Parallel Construction of Multidimensional Binary Search Trees

Ibraheem Al-furaih, Srinivas Aluru, *Member*, *IEEE*, Sanjay Goil, and Sanjay Ranka, *Member*, *IEEE*

Abstract—Multidimensional binary search tree (abbreviated k-d tree) is a popular data structure for the organization and manipulation of spatial data. The data structure is useful in several applications including graph partitioning, hierarchical applications such as molecular dynamics and n-body simulations, and databases. In this paper, we study efficient parallel construction of k-d trees on coarse-grained distributed memory parallel computers. We consider several algorithms for parallel k-d tree construction and analyze them theoretically and experimentally, with a view towards identifying the algorithms that are practically efficient. We have carried out detailed implementations of all the algorithms discussed on the CM-5 and report on experimental results.

Index Terms—k-d trees, hypercubes, meshes, multidimensional binary search trees, parallel algorithms, parallel computers.

1 Introduction

Consider a set of n points in k dimensional space. Let $d_1, d_2, \ldots, d_k$ denote the k dimensions. A k-d tree [3] on the points is constructed as follows: The root of the tree corresponds to the set of all points. Choose a dimension d_l, and find the median coordinate of all the points along dimension d_l. We can partition the points into two approximately equal sized sets—one set containing all the points whose coordinates along dimension d_l are less than or equal to this median and a second set containing all the remaining points. The two subpartitions are represented by the children of the root node. The tree is built recursively until each leaf corresponds to one point. Typical strategies used for choosing the dimension to split at a node are: choosing dimension $d_{(i \bmod k)+1}$ for each node at level i in the tree (defining the root to be at level 0), choosing the dimension with the largest span where the span along a dimension is the difference between maximum and minimum coordinates of the points along the dimension. In homogeneous trees, internal nodes are used to store the median points. Nonhomogeneous trees store points only at the leaves.

Several applications require partial construction of k-d trees. In parallel graph partitioning, we are interested in creating p partitions to distribute the graph to p processors, requiring the construction of only the first $\log p$ levels of the k-d tree. In hierarchical applications like the n-body simulation, clustering of physically proximate objects is essential and the k-d tree offers such a clustering scheme. In databases, records can be treated as points in an appropriate space by mapping each key to a coordinate and the resulting point set can be organized using a k-d tree. In constructing the tree, a node is partitioned only if all its records do not fit in one disk sector.

In this paper, we focus on the efficient parallel construction of balanced, nonhomogeneous k-d trees on coarse-grained distributed memory parallel computers. Other variations can be easily implemented with minor changes in our algorithms without significantly affecting their running time. We have considered the two standard approaches used in constructing k-d trees: 1) to use explicit median finding, and 2) to use sorting as a preprocessing step to eliminate median finding. In addition, we propose a new approach that induces a partial order in the data and refines it as necessary, to cover the spectrum of possibilities between these two commonly used approaches. We also present a new strategy to reduce the communication complexity of parallel k-d tree construction, which readily generalizes to other divide and conquer applications. Our results challenge the conventional wisdom that a sort-based strategy is the best algorithm for k-d tree construction.

The rest of the paper is organized as follows: In Section 2, we describe our model of parallel computation and outline some primitives used in our algorithms. In Section 3, we describe several algorithms for the construction of k-d trees and analyze their running times on hypercubes and meshes. We have implemented our algorithms on the CM-5 and each algorithm is accompanied by experimental results. Section 4 presents an algorithm that reduces the communication complexity. Section 6 concludes the paper.

2 Model of Parallel Computation

Multiprocessor platforms have converged to an architecture consisting of a moderate number (10 to a few thousand) of fast and powerful processors connected by a high-speed interconnection network. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space. Popular interconnection topologies are buses (SGI Challenge), 2D meshes (Paragon, Delta), 3D meshes

- *I. Al-furaih is with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244.*
- *S. Aluru is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011. E-mail: aluru@iastate.edu.*
- *S. Goil is with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208.*
- *S. Ranka is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611.*

Manuscript received 9 Jan. 1996; revised 29 July 1998; accepted 16 Dec. 1998. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100088.

TABLE 1
Running Times of Various Parallel Primitives on Cut-through Routed Hypercubes and Square Meshes with p Processors

Primitive	Running time on a p processor	
	Hypercube	Mesh
Broadcast	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Combine	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Parallel Prefix	$O((t_s + t_w)\log p)$	$O((t_s + t_w)\log p + t_h\sqrt{p})$
Gather	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
Global Concatenate	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
All-to-All Communication	$O((t_s + t_w m)p + t_h p \log p)$	$O((t_s + t_w mp)\sqrt{p})$
Transportation Primitive	$O(t_s p + t_w r + t_h p \log p)$	$O((t_s + t_w r)\sqrt{p})$
Order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$
Non-order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p)$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$

(Cray T3D), hypercubes (nCUBE), fat tree (CM5), and hierarchical networks (cedar, DASH). We call such an architecture a *coarse-grained machine* to reflect the high computation-to-communication ratio.

CGMs have cut-through routed networks, which will be used for modeling the communication cost of our algorithms. In the absence of network contention, a message of size m traversing d hops incurs a communication delay given by $T_{comm} = t_s + t_h d + t_w m$, where t_s is the hand–shaking cost, t_h is the signal propagation and switching delay, and t_w is the inverse bandwidth of the communication network. Let t_c be the time to do a unit computation. Typically, t_w is an order of magnitude larger than t_c and t_s is two orders of magnitude larger than t_w. The per-hop component $t_h d$ can often be subsumed into the startup time t_s without significant loss of accuracy. This is because the diameter of the network, which is the maximum of the distance between any pair of processors, is relatively small for most practical sized machines, and t_h also tends to be small.

Instead of making network-specific assumptions, we describe our algorithms in terms of basic communication primitives. A network-specific analysis can be done by substituting the running times of the primitives. We provide such an analysis for hypercubes and two–dimensional meshes. The analysis for permutation networks and hypercubes is the same in most cases. These cover nearly all commercially available machines. A permutation network is one for which almost all of the permutations (each processor sending and receiving only one message of equal size) can be completed in nearly the same time (e.g., CM-5 and IBM SP Series).

Table 1 presents the running times of the primitives used in our algorithms (assuming p processors) on the hypercube and the mesh. For details on commonly used primitives, we refer the reader to [9]. In a *Broadcast* operation, one processor sends a message of size m to all other processors. Given a vector of size m on each processor and a binary associative operation $\oplus$, the *Combine* operation computes a resultant vector of size m, stored on every processor, by combining the ith element on each processor for each of the m entries. Let processor P_i contain data element x_i and let $\otimes$ be a binary associative operation. *Parallel Prefix* stores the value of $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i. Given a vector of size m on each processor, *Gather* collects all the data and stores the resulting vector of size mp on one of the processors. *Global Concatenate* is the same as Gather except that the collected data is stored on each processor P_i. Each processor sends a distinct message of size m to every processor P_i in an *All-to-all Communication*. The more complex primitives are described below:

1. **Transportation Primitive.** It performs many-to-many personalized communication with possibly high variance in message size. Let r be the maximum of outgoing or incoming traffic at any processor The transportation primitive breaks down the communication into two all-to-all communication phases where all the messages sent by any particular processor have uniform message sizes [12]. If $r \geq p^2$, the running time of this operation is equal to two all-to-all communication operations with a maximum message size of $O(\frac{r}{p})$.
2. **Order Maintaining Data Movement.** Consider the following abstraction of the data movement patterns repeatedly encountered in k-d tree construction algorithms: Initially, processor P_i contains two integers s_i and r_i, and has s_i elements of data such that $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$. Let $s_{max} = max_{i=0}^{p-1} s_i$ and $r_{max} = max_{i=0}^{p-1} r_i$. The objective is to redistribute the data such that processor P_i contains r_i elements. Suppose that each processor has its set of elements stored in an array. We can view the $\sum_{i=0}^{p-1} s_i$ elements as if they are globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. This global order should be preserved after the redistribution.

