Maintaining Spatial Data Sets in Distributed-Memory Machines [*]

Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
seh@cs.purdue.edu

Ashfaq A. Khokhar
Department of Electrical Engineering
University of Delaware
Newark, DE 19716
ashfaq@eecis.udel.edu

Abstract

We propose a distributed data structure for maintaining spatial data sets on message-passing, distributed memory machines. The data structure is based on orthogonal bisection trees and it captures relevant characteristics of parallel machines. The operations we consider include insertion, deletion, and range queries. We introduce parameters to control how much imbalance is tolerated at processors and how close to an optimal load to balance. When balancing, we first determine and broadcast point counts of a data-dependent partition of the data. Based on this partition, we propose load balancing methods with different communication and computation requirements. We present initial experimental results for the Cray T3D.

1. Introduction

In a distributed computing environment, uniform workload across processors is essential for fast execution times. In most applications, workload on a processor is directly related to the amount of data assigned to the processor. If data is organized in a data structure, the distribution of the data structure among the processors needs to be decided. Two issues influencing data distribution are how to efficiently determine where a data element is located and, as the data set changes, how to maintain a balanced load among processors. For simple structures, such as arrays, data location and load balancing have have been studied, analyzed, and incorporated into compiler tools [1, 2, 6, 9, 11, 12]. The distribution of recursive structures such as linked lists and trees have also been investigated recently [4, 5, 13, 15]. Most of the techniques developed for dynamic redistribution of arrays, linked lists, or trees assume that data elements represented by these data structures have no spatial relationship among each other. In applications such as medical imaging, geographical information systems, simulation of physical systems, weather prediction, and time varying images, data elements are spatially related to each other. Reassignment of data during balancing is dependent on its location. If spatial characteristics of a data set are lost, processing can become inefficient.

In this paper we address dynamic load balancing of distributed data structures for spatial data sets on message-passing, distributed memory machines. We propose a distributed data structure based on orthogonal bisection trees (OBT), also known as 2-d K-D trees [3, 14], which broadens the definition of OBTs to capture relevant characteristics of parallel machines. The operations we consider include insertion, deletion, and range-queries. To facilitate efficient location of data, we allow each processor to store a copy of the OBT. Load balancing thus requires that data remain organized as an OBT. We propose a method for monitoring load distributions during the processing of operations. We introduce parameters which (i) control how much imbalance is tolerated at the processors during processing and before balancing, and (ii) determine how close the generated load will be to an optimal load. These parameters give a tradeoff between the number of times the OBT is balanced and the amount of imbalance experienced during processing.

We develop space- and communication-efficient methods for balancing and analyze their performance. When an OBT is to be balanced, processors need to know more than the structure of the OBT and the total number of points assigned to each processor. Ideally, processors should know where the changes in the data set occurred. Space constraints do not allow a processor to know the location of all points or all changes in locations. We propose a solution which partitions the data set into finer regions as done by the OBT. This partitioning is data-dependent and it adjusts, in the number of cuts as well as the position of the cuts, to changes in the data set. The number of data points in the finer regions are made available to the processors. We present

[*]Research supported in part by DARPA under contract DABT63-92-C-0022ONR. The views and conclusions in this paper are those of the authors and should not be interpreted as representing official policies of the U.S. government.

balancing approaches using these counts and we compare their space, time, and communication requirements. We assume that the data set corresponds to a set of points. Let N be the total number of points distributed among p processors, $N >> p$. We assume that the amount of space available at each processor is $O(k) + f(p)$, where $k = \lceil N/p \rceil$ and $f(p)$ is a small polynomial in p.

2. Balanced Orthogonal Bisection Trees

An Orthogonal Bisection Tree (OBT) of size p is a binary tree containing p leaves. For $p = 1$, the OBT consists of a single node. For $p \geq 2$, the root has two children, with the left (resp. right) child being the root of an OBT of size $\lfloor p/2 \rfloor$ (resp. $\lceil p/2 \rceil$). Every node of the OBT has associated with it a point set. The original point set of size N is associated with the root. Let v be a node in the OBT and let v_l and v_r be its left and right child, respectively. The point set associated with node v is partitioned into two sets through an orthogonal bisection and these sets are associated with v_l and v_r, respectively. When making bisections, we alternate vertical and horizontal cuts.

