simple BT strategy has a potential problem in causing serious external fragmentation, therefore we introduce the concept of the imbalance tolerance value to overcome this difficulty. We define a threshold value called *imbalance tolerance*, denoted as α, that serves as a guideline for allocation within a rectangle. When BT selects one sub-rectangle from two children for allocation, it tries to allocate the task into the child that has *heavier* load without causing an imbalance that exceeds the tolerance α. After the allocation, BT checks the imbalance by comparing the difference of the number of available processors of two children with the tolerance α. If the imbalance does not exceed α, the allocation is valid. Otherwise the task will be allocated to the child that has *lighter* load.

3.2. Non-contiguous Binary-Tree Allocations

Although SBT and DBT solve the external fragmentation problem to some extents, there could be some problem due to the contiguous allocation strategy they enforce [4]. It seems that contiguous allocation suffers from both external and internal fragmentation, therefore we would like to adapt our BT into non-contiguous allocations (NCBT). We adopt a technique from MBS allocation [4] that breaks a large task into small sub-tasks, and allocates a rectangle for each of these smaller sub-tasks.

After we divide the task into subtasks we allocate processors for them independently or dependently. Those NCBTs that use the first strategy are refereed to as independent NCBTs, including ind-BT, ind-SBT and ind-DBT. Those NCBTs that adopt the second strategy are refereed to as dependent NCBTs, including d-BT, d-SBT and d-DBT. Figure 1 shows the binary allocation tree after the ind-BT allocates two tasks of size 5 and 3 each.

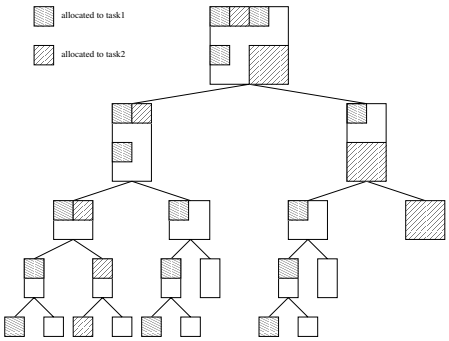

Figure 1. The binary tree status after ind-BT finished allocating the two tasks.

The goal of the dependent NCBTs is to place sub-tasks as close as possible. When dependent NCBTs successfully allocate processors for the biggest sub-task, they remember the location of the rectangle allocated (denoted by A). When dependent NCBTs try to allocate the next sub-task, they start from A's parent, instead of from the root. The sibling of A will be considered first in allocating the second sub-task so that it will be placed as close as to the first one. The subsequent sub-tasks are handled in a same manner. Figure 2 shows the binary allocation tree after the d-BT allocates the same two tasks from the previous example.

4. Compact Allocation

This section describes the concept of *compact allocation* with a formal mathematical definition. Intuitively a compact allocation method assigns adjacent processors to a task

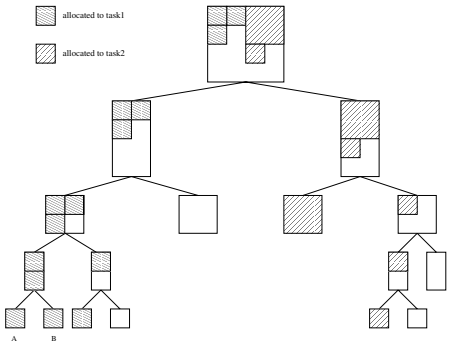

Figure 2. The binary tree status after d-BT finished allocating the two tasks.

so that the communication among the processors is minimized. Geometrically speaking, an allocation algorithm is compact if the processors assigned to a task are close to each other, instead of being scattered all over the geometric space. We assume that the network topology is a $2^m \times 2^n$ mesh and $m \geq n$. The goal of a compact allocation is that after we allocate processors to a task, the remaining free processors can be divided into squares with side length 2^i, and the number of squares is minimized. That is, if we could form a processor square of size $2^i \times 2^i$ within the remaining free processors, we won't divide it into four $2^{i-1} \times 2^{i-1}$ squares. Formally we represent the geometry configuration of the remaining free processors a set of square blocks. Let d_i be the number of squares of size $2^i \times 2^i$, for $0 \leq i \leq n$.

Definition 1 *An allocation algorithm is compact if it maintains the minimum number of squares after every allocation, i.e., the quantity $D = \sum_{i=0}^{n} d_i$ is minimum throughout the allocation process.*

4.1. Spatially Compact Allocation

This section defines a compactness criteria called *spatially compact*, which employs spatial information in judging the effectiveness of allocation algorithms. Before we formally define the spatially compact property, we describe a processor organizing mechanism by which the spatially compact property can be defined.