   The algorithm first performs Parallel Prefix operations on the s_i's and r_i's to find the position in the global order of the elements each processor initially contains and should finally contain. Using these, each processor can find the data it should send to each of the other processors and the data it should receive from each processor. The communication is

performed using the transportation primitive. The maximum number of elements sent out (received) by any processor is s_{max} (r_{max}).

3. **Nonorder Maintaining Data Movement.** The order maintaining data movement algorithm may generate much more communication than necessary if preserving the global order of elements is not necessary. For example, consider the case where $r_i = s_i$ for $1 \leq i < p-1$ and $r_0 = s_0 + 1$ and $r_{p-1} = s_{p-1} - 1$. The optimal strategy is to transfer the one extra element from P_{p-1} to P_0. However, this algorithm transfers one element from P_i to P_{i-1} for every $1 \leq i < p-1$, generating $(p-1)$ messages.

   If preserving the order of the data is unimportant, communication can be reduced using the following strategy: Processor P_i retains $min\{s_i, r_i\}$ of its elements. If $s_i > r_i$, the processor has $(s_i - r_i)$ elements in excess and is labeled a source. Otherwise, the processor needs $(r_i - s_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are ranked separately using two Parallel Prefix operations. The data is transferred from sources to sinks using a strategy similar to order maintaining data movement. The worst-case running time is the same as the order maintaining data movement, but the performance is expected to be better in practice.

3 Parallel Construction of k-d Trees

We consider the task of building a balanced k-d tree of N points in k dimensional space up to an arbitrary number of levels using p processors. For simplicity and convenience of presentation, we assume that both N and p are powers of two. We also assume that $N \geq p^2$ and that we build the tree at least up to $\log p$ levels. The first $\log p$ levels of the tree are constructed in parallel and the remaining levels are constructed locally.

The median of the N points along dimension d_1 is found and the points are separated into two partitions, one containing points with coordinate along d_1 less than or equal to the median and the other containing the remaining points. The partitions are redistributed such that the first half of the processors contain one partition and the remaining processors contain the other. This is repeated recursively until the first $\log p$ levels of the tree are constructed. At this point, each processor contains $n = \frac{N}{p}$ points belonging to a partition at level $\log p$ of the tree. The local tree for these n points is constructed up to the desired number of levels. Let the number of levels desired for the local tree be $\log m$ ($m \leq n$).

Any median finding algorithm can be used for the the construction of k-d trees. On the other hand, since the task involves finding repeated medians, some preprocessing can be used to help speed up the median computations. We present two algorithms that use such a preprocessing. The standard method for constructing a complete k-d tree uses presorting the points along each dimension to completely eliminate explicit median finding [4], [10]. The parallel tree construction can be decomposed into two parts: constructing the first $\log p$ levels of the tree in parallel, followed by the local tree construction. Potentially a different strategy can be used for the two parts. For this reason, we describe and analyze each of the three strategies for both the parts.

3.1 Local Tree Construction

3.1.1 Sort-Based Method

In order to avoid the overheads associated with explicit median finding at every internal node of the k-d tree, we use an approach that involves sorting the points along every dimension exactly once [10]. We maintain k arrays $A_1, A_2, \ldots, A_k$. Initially, A_l contains all the points sorted according to dimension d_l. Any node in the k-d tree corresponding to a partition that is yet to be split has two pointers i and j ($i < j$) associated with it such that the subarray $A_l[i..j]$ contains the points of the partition sorted along d_l. If the partition is split based on d_1, splitting the array $A_1[i..j]$ can be done in constant time, as it requires just computing the pointers of the subpartitions. Consider splitting $A_l[i..j]$ for any other dimension d_l. If we simply scan $A_l[i..j]$ from either end and swap points when necessary (as can be done in a median finding method), the sorted order will be destroyed. Therefore, it is required to go over the points twice: once to count the number of points in the two subpartitions and a second time to actually move the data. Counting is necessary because the number of points less than or equal to the median need not be exactly equal to half the points.

Also, the arrays contain pointers to records and not actual records. With this, copying a point requires only $O(1)$ time. This method requires a preprocessing step of sorting the n points along each of the k dimensions, taking $O(kn \log n)$ time. Constructing each level of the k-d tree involves scanning each of the k arrays and takes $O(kn)$ time. After preprocessing, $\log m$ levels of the tree can be built in $O(kn \log m)$ time. In some cases, such as when the sort-based method is used to construct the first $\log p$ levels of the tree, the data may already be present in sorted order.

It is possible to reduce the $O(kn)$ time per level to $O(n)$ by using the following scheme: There are n records of size k, one corresponding to each point. Use an array L of size n. An element of L contains a pointer to a record and k indices indicating the positions in the arrays $A_1, A_2, \ldots, A_k$ which correspond to this record. An element of array A_l is now not a pointer to a record but stores the index of L which contains a pointer to the record. Thus, we need to dereference three pointers to get to the actual record pointed to by an element of any array A_l. Initially, $L[i]$ contains a pointer to record i and all the arrays can be set up in $O(kn)$ time. The arrays $A_1, A_2, \ldots, A_k$ are sorted in $O(kn \log kn)$ time as before. The array L is used to keep track of all the partitions.

Suppose that the tree is constructed up to i levels and the array L is partitioned into 2^i subarrays corresponding to the subpartitions accordingly. Let l be the dimension along which each of these subpartitions should be split to form level $i+1$ of the tree. We first want to organize the array A_l into 2^i subpartitions. Label the partitions of L using $1 \ldots 2^i$ from left to right. For every element of L, using the index for

A_l, label the corresponding element of A_l with the partition number. Permute the elements of A_l such that all elements with a lower label appear before elements with a higher label and within the elements having the same label, the sorted order is preserved. All of this can be done in $O(n)$ time. As before, A_l can now be used to pick the median of each subpartition.

Although this method reduces the run-time per level from $O(kn)$ to $O(n)$, the method is not expected to perform better for small values of k such as 2 and 3, which cover many practical applications such as graph partitioning and hierarchical methods.

3.1.2 Median-Based Method

In this approach, a partition is represented by an unordered set of points and the median is explicitly computed in order to split the partition. We have tested various median finding algorithms and found that the randomized algorithm of Floyd and Rivest [7] results in the best performance. The algorithm works by picking a random element of the set, partitioning the set based on this element, discarding the partition not containing the desired element, and repeatedly performing the same procedure on the partition containing the desired element. The worst-case run time is $O(n^2)$, but the expected run time is only $O(n)$. The expected number of iterations is $O(\log n)$. A different approach can be used to reduce the expected number of iterations to $O(\log \log n)$ [11]. We found that the parallel versions of these two methods have comparable running times, but Floyd's algorithm fares better sequentially [1].

Note that the very process of computing the median of a partition splits the partition into two subpartitions. In constructing level i of the local tree, we have to compute 2^i medians, each on a partition containing $\frac{n}{2^i}$ points. As the total number of points in all partitions at any level of the local tree is n, building each level takes $O(n)$ time. The required $\log m$ levels can be built in $O(n \log m)$ time.

3.1.3 Bucket-Based Method

The sequential complexity of the median-based methods is proportional to $n \log m$. Even though the constant associated with sorting is small compared to median finding, the complexity of sorting is proportional to $kn \log n$. The improvement in the constant may not be able to offset the higher complexity of the sort-based method. To resolve this problem, we start with inducing partial order in the data, which is refined further only as it is needed.

Sample a set of n^ϵ points $(0 < \epsilon < 1)$ and sort them according to dimension d_1. This takes $O(n)$ time. Using the sorted sample, divide the range containing the points into b ranges, called buckets. After defining the buckets, find the bucket that should contain each of the n points. This is done using binary search in $O(\log b)$ time per point. The n points are now distributed among the b buckets and the expected number of points in a bucket is $O(\frac{n}{b})$ with high probability (assuming b is $o(\frac{n}{\log n})$). The same procedure is repeated to induce a partial sorting along all dimensions. The total time taken for computing the partial sorted orders along all dimensions is $O(kn \log b)$. This is the preprocessing required in bucket-based method. This method can be viewed as a hybrid approach combining sorting and median finding. If $b = 1$, the hybrid approach is equivalent to median finding and if $b = n$, this approach is equivalent to sorting the data completely.