Let n_v, n_l, and n_r be the sizes of the point sets associated with nodes v, v_l and v_r, respectively, $n_l + n_r = n_v$. Let p_v be the number of leaves in the tree rooted at v and let $k = \lceil N/p \rceil$. In a *completely balanced OBT* the number of points associated with a leaf is either k or $k-1$. In a distributed-memory environment, the point set associated with leaf i is assigned to processor P_i. P_i is thus assigned points within a rectangular region R_i (which can be unbounded). See Figure 1 for an example. Enforcing the condition that a processor is either assigned k or $k-1$ points in a completely balanced OBT would lead to frequent load balancing steps in a distributed-memory machine. Hence, one can expect overall poor performance for completely balanced OBTs.

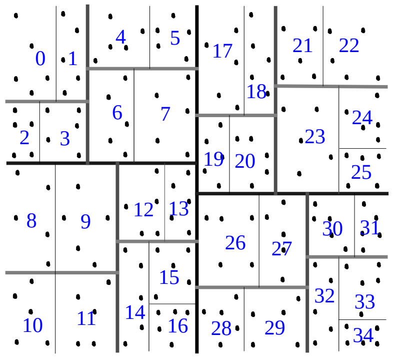
We propose the use of balanced OBTs in which the conditions on the sizes of the associated point sets are relaxed. Let δ be a given constant, $0 \leq \delta \leq 1$. In a *balanced OBT* we have for every node v

$$(k-1)\lfloor \frac{p_v}{2} \rfloor (1-\delta) \quad \leq n_l \leq \quad k\lfloor \frac{p_v}{2} \rfloor (1+\delta) \text{ and} \quad (1)$$

$$(k-1)\lceil \frac{p_v}{2} \rceil (1-\delta) \quad \leq n_r \leq \quad k\lceil \frac{p_v}{2} \rceil (1+\delta). \quad (2)$$

We thus allow the size of the point set associated with a node to deviate from the size allowed in a completely balanced OBT by a constant fraction. For example, for $\delta = \frac{1}{2}$, the number of points assigned to a processor in a balanced OBT lies between $\frac{1}{2}(k-1)$ and $\frac{3}{2}k$.

Relaxing the conditions on the size of a point set assigned to a processor does not solve the problem of repeated balancings. For example, if a processor is assigned $k(1+\delta)+1$ points before balancing and $k(1+\delta)$ points after balancing, the insertion of a single point can cause the OBT to be unbalanced again. To solve this problem, we use another set of parameters. This set includes ϵ_1 and ϵ_2 with $\delta \leq \epsilon_1 \leq 1$ and $\delta \leq \epsilon_2$. Parameters ϵ_1 and ϵ_2 control the imbalance tolerated at the processors during the processing phase. As long as the number of points assigned to a processor lies between $(k-1)(1-\epsilon_1)$ and $k(1+\epsilon_2)$, the processor continues processing. For example, for $\delta = \frac{1}{2}$, $\epsilon_1 = \frac{3}{4}$, and $\epsilon_2 = 2$, we allow the number of points to lie between $\frac{(k-1)}{4}$ and $3k$ during the processing of queries. The relative effect of δ and ϵ_1 & ϵ_2 on the value of k is graphically depicted in Figure 2. Let l_i be the number of points in region R_i at some point