We assume that the cluster is is a $2^m \times 2^n$ mesh, and $m \geq n$. A rectangle processor block is divided into two equal sized halves by cutting in the middle of the *longer* side to maintain a better aspect ratio. For example in our $2^m \times 2^n$ mesh the root rectangle is divided into two rectangles that both are of size $2^{m-1} \times 2^n$. The two newly generated rectangles will be the children of the original rectangle. The cluster is recursively divided into smaller blocks until each processors is put into its 1×1 block. Note that if we are to divide a square processor block, the division could be

done in either direction. We call this partition method "orthogonal", and by definition, an orthogonal division first divides a $2^m \times 2^n$ mesh into $2^n \times 2^n$ squares, then within each $2^n \times 2^n$ square the division divides the block in alternating directions.

Definition 2 *Let C be a cluster of $2^m \times 2^n$ mesh where $m \geq n$, and T be the result of an orthogonal division on C. An allocation method A is spatially compact if A does not allocate any processor from a subtree until all processors in its sibling subtree are allocated.*

Lemma 1 *An allocation algorithm is spatially compact if and only if the after every allocation step the number free processor blocks for each size is either 0 or 1, and if p and q are two free processor blocks and p is bigger than q, the parent of p is an ancestor of q.*

4.1.1. Spatially Compact Binary Tree Allocation According to Definition 2, we give an example of spatial compactness by modifying our BT allocation method so that it satisfies the spatial compactness requirement. The resulting allocation algorithm is called *spatially compact BT allocation* (SCBT).

We first introduce some notations. Let T be an orthogonal division on the cluster C. Let $S(p)$ be the number of the currently available processors of in tree node n, and $L(p)$, $R(p)$ and $P(p)$ be the left child, the right child, and the parent of p respectively. Note that $S(p)$ is the *current* number of free processors in p so it will decrease after processors in p are allocated.

Initially there is only one available block of size $2^m \times 2^n$ in T. Note that the tree T is a dynamic structure that reflects the current allocation status. We may need to divide a rectangle into two halves by the direction mandated by orthogonal division. In that case the tree T "grows" two children from the tree node that was partitioned.

We assume that the size of the incoming task is k and $0 < k \leq 2^{m+n}$. We first represent k as a binary number, i.e., $k = \sum_{i=0}^{\lfloor log_2 k \rfloor} d_i \times 2^i$. This is equivalent to dividing the task into requests according to its binary representation. To allocate processors for the entire task is equivalent to allocating d_i blocks of size 2^i for every i.

For each request of size 2^i, the spatially compact BT does the following. First p is set to the root and z, the size of the request, is set to 2^i. We will consider two cases – whether the current node p is a leaf node or not. If p is an internal node, we check whether we can allocate the request entirely within the left subtree of p, i.e., we compare $S(L(p))$ with z. If $S(L(p)) \geq z$, we set p to be the left subtree $L(p)$ and repeats the allocation process for next request. If the request requires more processors than the left subtree can offer, i.e., $S(L(p)) < z$, we allocate $L(p)$ for the task, subtract $S(L(p))$ from z, and starts all over from the right subtree by setting p to $R(p)$.

If we have reached a leaf node during the previous allocating process, we consider three sub-cases. In the first sub-case we have $S(p) = z$, we then allocate p to the request and sets p to the root, and try to allocate processors for the next request from there. In the second sub-case when $S(p) > z$, we divide the current tree node p into two children. We then set p to be the left child $L(p)$ and repeats the allocation process. In the third case when $S(p) < z$, we allocate p to the request, subtracts $S(p)$ from z, sets p to the right subtree $R(P(p))$ and repeats the allocation process.

Before we establish that the spatially compact BT allocation is indeed spatially compact, we show that the number of remaining free processor for each size is 0 or 1.

Lemma 2 *The number of free processor blocks for each size from the spatially compact algorithm A is always 0 or 1. That is, if the number of free processors is f, then it can be represented as $f = \sum_{i=0}^{l} d_i \times 2^i, 0 \leq d_i \leq 1$. In addition, the If p and q are two free processor blocks and p is bigger than q, the parent of p is an ancestor of q.*

From Lemma 1 and Lemma 2 we conclude that the BT allocation described earlier is spatially compact. We can also relate the concepts of compactness and spatially compactness with the following theorem.