At any stage of the algorithm, we have a partition and k arrays storing the points of the partition partially sorted into buckets using their coordinates along each dimension. Without loss of generality, assume that the partition should be split along d_1. The bucket containing the median is easily identified in time logarithmic in the number of buckets. Finding the median translates to finding the element with the appropriate rank in the bucket containing the median. This is computed using a selection algorithm in time proportional to the number of points in the bucket. To split the partition, we need to compute the partially sorted arrays corresponding to the subpartitions. This is accomplished along d_1 by merely splitting the bucket containing the median into two buckets. To create the arrays along any other dimension d_l, each bucket in the partially sorted array along d_l is split into two buckets. All the buckets with points having a smaller coordinate along d_1 than the median are grouped into one subpartition and the rest of the buckets are grouped into the second subpartition.

When a partition is split into two subpartitions, the number of points in the partition is split into half. The number of buckets remains approximately the same (except that one bucket may be split into two) along the dimension which is used to split the partition. Along all other dimensions, the number of buckets increases by a factor of two. At a stage when level i of the tree is to be built, there are 2^i partitions and $\Theta(b2^{i(k-1)/k})$ buckets. Thus, building level i of the tree requires solving 2^i median finding problems each working on a bucket of expected size $O(\frac{n}{b2^{i(k-1)/k}})$. This time is dominated by the $O(kn)$ time to split the buckets along $k-1$ dimensions. Since the constant associated with median finding is high, this method has the advantage that it performs median finding on smaller sized data. The asymptotic complexity for building $\log m$ levels is $O(kn \log m)$.

Strategies similar to the one presented for sorting can be used for reducing the time to $O(n \log m)$. However, these strategies are not expected to perform better for small values of k such as 2 and 3 due to high overheads.

3.1.4 Experimental Results

The computation time required for local tree construction, as summarized in Table 2, can be decomposed into different parts: the time required for preprocessing (X), the cost of finding the median at every level (c), the data movement time to decompose arrays into subarrays based on the median (s). The sorting and bucket-based methods are required to have stable order property. This means that the relative ordering of data has to be preserved during the data movement, resulting in a higher value for s. Data has to be moved for $k-1$ lists in the sort-based and bucket-based strategies as compared to a single list in the median-based strategy. There is an overhead r attached with maintenance and processing of sublists, which double at every level. This overhead is the smallest for sorting, slightly larger for median-method, and highest for bucketing since the cost for

TABLE 2
Computation Time for Local Tree Construction

Method	Tree construction up to $\log m$ levels		
	Preprocessing (X)	Median Finding(c)	Data Movement (s)
Median-based	-	$O(n \log m)$	$O(n \log m)$
Sort-based	$O(kn \log n)$	$O(m)$	$O(kn \log m)$
Bucket-based	$O(kn \log b)$	$O(\frac{n}{b} \frac{m^{1/k}-1}{2^{1/k}-1})$	$O(kn \log m)$

bucketing grows exponentially with the increase in number of levels.

An important practical aspect of median finding is that the data movement step and median finding step can be combined, resulting in a small overall constant (one of the reasons quicksort has been shown to work well in practice). We limit our experimental results to the two-dimensional case because the results we obtained are such that conclusions about higher dimensional point sets can be drawn.

A comparison of sort-based method and the bucket-based methods shows that bucket-based strategy has a lower value of X, similar value of s, and much larger values of r and c. We experimented with different bucket sizes for the bucketing strategy. There is a trade-off between r and c. The former is directly proportional to the number of buckets, while the latter is inversely proportional to the number of buckets. The effect of the overhead r is per list and increases exponentially with increase in the number of levels. We found that the values of r and c are sufficiently large that sort-based method is better than the bucket-based method except when the number of levels is very small (less than four). However, for these cases, the median-based approach works better. In fact, we found that the median-based strategy is much better than the bucket-based strategy for small levels, even ignoring the time required for bucketing.

A comparison of the median and sort-based methods show that the median-based strategy has zero value of X, smaller value of s, a higher value of r, and much larger value of c, as seen in Table 2 and verified by our experimental results. One would expect the median-based method to be better for small number of levels and sort-based strategy to be better for larger number of levels, expecting the preprocessing cost to be amortized over several levels. A comparison of these methods for different data sets (of size 8K, 32K, and 128K) is provided in Fig. 1. These results show that median-based method is better than the sort-based method except when the number of levels is close to $\log N$. For larger levels, the increase due to higher c becomes significant for the median-based method. These results also show that when data is already sorted, the sort-based method has a better time than the median-based method.

Median-based strategy is the best unless the number of levels is close to $\log N$ or if the data is already available sorted, for which the sort-based strategy is the best. For larger values of k ($k \geq 3$), using a median-based strategy would be comparable to or better than the other strategies unless preprocessing information used for sorting method is already available.

3.2 Parallel Tree Construction for $\log p$ Levels

3.2.1 Sort-Based Method

In the parallel algorithm, each processor is given $\frac{N}{p}$ elements. The elements are sorted using a parallel sorting algorithm to create the sorted arrays $A_1, A_2, \ldots, A_k$ distributed evenly among the processors. We use a variation of parallel sample sort [14] because of its practical efficiency. The total time required for one sort is

$$O\left(\frac{N}{p}\log N + p\log^2 p + t_s p + t_w\left(\frac{N}{p} + p^2\right) + t_h p \log p\right)$$

on a hypercube and

$$O\left(\frac{N}{p}\log N + p\log^2 p + t_s\sqrt{p} + t_w\left(\frac{N}{\sqrt{p}} + p^2\right) + t_h\sqrt{p}\right)$$

on a mesh. For large values of N ($N \geq O(t_s p^2 + t_w p^3)$), the running time is $O(\frac{N}{p}\log N + t_w \frac{N}{P})$ on a hypercube and $O(\frac{N}{p}\log N + t_w(\frac{N}{\sqrt{p}}))$ on a mesh.

Once the data is preprocessed by sorting it along each dimension to create k sorted arrays, the work of finding

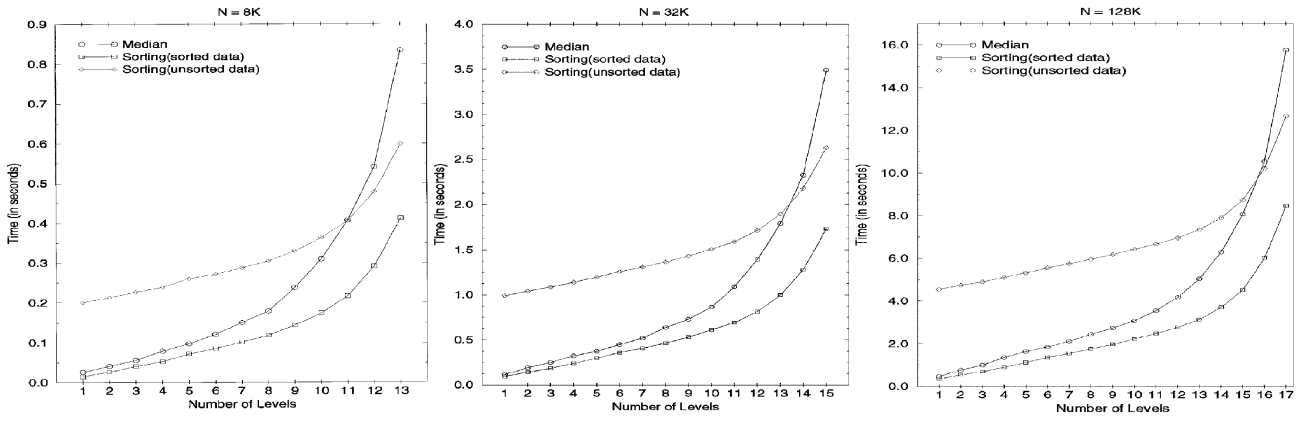


Fig. 1. Local tree construction for random distribution of data of sizes 8K, 32K, and 128K.

medians is completely eliminated. Without loss of generality, assume that the partition should be split along dimension d_1. The processor containing the median coordinate along d_1 broadcasts it to all the processors. We want to split the partition into two subpartitions and assign one subpartition to half of the processors and assign the second subpartition to the other half. Assigning a subpartition amounts to computing the sorted arrays for the subpartition. This is already true along d_1. For any other dimension d_l, each processor scans through its part of the array sorted by d_l and splits it into two subarrays depending upon the coordinate along d_1. All the points less than the median d_1 coordinate are moved to the first half of the processors with an algorithm similar to order–maintaining data movement. Points greater than the median d_1 coordinate are moved to the second half of the processors. Once the initial sorted arrays are computed, splitting partitions at every node of the tree merely requires moving the elements of the arrays to the appropriate processors.