(a) bisection cuts for $N = 175$ and $p = 35$

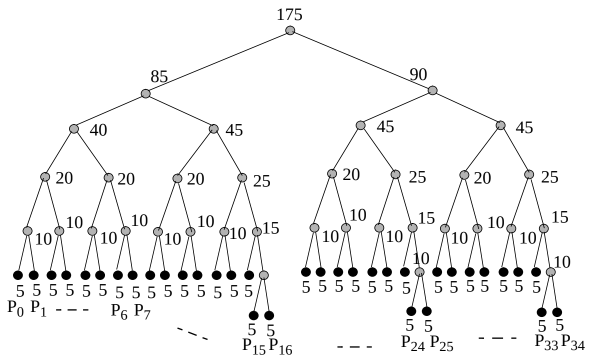

(b)) corresponding OBT with $k = 5$

Figure 1. Example of an OBT

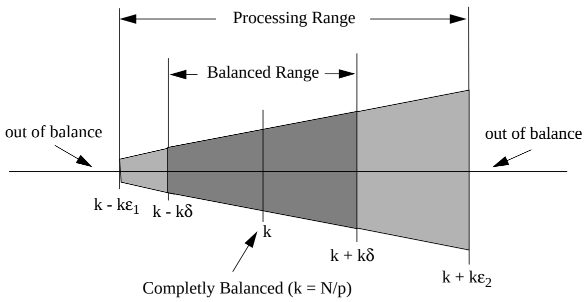


Figure 2. Load balancing parameters

during the processing phase. If l_i is in the "out-of-balance" range (i.e., $l_i < (k-1)(1-\epsilon_1)$ or $l_i > k(1+\epsilon_2)$), processor P_i initiates a check to determine whether the current OBT is indeed unbalanced.

3. Processing and balancing overview

At the beginning of a processing phase the OBT is balanced and the number of points assigned to each processor lies in the balanced range. Every processor holds a copy of the OBT and knows N, the current total number of points. In the processing phase, the point distribution changes dynamically. We select as representative operations on spatial data the operations insert, delete, and range-count. When processor P_i issues the insertion of point (x, y), it uses its local OBT to determine the processor assigned the region containing (x, y) and then initiates the insertion at the appropriate processor. A deletion is handled in a similar way. A range-count operation consists of two points (x_1, y_1) and (x_2, y_2), which are the lower left and upper right corner of a rectangle Q. When P_i issues a range-count, it needs to know the current number of points in rectangle Q. If Q does not lie within R_i, P_i determines the processors whose assigned region overlaps with Q. Then, P_i issues a send to each one of these processors for their current point count in Q. The implementation of all 3 operations is straightforward and it is not the contribution of this work.

During the processing phase each processor maintains the number of points currently assigned to its region. We refer to this number as the region count. If a region count falls in the "out-of-balance" range with respect to N and k determined at the beginning of processing, processing is suspended and balancing starts. The first step of the balancing phase is to update values needed for balancing. This corresponds to making the region counts of all processor available to every processor. Let N' and k' be the new total number of points and the new load in a completely balanced OBT, respectively. Load balancing proceeds if at least one processor is not balanced with respect to N' and k'. Hence, no load balancing is required if, for example, the global trend of change in the workload was experienced by all processors.

The objective of the balancing phase is to generate a new, balanced OBT. Generating a balanced OBT involves determining new cuts. New cuts induce new regions. New regions imply that points have to reassigned. Observe that the shape of the OBT tree is determined by p and does not change. Balancing starts at the root of the OBT and, if a node of the OBT does not satisfy conditions (1) and (2) for N' and k', a new cut is determined. In order to determine a new cut, processors need more information than the current number of points in each region. Our approach is to divide the point space into smaller regions and make the number of points in each smaller region available to the processors. The goal is to generate a partition into small regions so that the partition is data-dependent and the partition is not fixed from one balancing instance to another and can thus adjust to changes in the data set.

In the following we give an overview of how to determine such a partition. Resulting balancing approaches differ in the amount of information required by each processor and the amount of communication needed. The orthogonal bisections used for building the OBT give a crude reflection of the distribution of data. A first partition is obtained by extending all bisection cuts across the entire data space. The dashed lines shown in Figure 3 show the partition obtained from these cuts for a particular OBT. We refer to the cuts inducing the partition as the *projected cuts*. All our balancing approaches include the cuts of the OBT in the pool of projected cuts. Additional cuts are added when a finer partition is needed. We assume that balancing uses $O(p)$ projected cuts.