Theorem 1 *A spatially compact allocation algorithm is also a compact allocation algorithm.*

5. Experiments

5.1. Experimental Environment

We model a mesh cluster system consisting of 16×16 compute nodes and an I/O subsystem of 16 I/O nodes located at the left-hand side of the system. The bandwidth of the network is set to be 11.65MB/sec – a measurement from our fast ethernet that connects our cluster system. The writing speed of disks is set to be 4.545MB/sec – also an actual measurement from our IDE disks. The allocation input consists of 16 tasks and the size of each task is taken from an exponential distribution of mean value 16. All tasks simultaneously enter the system at the beginning of the simulation. The system utilization ranges from 60% to 75% from our simulation.

5.2. Simulation Guidelines

Our simulated parallel application alternates between computation and I/O phases. During the computation phase, processors conduct inter-processor communication to send and receives messages from each other. After a computation phase the processors perform I/O operation so that the result created from the computation is transferred to disk for permanent storage, or new data may be transferred from

the disks for processing in the incoming computation stage. We let all tasks communicate first and then do one I/O operation. We control the frequency of communication to control the ratio of I/O traffic to the communication traffic. For each simulation case we measure the average values from 1000 independent runs.

We assume that the files are striped and distributed among I/O nodes. Therefore tasks transfer data according to a complete bipartite I/O pattern, i.e. each processor allocated to the task sends a personalized message to every I/O node.

5.3. Experimental Results

5.3.1. The Impacts of Configurable Parameters We first describe the configuration parameters for all algorithms. The SBT and DBT can tune their performance by adjusting the global tolerance value and the tolerance ratio.

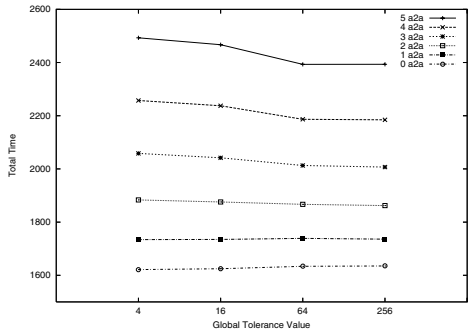

Figure 3. Timing results of SBT. The x coordinate is the global tolerance value: 4, 16, 64, and 256. Different lines represent different number of communications between two I/O phases.

First, the lines for 100% pure I/O are all flat, i.e., in I/O only traffics the performance difference due to different tolerance values is small. We conjecture that the phenomenon is due to *system load*. We start the simulation with a mesh full of tasks. When the system load becomes very heavy, it does not make much difference how the algorithm balances the allocations.

Second in Figure 3 we observe that a large global tolerance value results in better performance when the communication traffic increases, but there is no such performance improvement when the traffic is I/O only. In Section 3.1 we described that the tasks are allocated more compactly when the global tolerance value becomes larger. Allocating tasks more compactly results in better system utilization, which reduces the total time because more processors can work simultaneously. Therefore we expect that in Figure 3 a larger global tolerance should provide better performance, i.e., less total simulating time. However, the last line down below (for 100% I/O) in Figure 3 does not conform with our expectation. The reason is that the system has more I/O-based contentions when tasks are allocated more compactly [6], therefore the increased time due to more I/O-based contention neutralizes the reduced time due to better system utilization. As the communication traffic increases, the effect resulted from better system utilization becomes more obvious. We make the same conclusions for Figure 4 from experiments results of DBT – as the tolerance ratio becomes larger, the tasks are allocated more compactly.

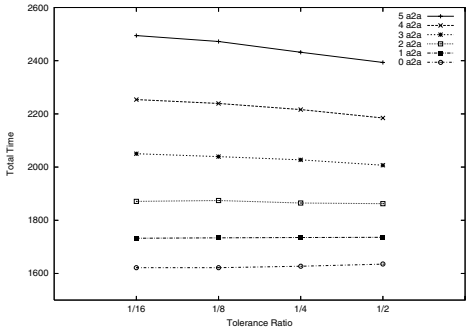

Figure 4. Timing results of DBT. The x coordinate is the tolerance ratio: 1/16, 1/8, 1/4, and 1/2. Different lines represent different number of communications between two I/O phases

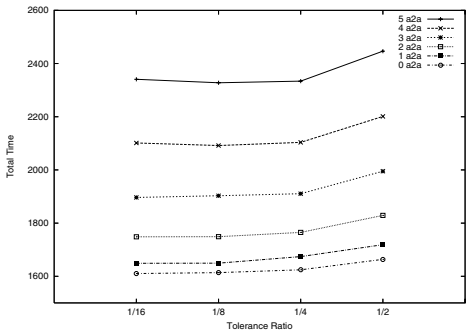

Figure 5. Timing results of d-DBT. The x coordinate is the tolerance ratio: 1/16, 1/8, 1/4, 1/2. Different lines represent different number of communications between two I/O phases