Consider the time required for building the first $\log p$ levels of the tree: At level i of the tree, we are dealing with 2^i partitions containing $\frac{N}{2^i}$ points each. A partition is represented by k sorted arrays distributed evenly on $\frac{p}{2^i}$ processors. Splitting the local arrays and preparing the data for communication requires $O((k-1)k\frac{N}{p})$ time. This is because the $k-1$ arrays can potentially contain different records and the size of each record is $O(k)$. The required data movement must be done using Order maintaining data movement operation since the sorted order of the data must be preserved.

Given sorted data, the time required to build the first $\log p$ levels of the tree on a hypercube is

$$\sum_{i=0}^{\log p-1} O\left(k(k-1)\frac{N}{p}+t_s\frac{p}{2^i}+t_w k(k-1)\frac{N}{p}+t_h\frac{p}{2^i}\log\frac{p}{2^i}\right)$$
$$= O\left(k(k-1)\frac{N}{p}\log p+t_s p+t_w k(k-1)\frac{N}{p}\log p+t_h p\log p\right).$$

The corresponding time on a mesh is

$$\sum_{i=0}^{\log p-1} O\left(k(k-1)\frac{N}{p}+t_s\sqrt{\frac{p}{2^i}}+t_w k(k-1)\frac{N}{p}\sqrt{\frac{p}{2^i}}+t_h\sqrt{\frac{p}{2^i}}\right)$$
$$= O\left(k(k-1)\frac{N}{p}\log p+t_s\sqrt{p}+t_w k(k-1)\frac{N}{\sqrt{p}}+t_h\sqrt{p}\right).$$

For large values of N ($N \geq O(t_s p^2 + t_w p^3)$), the total time required by the algorithm is $O(k(k-1)t_w\frac{N}{p}\log N)$ for the hypercube and $O(k(k-1)t_w(\frac{N}{\sqrt{p}}))$ on the mesh. As in the sequential sort-based method, it is possible to reduce the computational cost at every level from $O(kn)$ to $O(n)$. However, the method requires dereferencing pointers to obtain points on other processors. The resulting communication makes this method impractical even for large values of k.

3.2.2 Median-Based Method

We compared the different deterministic and randomized parallel selection algorithms for coarse grained machines [1] and found that a straightforward parallelization of Floyd's sequential algorithm results in the best performance for random data. Nonrandom data can be randomized by allocating each point to a random processor and using a transportation primitive to move points to appropriate processors. The cost of this randomization is insignificant compared to the cost of construction of the k-d tree.

Let $N_i^{(j)}$ be the number of elements in processor P_i at the beginning of iteration j of the median finding algorithm. Let $N^{(j)} = \sum_{i=0}^{p-1} N_i^{(j)}$. Let $k^{(j)}$ be the rank of the desired element. All processors use the same random number generator with the same seed to produce identical random numbers. Consider the behavior of the algorithm in iteration j. First, a parallel prefix operation is performed on the $N_i^{(j)}$s. A random number between 1 and $N^{(j)}$ is used to pick the estimated median. From the parallel prefix operation, each processor can determine if it has the estimated median, and if so, broadcasts it. Each processor scans through its set of points and splits them into two subsets based on the estimated median. A Combine operation and a comparison with $k^{(j)}$ determines which of these two subsets is to be discarded and the value of $k^{(j+1)}$ needed for the next iteration.

Let $N_{max}^{(j)} = max_{i=0}^{p-1} N_i^{(j)}$. Thus, splitting the set of points into two subsets based on the median requires $O(N_{max}^{(j)})$ time in the jth iteration. For random data, it can be shown that the number of remaining points left after each iteration are mapped equally among all the processors with high probability (i.e., the maximum is close to mean) unless the number of remaining points is very small. Thus, the total expected time spent in computation is $O(\frac{N}{p})$. Another option is to ensure that a load balancing is done after every iteration. However, such a load balancing always results in an increase in the running time [1].

The number of iterations required for N points is $O(\log N)$ with high probability. Therefore, the expected running time of parallel median finding, is

$$O\left(\frac{N}{p}+(t_s+t_w)\log p\log N\right)$$

on the hypercube and

$$O\left(\frac{N}{p}+(t_s+t_w)\log p\log N+t_h\sqrt{p}\log N\right)$$

on the mesh.

In building the first $\log p$ levels of the tree, the task at level i of the tree is to solve 2^i median finding problems in parallel with each median finding involving $\frac{N}{2^i}$ points and $\frac{p}{2^i}$ processors. After finding the median of $\frac{N}{2^i}$ points on

$\frac{p}{2^i}$ processors, all the elements less than or equal to the median are moved to the first $\frac{p}{2^{i+1}}$ processors, while the other elements are move to the next $\frac{p}{2^{i+1}}$ processors. The maximum number of elements sent out or received by any processor is $\frac{N}{p}$. We assume that this data movement results in a random distribution of the two lists to the two subsets of processors. This can be ensured by randomly permuting the data without increasing the asymptotic complexity. Even if pointers are used for local computation instead of records, the records have to be communicated when data is moved across processors.

Building the first $\log p$ levels of the tree on the hypercube requires

$$\begin{aligned}\sum_{i=0}^{\log p-1} O\Bigg(&\frac{N}{2^i}/\frac{p}{2^i}+(t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i}+k\frac{N}{p}\\ &+t_s\frac{p}{2^i}+t_w\frac{kN}{p}+t_h\frac{p}{2^i}\log\frac{p}{2^i}\Bigg)\\ &=O\Bigg(\frac{kN}{p}\log p+t_s(p+\log^2 p\log N\Bigg)\\ &+t_w\Bigg(k\frac{N}{p}\log p+\log^2 p\log N)+t_h p\log p\Bigg)\end{aligned}$$

time. The time required on the mesh is

$$\begin{aligned}\sum_{i=0}^{\log p-1} O\Bigg(&\frac{N}{2^i}/\frac{p}{2^i}+(t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i}+t_h\sqrt{\frac{p}{2^i}}\log\frac{N}{2^i}+k\frac{N}{p}\\ &+\Bigg(t_s+t_w\frac{kN}{p}\Bigg)\sqrt{\frac{p}{2^i}}+t_h\sqrt{\frac{p}{2^i}}\Bigg)\\ &=O\Bigg(\frac{kN}{p}\log p+t_s(\sqrt{p}+\log^2 p\log N)\\ &+t_w\Bigg(\frac{kN}{\sqrt{p}}+\log^2 p\log N\Bigg)+t_h\sqrt{p}\log N\Bigg).\end{aligned}$$

For large values of N ($N \geq O(t_s p^2 + t_w p^3)$), the total time required by the algorithm is $O(kt_w\frac{N}{p}\log N)$ for the hypercube and $O(kt_w(\frac{N}{\sqrt{p}}))$ on the mesh. Thus, the data movement time dominates.