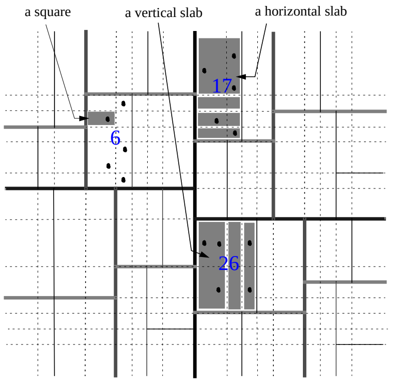


Figure 3. Projected cuts

The horizontal and vertical projected cuts induce $O(p^2)$ rectangles. To avoid confusion with the rectangular regions assigned to processors, we refer to these rectangles as *squares*. We refer to the number of points in a square as the *square count*. Observe that it is possible for a region to be intersected by $O(p)$ projected cuts and thus to contain $O(p^2)$ squares. When balancing using square counts, every processor knows all square counts and thus requires $O(p^2)$ space. Square count methods determine first whether an existing projected cut can be used as a new cut for the node. If it can, no further communication is needed for balancing the node.

Slab count methods find the same set of new cuts, but use less space. However, book-keeping can be more expensive. Consider region R_i assigned to processor P_i. The horizontal projected cuts intersecting R_i induce horizontal slabs of the width of R_i. The vertical projected cuts intersecting R_i induce vertical slabs of the height of R_i. The number of points in a particular slab is a *slab count*. The number of slabs within R_i can be $O(p)$. However, the total number of slabs is $O(p^{3/2})$. We describe slab balancing methods requiring $O(p^{3/2})$ and $O(p \log p)$ space, respectively.

Figure 4 summarizes the costs arising in the different implementations of the above approaches. If no existing projected cut can be used to balance a node, the projected cuts and the available counts guide the identification of a region in which possible cuts lie.

4. Balancing a node of the OBT

In order to balance node v, a new cut needs to be determined. We to first determine whether a projected cut can be used. If no projected cut can satisfy the balancing condition for v, slab and square methods identify a narrow region in which a new cut must lie. We assume the point set assigned to node v is bisected using a vertical cut at x_v. W.l.o.g. assume that points assigned to the right child of v need to be reassigned to left child of v; i.e., the new cut lies to the right of x_v. Let m_l (resp. m_u) be the minimum (resp. maximum) number of points to be reassigned. Balancing node v means determining a position x_r so that the number of points (x, y) with $x_v \leq x \leq x_r$ lies between m_l and m_u. Let R be the region between the vertical cuts x_v and x_r.

When balancing node v using square counts, the points in a square can be assigned to a different processor. Such reassignments are recorded, but no point sets are moved. Our philosophy is to delay sending points as long as possible. Points sets are sent only when no projected cut can be used (and they are necessary to determine a new cut) or and after the new OBT has been found. Square counts induce a 2-dimensional array and it is thus natural for processors to store square counts as such. When square counts are stored in an array, say array SC, we associate with each square two entries: the processor whose region currently contains the points in the square and the processor who is currently assigned the points. Determining whether v can be balanced using a projected cut can be done in $O(\alpha p)$ time, where α is the number of projected cuts between x_v and x_r. If a projected cut can be used, no communication is necessary to update counts and reassignments. If no projected cut balances node v, we invoke the cut finding procedure described in Section **??**.

Using matrix SC to store square counts is simple, but can lead to repeated computations in matrix SC. The use of binary trees avoids this. The amount of space and the initial set-up cost are $O(p^2)$. Projected cut x_r is now determined by making queries on vertical trees, starting with the vertical tree associated with the vertical cut at x_v. Finding the position of x_r costs $O(\alpha \log p)$ time, compared to $O(\alpha p)$ time for the matrix. The array maintains the association between squares and processors. In the tree implementation, we no longer maintain this association in an explicit way, but generate this information when needed. Region counts and necessary associations between squares and processors can be maintained in $O(\min\{\alpha\sqrt{p_v}\log p, p\log p\})$ time. Details are described in [8].