Figure 5, shows the results from d-DBT, and we observe that a larger global tolerance ratio results in worse performance, and unlike independent allocation strategy, this per-

formance penalty becomes more obvious when the communication traffic increases. Dependent allocation strategy has already made tasks as compact as possible, hence a large tolerance value does not help much with respect to the compactness. A large tolerance value, however, makes tasks close to each other, and this reduces the "space" around the first allocated request from each task. Now since there is not enough space around the first allocated request, the remaining requests from the same task are not likely to be allocated close nearby, which creates more communication interference among tasks. As a result in figure 5 a larger tolerance ratio degrades performance when the communication traffic dominates.

We now compare DBT, d-DBT, PLAS and MBS. Each algorithm is represented by the best performing configuration parameters. From Figure 6 we conclude that DBT provide best performance when communication traffic dominates – it outperforms d-DBT, MBS, and PLAS. This advantage is more obvious when the number of communications between two consecutive I/O increases, as indicated in Figure 6. That means when DBT has the best performance when the inter-processor communication is heavy, despite the potential fragmentation problems.

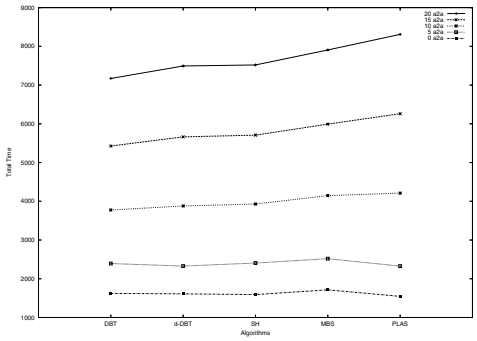

Figure 6. Timing results from DBT, d-DBT, MBS and PLAS algorithms. Different lines represent different number of communications between two I/O phases

6. Conclusions

In this paper we developed allocation algorithms to reduce both communication-based and I/O-based contentions. We develop a group of binary tree based allocation methods that follow the principle of keeping "good" spatial layout to reduce the I/O-based contention and simultaneously maintain "compactness" to deduce the communication-based contention. We also introduce the concepts of compact allocation and spatially compact allocation. We prove that MBS is a compact allocation algorithm, and every spatially compact allocation algorithm is also compact.

We have developed 9 BT allocation algorithms. The initial BT algorithm suffer the external fragmentation. To deal with fragment issues we developed the concept of tolerance value and thus we made the distinction of static and dynamic BT methods. Later in order to remove both internal and external fragmentation, we derived the non-contiguous versions of BT. According to the way to allocate processor for subtasks we have independent and dependent strategy.

We made the following conclusions from our experimental results. First, dynamic and static BT successfully reduce the external fragmentation problem and thus outperform BT. Second, when we derive a non-contiguous version of BT allocations, the dependent strategy is more suitable for dynamic BT and the independent strategy is more suitable for static BT. Third, among all NCBT algorithms, d-DBT has the best performance in most cases. Among all BT allocations, SBT and DBT has best performance only when the communication among compute nodes dominates the overall time. d-DBT has the best performance when the I/O and communication traffic are well mixed. Therefore d-DBT is the most robust allocation algorithm under all possible mixtures of communication and I/O traffics.

References

[1] Y. Cho, M. Winslett, S. Kuo, Y. Chen, J. Lee, and K. Motukuri. Parallel I/O on networks of workstations: Performance improvement by careful placement of i/o servers. In *Proceedings of High Performance Computing on Hewlett-Packard Systems*, 1998.

[2] S. Garg. Parallel I/O architecture of the first asci tflops machine. In *Proceedings of Intel Supercomputer Users Group*, 1997.

[3] V. J. Leung, E. M. Arkin, M. A. Bender, J. Johnston, A. Lal, J. S. B. Mitchell, C. Phillips, and S. S. Seiden. Processor allocation on cplant: Achieving general processor locality using one-dimensional allocation strategies. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, pages 296–304, 2002.

[4] V. Lo, K. Windisch, W. Liu, and B. Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, July 1997.

[5] J. Mache, V. Lo, and S. Garg. How to schedule parallel I/O intensive jobs. In *Proceedings of the 6th Conference on Parallel and Real-Time Systems*, 1999.

[6] J. Mache, V. Lo, M. Livingston, and S. Garg. The impact of spatial layout of jobs on parallel I/O performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, 1999.