3.2.3 Bucket-Based Method

To use this method, we first create the required bucketing using p processors and construct the first $\log p$ levels of the tree in parallel as before. The required bucketing along a dimension, say d_1, can be computed as follows: Select a total of n^ϵ points ($0 < \epsilon < 1$) and sort them along d_1 using a standard sorting algorithm such as bitonic merge sort. The time for bitonic merge sort on p processors is

$$O\Bigg(\frac{N^\epsilon\log N^\epsilon}{p}+\frac{N^\epsilon}{p}\log^2 p+\Bigg(t_s+t_w\frac{N^\epsilon}{p}\Bigg)\log^2 p\Bigg)$$

time on a hypercube and

$$O\Bigg(\frac{N^\epsilon\log N^\epsilon}{p}+\frac{N^\epsilon}{p}\log^2 p+\Bigg(t_s+t_w\frac{N^\epsilon}{p}\Bigg)\sqrt{p}\Bigg)$$

on a mesh. Using the sorted sample, divide the range containing the points into p intervals called buckets. Using a global concatenate operation, the p intervals are stored on each processor. Each processor scans through its $\frac{N}{p}$ points and for each point determines the bucket it belongs to in $O(\frac{N}{p}\log p)$ time. The points are thus split into p lists, one for each bucket. It is desired to move all the lists belonging to bucket i to processor P_i. The points are sent to the appropriate processors using the transportation primitive. The expected number of points per bucket is $O(\frac{N}{p})$, with high probability. Apart from the bitonic sort time, the time for bucketing is $O(\frac{N}{p}(k+\log p)+t_s p+t_w\frac{kN}{p}+t_h p\log p)$ on a hypercube and $O(\frac{N}{p}(k+\log p)+t_s\sqrt{p}+t_w\frac{kN}{\sqrt{p}}+t_h\sqrt{p})$ on a mesh. Clearly, this dominates the time for bitonic sorting on both the mesh and hypercube.

Consider building the first $\log p$ levels of the tree. Suppose that the first split is along dimension d_1. By a parallel prefix operation, the bucket containing the median is easily identified. The median is found by finding the element with the appropriate rank in this bucket using the sequential selection algorithm. The median is then broadcast to all the processors which split their buckets along all other dimensions based on the median. Using order–maintaining data movement, the buckets are routed to the appropriate processors. Since the bucket size is smaller than the number of elements in a processor, the time for locating the median in the bucket is dominated by the time for splitting the buckets along each dimension. The first $\log p$ levels of the tree can be built in

$$\begin{aligned}&\sum_{i=0}^{\log p-1} O\Bigg(k(k-1)\frac{N}{p}+t_s\frac{p}{2^i}+t_w k(k-1)\frac{N}{p}+t_h\frac{p}{2^i}\log\frac{p}{2^i}\Bigg)\\ &=O\Bigg(k(k-1)\frac{N}{p}\log p+t_s p+t_w k(k-1)\frac{N}{p}\log p+t_h p\log p\Bigg)\end{aligned}$$

time on a hypercube and

$$\begin{aligned}&\sum_{i=0}^{\log p-1} O\Bigg(k(k-1)\frac{N}{p}+t_s\sqrt{\frac{p}{2^i}}+t_w k(k-1)\frac{N}{\sqrt{p}}+t_h\sqrt{\frac{p}{2^i}}\Bigg)\\ &=O\Bigg(k(k-1)\frac{N}{p}\log p+t_s\sqrt{p}+t_w k(k-1)\frac{N}{\sqrt{p}}\log p+t_h\sqrt{p}\Bigg)\end{aligned}$$

time on a mesh.

3.2.4 Experimental Results

In this section, we compare the three algorithms experimentally using implementations on the CM-5 for which most of the analysis presented for the hypercube is applicable. The execution time is decomposed into several parts, as summarized in Table 3 (the t_h term is insignificant in the overall communication time and is ignored): the time required for preprocessing (X), the cost of finding the median at every level (c), the local processing time to rearrange data based on the median (s), and the communication due to

TABLE 3
Time for Tree Construction up to $\log p$ Levels on p Processors on a Hypercube

Method	Preprocessing (X)	Median finding (c)	Local processing (s)	Communication du e to data m ovcment(T)
Median	–	$O(\frac{N}{p}\log p + (t_s + t_w)\log^2 p \log N)$	$O(k\frac{N}{p}\log p)$	$O(t_s p + t_w(k\frac{N}{p}\log p))$
Sort	$O(\frac{N}{p}\log N + p^2 \log p + t_s p + t_w(\frac{N}{p} + p^2))$	$O(\log p)$	$O(k(k-1)\frac{N}{p}\log p)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}\log p))$
Bucket	$O(\frac{N}{p}(k + \log p) + t_s p + t_w(k\frac{N}{p}))$	$O(\frac{N}{p})$	$O(k(k-1)\frac{N}{p}\log p)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}\log p))$

data movement (T). The data movement cost for $k-1$ arrays is expected to be higher for the bucket and sort-based methods, as they require preserving the order of the data as opposed to a non-order-maintaining data movement for only one array in the median-based method. Overhead cost r is associated with maintaining and processing sublists at every level on a per list basis and grows exponentially with the increase in number of levels.

Once again, we limit our experimental results to the two-dimensional case, as it proved to be sufficient to draw conclusions for higher dimensional problems. A comparison of sort-based and median-based approaches shows that the former has higher values of X, s, and T, and a smaller value of r. For small values of $\frac{N}{p}$, the median-based method is not expected to parallelize well as the communication cost in c dominates, which increases with an increase in p. However, for small p, the median-based method performs well when it offsets the large value of X in the sort-based method. For large values of $\frac{N}{p}$, one would expect median finding to parallelize reasonably well. The communication cost in median finding is significantly lower than T, required for both strategies. The value of T is smaller for the median-based approach as compared to the sort-based approach since it does non-order-maintaining data movement. One would expect the median-based method to perform better, except for very large values of p.

A comparison of bucket-based and median-based finding approaches shows that the former has larger values of X, s, r, and T. For small values of $\frac{N}{p}$, we would expect the median-based method to perform worse than the bucket-based approach, which has small communication overhead for median finding unless p is small. A small p would keep c low for median-based method, but X will still be large for the bucket-based method. For large values of $\frac{N}{p}$, c in median finding should not dominate the overall cost and should be significantly lower than T, required for both strategies at every level. Bucket-based method requires order maintaining data movement, and hence, has a higher T than the median-based method. Hence, one would expect the median-based method to be better than the bucket-based method, except for a very large p.

The bucket-based method has lower values for X and c when compared to the sort-based method. Hence, the bucket-based method would work better when the difference in preprocessing time is larger than the total time for median finding. The cost of the latter decreases at every level (as the bucket sizes decrease with increase in number of levels).

Fig. 2 presents experimental results for k-d tree construction up to $\log p$ levels for different values of $\frac{N}{p}$. These show that the bucket-based method is always better than the sort-based method. The median-based method is the best approach for large values of $\frac{N}{p}$ (greater than 8K per