Balancing methods based on slab counts use the same pool of projected cuts and determine the same cuts. Slab counts give a somewhat coarser partitioning of the point space and require less space. In the initial all-to-all broadcast sending out slab counts, a processor receives and stores $O(p^{3/2})$ counts, as compared to $O(p^2)$ counts for square counts. When balancing node v by using a projected cut it may now be necessary to generate and broadcast counts. To identify where the difficulties arise, assume first that slab counts are stored in matrices. Let HS (VS) be a matrix for the horizontal (vertical) slab counts. Determining whether a projected cut can be used to balance node v is straightforward.

Regions lying between cuts x_v and x_r (i.e., within region R) have a new region count of 0. If a region R_i is cut by x_r, its new region count can be determined from the available vertical slabs counts. Consider now a region R_i immediately to the left of x_v. R_i's region count increases. Neither vertical or horizontal slab counts contain the information needed to update R_i's region count. Let y_i' and y_i'' be the upper and lower y-position of region R_i, respectively. To determine R_i's region count from horizontal slab counts, consider all horizontal projected cuts between y_i' and y_i''. Let y be one of them. For any region R_j lying entirely in R, we add $HS(y, j)$ to the region count of R_i. For any region R_j intersected by x_r, we need the number of points in $HS(y, j)$ which are to the left of x_r. This information can only be generated by processor P_j, the processor currently containing the points in R_j. A processor whose region is intersected by x_r determines how each one of its horizontal slabs partitions around x_r. These counts are broadcast to the processors that need this value to update region counts.

The updating of the horizontal slab counts can be done by using the point counts made available for the updating of region counts. In order to update the vertical slabs, the points in region R need to be organized into vertical slabs whose positions correspond to the regions of the points immediately to the left of cut x_v. This means that all processors in region R (not just the ones immediately to the left of x_r) need to generate point counts. This additional communication overhead can be avoided by collapsing the vertical slabs between x_r and x_v into single vertical slab. When this is done, no additional communication is required, but it does not allow descendents of v to use the projected cuts between x_v and x_r as new cuts.

When slab counts are stored in matrices, one iteration uses $O(\alpha p_v)$ sequential time and performs one broadcast. In [8] we describe two improved implementations. The first one uses $O(p^{3/2})$ space and associates with every horizon-

	square counts matrix	square counts trees	slab counts matrix	slab counts trees	slab counts multi-res
initial all-to-all broadcast	$\Theta(p^2)$	$\Theta(p^2)$	$O(p^{3/2})$	$O(p^{3/2})$	$O(p^{3/2})$
initial set up cost	$\Theta(p^2)$	$\Theta(p^2)$	$O(p^2)$	$O(p^{3/2})$	$O(p^{3/2})$
total space	$\Theta(p^2)$	$\Theta(p^2)$	$\Theta(p^2)$	$O(p^{3/2})$	$O(p \log p)$
communication	none	none	$\sqrt{p_v}$-to-p_v broadcast		
computation	$O(\alpha p)$	$O(\min\{\alpha\sqrt{p_v} \log p, p \log p\})$	$O(\alpha p_v)$	$O(p \log p)$	$O(p \log p)$

Figure 4. Comparing different methods for balancing a node v

tal projected cut (resp. vertical projected cut) a binary tree with $\sqrt{p}$ leaves. Using these trees, we can determine in $O(\alpha \log p)$ time whether a projected cut can be used to balance v. Region and slab counts can be updated in $O(p \log p)$ time. The second slab count implementation, the multi-resolution method, reduces the space to $O(p \log p)$ and balances a node in $O(p \log p)$ sequential time. Every processor uses now $2 \log p$ arrays, each of size $O(p)$: $\log p$ arrays contain sums of vertical slabs counts and the other $\log p$ arrays contain sums of horizontal slab counts. The arrays can be viewed as holding sums pertaining to different resolutions.