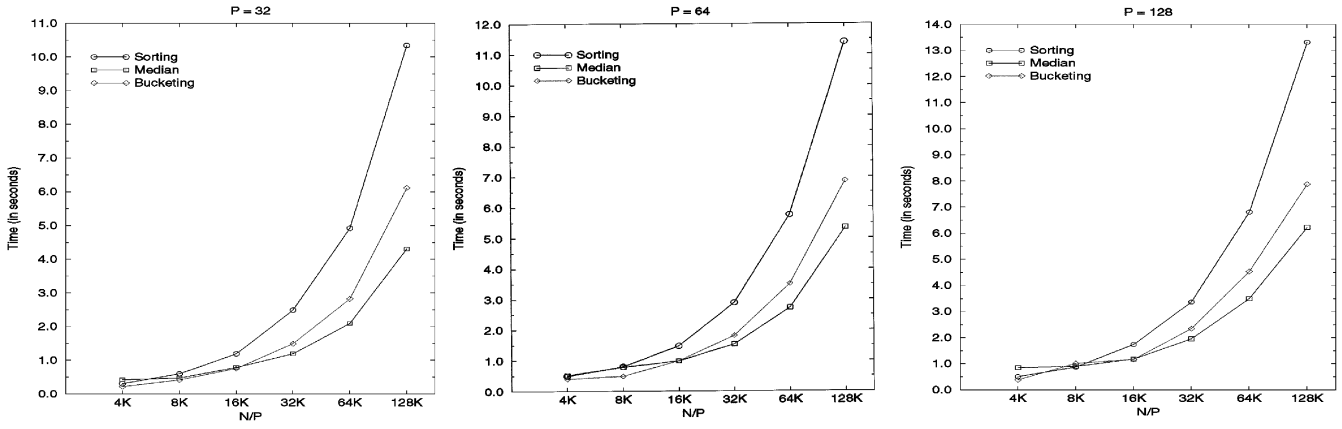


Fig. 2. Tree construction to $\log p$ levels for p = 32, 64, and 128 using random distribution of data.

processor) while the bucket-based approach is the best for small values of $\frac{N}{p}$ (less than 4K). The improvements of each of these methods over the other are substantial for these ranges. For larger values of k, it is expected that the time requirements of the bucket-based strategy would grow faster than the median-based strategy due to overheads in the lists and bucket management.

We emphasize that the above conclusions are drawn for the randomized median-based method. The constants involved in a deterministic algorithm for median finding may make sort-based and bucket-based methods better for small values of k and for a wide range of $\frac{N}{p}$.

3.3 Global Tree Construction

The parallel tree construction can be decomposed into two parts: constructing the tree till $\log p$ levels followed by local tree construction. A different strategy potentially can be used for the two parts. This results in at least nine possible combinations. Based on the discussion in the previous two sections, the following are the only viable options to be considered for the global tree construction:

G1 Sort-based approach up to $\log p$ levels, followed by using sort-based approach locally. Sort-based approach up to $\log p$ levels has an added benefit that the local data is left sorted. Thus, no preprocessing is required for local tree constructions. Also using any other approach for local tree construction would not be better due to this reason.

G2 Median-based approach up to $\log p$ levels, followed by using sort-based approach locally.

G3 Median-based approach up to $\log p$ levels, followed by using a median-based approach locally.

G4 Bucket-based approach up to $\log p$ levels, followed by using a median-based approach locally.

G5 Bucket-based approach up to $\log p$ levels, followed by sorting each of the buckets, followed by using a sort-based approach locally.

These five approaches are compared for a different number of levels ($\log p$ to $\log N$), for a different number of processors (8, 32, 128), and for different values of N/p (4K, 16K, and 128K) (see Fig. 3). These results show that for large values of N/p, strategy G3 is the best unless the number of levels are close to $\log N$, for which G2 may be preferable. For small values of N/p, the strategy G4 is preferable. If the number of levels are close to $\log N$, G1 and G5 are the best. Total run-times as a function of the number of tree levels using strategy G2 are shown in Fig. 4.

For larger values of k, we would expect that one of the median or bucket-based strategies should be used to construct the tree until $\log p$ levels. This would depend on the value of N/p and the target architecture. The bucket-based strategy would be better for small values of $\frac{N}{p}$ and k. The local tree construction should use median-based strategy. Using the above approach should result in software which will be close to the best for nearly all values of the parameters.

4 Reducing the Data Movement

The algorithms described for parallel construction of the first $\log p$ levels of the tree require massive data movement using transportation primitive at every level. This can be significantly reduced by using a different approach.

Consider the median-based method. Initially, all the N points belong to one partition and are distributed uniformly on all p processors. After finding the median, the local data is divided into two subarrays (typically of unequal size), each belonging to one of the subpartitions. Instead of moving the data such that each subpartition is assigned to a different subset of processors, one can assume that these subpartitions are divided among all the processors. A median can be found for each of the subpartitions in parallel by combining the communication and computation for both. Because each processor always has a total of $\frac{N}{p}$ points, computational load is perfectly balanced among all the processors. This approach is repeatedly applied until the number of subpartitions is equal to p. At this stage, the data is distributed among the processors such that each processor has all the points of one of the p subpartitions and local trees are constructed.

Suppose that i levels of the tree are already constructed. At this stage, there are 2^i subpartitions, each divided among all the processors. It is desired to find the medians for all the subpartitions together. A parallel prefix operation is performed for each of the subpartitions to number the points in each processor belonging to a subpartition. All the 2^i parallel prefix operations can be combined together. By generating appropriate random numbers, each processor determines if it has the estimated median of each subpartition. A processor updates the corresponding entry in an array of size 2^i if it has guessed the median for the ith subpartition, otherwise it stores a zero. By a combine operation on this array using the "+" operation, the required medians are stored on each processor. Using the 2^i estimated medians, all processors together reduce the size of the subpartitions under consideration. The iterations are repeated until the total size of all the subpartitions falls below a constant. At this stage, all the subpartitions can be gathered in one processor and the required medians can be found.

The cost of this algorithm in constructing level $i+1$ of the tree from level i is $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ on a hypercube, and

$$O\left(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N\right)$$

on a mesh. Constructing the first $\log p$ levels of the tree on a hypercube requires

$$\sum_{i=0}^{\log p - 1} O\left(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N\right)$$

time, plus $O(\frac{kN}{p} + t_s p + t_w \frac{kN}{p} + t_h p \log p)$ time for the final data movement. Thus, the run time is

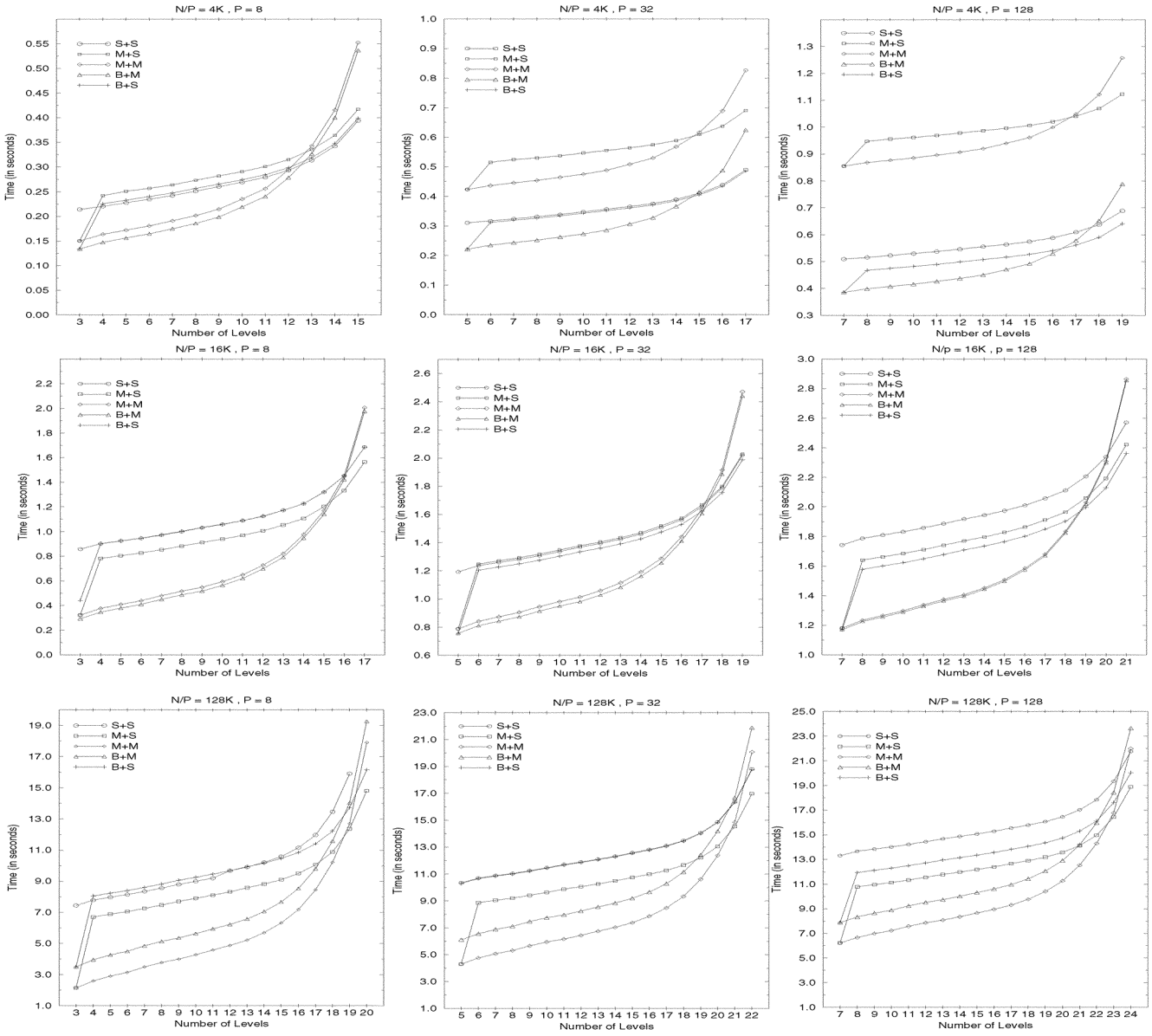


Fig. 3. Global tree construction for random distribution of data of size $\frac{N}{p}$ = 4K, 16K, 128K on $p = 8, 32, 128$ processors. The X-axis represents the number of levels up to which the tree is built.

$$O\left(\frac{N}{p}(\log p + k) + t_s(p + \log^2 p \log N) + t_w\left(\frac{kN}{p} + p \log p \log N\right) + t_h p \log p\right).$$

The corresponding time on the mesh is

$$\sum_{i=0}^{\log p - 1} O\left(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N\right) + O\left(\frac{kN}{p} + t_s\sqrt{p} + t_w \frac{kN}{\sqrt{p}}\right)$$
$$= O\left(\frac{N}{p}(\log p + k) + t_s(\sqrt{p} + \log^2 p \log N) + t_w\left(\frac{kN}{\sqrt{p}} + p \log p \log N\right) + t_h \sqrt{p} \log p \log N\right).$$

For both hypercubes and meshes, the computational cost reduces from $\frac{kN}{p} \log p$ to $\frac{N}{p}(k + \log p)$. This is the cost involved in local computations plus the cost in copying the records to arrays for data movement. For hypercubes (or permutation networks), the amount of data transferred reduces by a factor of $\log p$ from $\frac{kN}{p} \log p$ to $\frac{kN}{p}$. However, the data transferred improves only by a small constant factor on the mesh. A similar strategy can be used to reduce the data movement for sort-based and bucket-based methods.

5 Experimental Results

We limited ourselves to applying this strategy to the median-based approach only. Fig. 5 gives a comparison of the two approaches (with and without data movement) for different values of N/p for $\log p$ levels of parallel tree construction. These results show that using the new

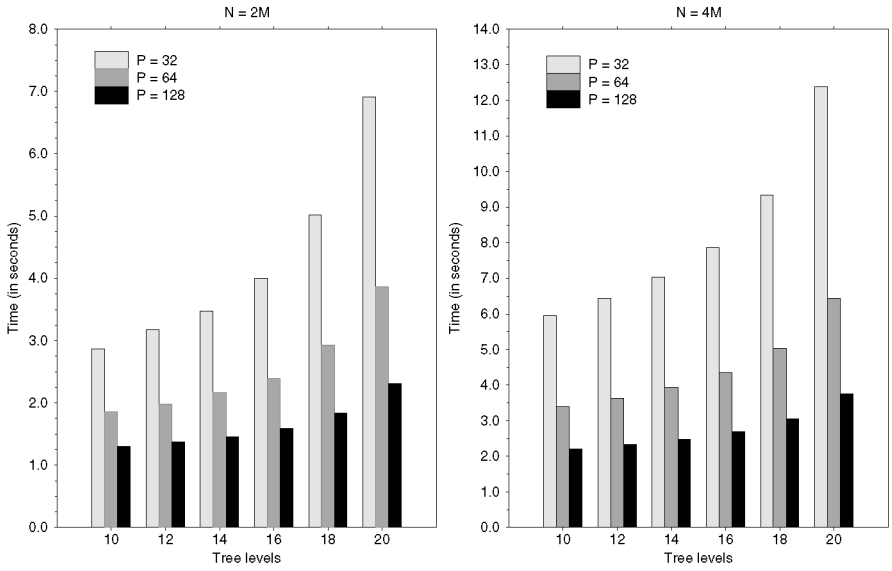


Fig. 4. Total time for tree construction on 32, 64, and 128 processors up to the specified level on 2M and 4M random distribution of data.

strategy gives significant improvements due to lower data movement. The cost of the simple scheme is approximately 50 percent more than the new scheme for large data sets. CM-5 without vector units has a very good ratio of unit computation to unit communication cost. This ratio is much higher for typical machines. For these machines, the improvement of the new strategy should be much better.

The median-based method with reduced data movement potentially reduces the data movement cost by a factor of $\log P$. Since the data movement cost is proportional to the size k of the individual records storing points, the effect of extra overhead due to communication in the new method reduces significantly for higher dimensional data. Thus, the new strategy should give improved performance as the number of dimensions is increased.

6 Conclusions

In this paper, we have considered various strategies for parallel construction of multidimensional binary search trees. Traditionally, a sort-based strategy is advocated for sequential construction [10] or parallel construction [4] of complete k-d trees. For two–dimensional point sets, we found that the the median-based strategy is the fastest for large values of $\frac{N}{p}$, unless the number of levels is very close to $\log N$. In such a case, using the median-based approach up to $\log p$ levels followed by using a sort-based approach locally performed better. For small values of $\frac{N}{p}$, using the bucket-based approach up to $\log p$ levels followed by using a median-based approach locally is the best, except when the tree is built almost completely. In such a case, using the bucket-based method followed by using sort-based method locally outperformed a complete sort-based strategy, except when the number of processors is very small.

It is interesting to note that a complete sort-based approach did not perform better even if the tree is built completely. The median-based approach also exhibits good scaling, as can be seen by the run-time analysis and experimental results. This is true mainly because of the performance of randomized median finding on randomly

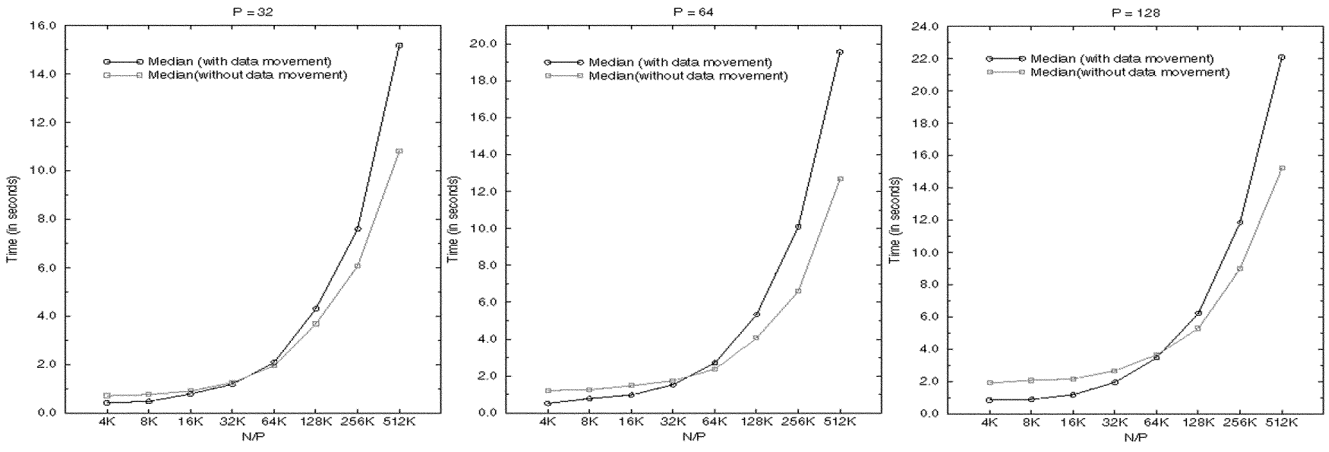


Fig. 5. Median-based tree construction for $\log p$ levels with and without reduced data movement.

distributed data sets. For arbitrary data sets, a randomization step can be performed without much additional cost. We found that deterministic median finding algorithms are slower by an order of magnitude and their use would lead to entirely different conclusions. Bucket-based strategy is useful for applications such as graph partitioning when the number of points per processor is small.