Let m_u and m_l be the upper and lower bound on the points needed to make v balanced. Projected cuts fail to balance node v if there exists a projected cut x_q inducing a region R' containing fewer than m_l points and projected cut x_{q+1} induces a region R'' containing more than m_u points. Hence, there exists a cut at x_r, $x_q < x_r < x_{q+1}$ inducing a desired region R. The points having their x-coordinate between x_q and x_{q+1} reside in $\sqrt{p_v}$ processors. The points in lying between x_q and x_{q+1} are likely to represent only a fraction of the assigned points. We designate one of the p_v processors as the leader and have processors send relevant points to the leader. The leader processor determines the position of x_r, the new cut. The leader also determines point counts reflecting how the new cut partitions the points in a square or slab. This new cut and computed counts are then broadcast to all processors that need to know the position of the cut. We expect that the number of points sent to the leader processor will, in general, be small. In [8] we describe other solutions discuss additional communication issues.

5. Overall balancing of the OBT

Balancing starts at the root. A node v is balanced when all nodes on the path from the root to v have been considered. Given an unbalanced OBT, the number of nodes that need to be balanced is not known ahead of time. We sketch two approaches for balancing an OBT which differ in communication and computation requirements. The first approach is to have every processor balance all nodes. The main motivation for this approach is simplicity. Balancing can proceed level by level or by using another traversal. Consider the square methods for balancing a node. If all nodes can be balanced by using projected cuts, no communication is needed to generate the balanced OBT. When a cut needs to be determined from points sets, communication is as described in Section **??**. Clearly, duplication of work occurs during the balancing .

Our second approach eliminates much of the duplication. We now have every processor P_i handle the nodes on the path from the root to the leaf associated with P_i. Processor P_i thus only balances the nodes necessary to determine its new region. Once all new regions are known, an all-to-all broadcast takes place to make the new cuts available to every processor. The communication arising when a node v is balanced changes slightly. For example, when a new cut is found from point sets, the leader processor broadcasts the new cut and generated point counts only to processors in the subtree rooted at v. The last step in balancing the OBT uses the old and the new OBT to send the points to the processors they are now assigned to.

6. Preliminary experimental results

The main objective of our experimental work is to generate evidence that balancing OBTs leads to overall better performance and to demonstrate that our methods for balancing are effective. We also want to establish guidelines under which conditions balancing an OBT improves the overall performance. We expect that, similar to results obtained for related work [7, 10], answers depend on parameters and factors influencing scalability. We report on preliminary performance results for OBTs with different loads for the Cray T3E. We refer to [8] for additional experimental results. Our code is written in C and uses MPI. The OBT is implemented using an array of size $2p$. Within each processor, the assigned point set is maintained sorted lists. We chose this representation over data structures like range trees for reasons of simplicity and flexibility. Using sorted lists results in logarithmic time for deletion and linear time for insertion. This allows us to experiment with different costs per operation. The range-count operation involves querying processors for information pertaining to the assigned data set and thus allows us to accurately model a real-world application in a simple way.

Figure 5 demonstrates the need for efficient balancing methods. The three curves were obtained by starting off with three different OBTs for the parameters $\delta = 0.5$, $\epsilon_1 = 0.7$, $\epsilon_2 = 2$, and $N = p^2$. The "balanced" curve corresponds to the performance when the initial OBT T_1 is balanced (i.e., $\frac{1}{2} \leq n_i \leq \frac{3}{2}k$, with all values in this range being equally likely). The "processing" curve starts off with an OBT T_2 with $\frac{3}{10} \leq n_i \leq 2k$ in which half the processors are in the balanced range and the other half are in the processing range only. The "out-of-range" curve starts off with an OBT T_3 in which one third of the processors is no longer in the processing range, one third is in the processing, but not balanced range, and the final third is in the balanced range only. The total number of queries executed is p^2 and a processor starting off with n_i points executes n_i queries, with each one of the three query types being equally likely. Processors execute the queries without invoking load balancing. Points for the queries are generated according to a uniform distribution. Hence, one can expect that the distribution of the points does not change significantly during the processing.