For data in k dimensions, the preprocessing cost and the cost of tree construction per level of sort-based and bucket-based methods increases proportional to k. The median-based method remains unaffected in this regard. While the data movement time in median-based methods increases proportional to k, it increases proportional to $k(k-1)$ in other methods. Thus, based on the dismal performance of sort-based and bucket-based methods for $k = 2$, we conclude that the median-based method is superior for $k \geq 3$.

Our experiments with comparing median finding with data movement at every stage and median finding with reduced data movement associate well with the idea of task versus data parallelism. For this, we showed that utilizing task parallelism leads to worse results as compared to "concatenated" data parallelism for large granularities. Often, problems amenable to the divide and conquer paradigm are solved in parallel by mapping the corresponding divide and conquer tree using task parallelism. Our technique can be effectively utilized to solve such problems efficiently, especially when the task sizes are nonuniform. We are currently exploring this strategy.

Random distribution of data has potential advantages in a number of applications. We feel that data-parallel languages such as High Performance Fortran should provide constructs for random distribution to facilitate coding such applications.

A generalized version of the problem we have considered in this paper is the construction of weighted multidimensional binary search trees. In this case, each point has a weight associated with it and a partition is split based on the weighted median. Therefore, the number of points in the subpartitions at level i of the tree can be very nonuniform. In such a case, our method with reduced data movement is clearly superior, since it keeps the number of points on each processor balanced throughout the construction of the tree.

If a perfectly balanced k-d tree is not required, a random sampling based approach can be used to reduce the overall cost significantly. Choose a small random sample of all the points. Construct the tree for this random sample using one of the methods described in the paper up to a fixed number of levels. Assign the remaining points to one of the leafs based on the "sample" tree. Each leaf node can be constructed recursively using the same strategy.

ACKNOWLEDGMENTS

This research was supported in part by a scholarship from the King Abdulaziz City for Science and Technology (KACST), as well as funding from the U.S. National Science Foundation CAREER under CCR-9702991, U.S. Army Research Office under DAAG55-97-1-0368 and DAAG55-97-1-0368, and Sandia National Laboratories. We would like to thank Northeast Parallel Architectures Center at Syracuse University and AHPCRC at the University of Minnesota for access to their CM-5. We would also like to thank S. Rajasekharan for several discussions on related topics.

REFERENCES

[1] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka, "Practical Algorithms for Selection on Coarse-Grained Parallel Computers," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 8, pp. 313-324, Aug. 1997.

[2] J.L. Bentley, "Multidimensional Binary Search Trees in Database Applications," *IEEE Trans. Software Eng.,* vol. 5, pp. 333-340, 1979.

[3] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Search," *Comm. ACM,* vol. 19, pp. 509-517, 1975.

[4] G.E. Blelloch, *Vector Modles for Data-Parallel Computing.* Cambridge, Mass.: MIT Press, 1990.

[5] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *J. Computer Systems Science,* vol. 7, pp. 448-461, 1972.

[6] F. Ercal, "Heuristic Approaches to Task Allocation for Parallel Computing," PhD Thesis, Ohio State Univ., 1988.

[7] R.W. Floyd and R.L. Rivest, "Expected Time Bounds for Selection," *Comm. ACM,* vol. 18, pp. 165-172, 1975.

[8] G. Fox, *Solving Problems on Concurrent Processors: Volume I–General Techniques and Regular Problems.* Englewood Cliffs, N.J.: Prentice Hall, 1988.

[9] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms.* San Francisco: Benjamin Cummings, 1994.

[10] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction.* New York: Springer-Verlag, 1985.

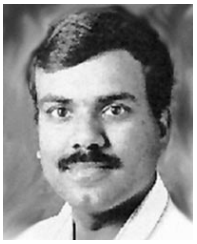
[11] S. Rajasekharan, W. Chen, and S. Yooseph, "Unifying Themes for Parallel Selection," *Proc. Fifth Int'l Symp. Algorithms and Computation,* 1994.

[12] S. Ranka, R.V. Shankar, and K.A. Alsabti, "Many-to-Many Communication with Bounded Traffic," *Proc. Frontiers of Massively Parallel Computation,* pp. 20-27, 1995.

[13] A. Schonhage, M.S. Paterson, and N. Pippenger, "Finding the Median," *J. Computer Systems Science,* vol. 13, pp. 184-199, 1976.

[14] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *J. Parallel and Distributed Computing,* vol. 14, pp. 361-370, 1992.

Ibraheem Al-furaih earned the BS in computer science from King Saud University, Saudi Arabia, in 1992. From 1992-1993, he worked as a researcher in King Abdulaziz City for Science and Technology (KACST), Saudi Arabia, which gave him a scholarship with which to continue his graduate studies. He earned the MS in computer science from Syracuse University, Syracuse, New York, in 1995. He is currently a PhD candidate at Syracuse University. His research interests include parallel computing, high-performance programming, and distributed systems.



Srinivas Aluru received the BTech degree in computer science and engineering from the Indian Institute of Technology, Madras, India, in 1989, and the MS and PhD degrees in computer science from Iowa State University in 1991 and 1994, respectively. Dr. Aluru is an assistant professor in the Department of Electrical and Computer Engineering at Iowa State University. Prior to this, he worked as an assistant professor in the Department of Computer Science at New Mexico State University. From 1994 to 1996, he worked as a visiting assistant professor at the School of Computer and Information Science at Syracuse University, Syracuse, New York. His research interests include parallel algorithms and applications, scientific computing, computational biology, and multidimensional databases. He is a recipient of the U.S. National Science Foundation CAREER award. He is a member of the ACM and the IEEE.

Sanjay Goil received the BTech degree in computer science from Birla Institute of Technology and Science, Pilani, India, in 1990, and the MS degree in computer science from Syracuse University, Syracuse, New York, in 1995. From 1991 to 1993, he was a research associate in the networked computing department at Bell Laboratories. He is currently a PhD student in the Computer Engineering Department and the Center for Parallel and Distributed Computing at Northwestern University. His research interests are in the areas of parallel and distributed computing, parallel algorithms, and high performance I/O for large databases and data mining.

Sanjay Ranka received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kanpur, in 1985, and the PhD degree in computer and information science from the University of Minnesota, Minneapolis, in 1988. He is a professor in the Department of Computer Science at the University of Florida, Gainesville. Prior to joining the University of Florida, he was an associate professor at Syracuse University from 1988 to 1995. His main research areas include models of parallel computation, parallel algorithms, runtime support for compilers, high-performance computing, and neural networks. Prof. Ranka has coauthored more than 120 papers, at least 40 of which have appeared in archival journals. He serves on the editorial board of the *Journal of Parallel and Distributed Computing (Algorithms and Scientific Computing)*. He has served on the program committees of several conferences, organized several workshops, and guest-edited special issues. He was one of the main architects of the Fortran 90D/High Performance Compiler. He is a member of the Parallel Compiler Runtime Consortium and a past member of the Message Passing Initiative Standards Committee. He is a coauthor of the books *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition* (Springer-Verlag, 1990) and *Elements of Artificial Neural Networks* (MIT Press, 1996).