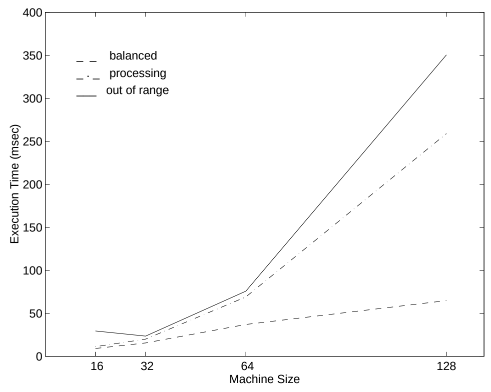


Figure 5. Performance with different loads

For $p = 128$, Figure 5 shows a 4- (resp. 5-) fold increase in the total processing time for OBT T_2 (resp. T_3) compared to OBT T_1. This is a typical behaviour we observed also for other parameters and it clearly demonstrates the need for balancing methods. OBTs T_2 and T_3 have a fairly large fraction of processors in the processing and "out-of-balance" range. Our expermiments have shown that the number of processors in the different ranges plays a role as well, but that by setting the parameters accordingly, the number of processors in different ranges can be ignored.

For larger p, our balancing methods generate more projected cuts and thus the counts represent a better sample of the current point set. We expect our balancing methods to be efficient for large p and to be able to balance by using projected cuts as new cuts. For small machine sizes (i.e., $p \leq 32$) we expect balancing to be expensive unless additional projected cuts are used to avoid determining new cuts from point sets. Preliminary implementations of the square count matrix method support these claims.

References

[1] G. Agrawal, A. Sussman, J. Saltz, "Compiler and Run-time Support for Structured and Block Structured Applications, *Proc. of Supercomputing '93*, pp. 578-587, 1993.

[2] D. Bader, J. Já Já, "Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection," Techn. Report, CS-TR-3494, University of Maryland, 1995.

[3] J. Bentley, "Multidimensional Binary Search Trees used for Associative Searching," *CACM*, Vol. 8, pp. 509-517, 1975.

[4] S. Chakrabarti, E. Deprit, E. Im, J. Jones, A. Krishnamurti, C. Wen, and K. Yelick, " Multipol: A Distributed Data Structure Library," Techn. Report, CSD-95-879, UC Berkeley, 1995.

[5] C. Chang, A. Sussman, J. Saltz, "Object-Oriented Runtime Support for Complex Distributed Data Structures," Techn. Report, UMIACS-TR-95-35, University of Maryland, 1995.

[6] R. Das, M. Uysal, J. Saltz, Y.S. Hwang, "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures," *JPDC*, pp. 462-479, 1994.

[7] S.E. Hambrusch, F. Hameed, A. A. Khokhar, "Communication Operations on Coarse-Grained Mesh Architectures," *Parallel Computing*, Vol. 21, pp. 731-751, 1995.

[8] S.E. Hambrusch, A. Khokhar, "Maintaining Spatial Data Sets in Distributed-Memory Machines", Techn. Report, 1997.

[9] S.R. Kohn, S.B. Baden, "A Robust Parallel Programming Model for Dynamic Non-uniform Scientific Computations," *Proc. of the High Perf. Comp. Conf.*, pp. 509-517, 1994.

[10] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, 1994.

[11] C.-W. Ou, S. Ranka, "Parallel Remapping Algorithms for Adaptive Problems," *Proc. of the Symp. on the Frontiers of Massively Parallel Computation*, pp. 367-374, 1995.

[12] R. Parsons, D. Quinlan, "Run-time Recognition of Task Parallelism within the P++ Parallel Array Class Library," *Proc. of 1993 Scalable Parallel Libraries Conf.*, 1993.

[13] A. Rogers, M.C. Carlile, J. Reppy, L.J. Hendren, "Supporting Dynamic Data Structures on Distributed-Memory Machines", *TOPLAS*, 17(2), pp. 233-263, 1995.

[14] H. Samet, *Applications of Spatial Data Structures, Computer Graphics, and Image Processing*, Addison Wesley, 1990.

[15] K. Yelick et al., "Portable Parallel Irregular Applications," *Workshop on Parallel Symbolic Languages and Systems*, Lecture Notes in Computer Science, 1995.