Parallel Construction of Multidimensional Binary Search Trees

Ibraheem Al-furaih, Srinivas Aluru, Sanjay Goil
School of CIS and
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244-4100
email: *alfuraih, aluru, sgoil@top.cis.syr.edu*

Sanjay Ranka
School of CISE

University of Florida
Gainesville, FL 32611
ranka@cis.ufl .edu

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 present several algorithms for parallel k-d tree construction and analyze them theoretically and experimentally. We have implemented our algorithms on the CM-5 and report on the experimental results.

1 Multidimensional Binary Search Trees

Consider a set of n points in k dimensional space. Let $d_1, d_2, \ldots, d_k$ denote the k dimensions. If we find the median coordinate of all the points along dimension d_1, we can partition the points into two approximately equal sized sets - one set containing all the points whose coordinates along dimension d_1 are less than or equal to this median and a second set containing all the points whose coordinates along dimension d_1 are greater than the median. Each of these two sets is again partitioned using the same process except that the median of each set along dimension d_2 is used. The partitioning is continued using dimensions $d_1, d_2, \ldots, d_k$ in that order until each resulting set contains one point. If all the dimensions are exhausted before completely partitioning the data, we "wrap around" and reuse the dimensions again starting with d_1. An example of such a partition for a two-dimensional data set containing 8 points is shown in Figure 1.

The process of organizing spatial data in the above manner is naturally represented by a binary tree. The root of the tree corresponds to the set containing all the n points. Each internal node corresponds to a set of points and the two subsets obtained by partitioning this set are represented by its children. The same dimension is used for partitioning the set of each internal node at the same level of the binary tree. For all nodes at level i of the tree (defining the root to be at level 0), dimension $d_{(i \bmod k)+1}$ is used. The resulting tree is called a multidimensional binary search tree (abbreviated k-d tree), first introduced by J.L. Bentley [3]. The k-d tree corresponding to the partitioning in Figure 1 is shown in Figure 2.

The above process of construction always produces a balanced k-d tree. Although k-d tree is not necessarily a balanced data structure, accessing and manipulating k-d trees is more efficient on balanced trees. It is difficult to keep the tree balanced if insertions and deletions of points are allowed. However, if all the data is known in advance, it is easy to construct a balanced tree using the above process. Also, it is not necessary that dimensions be used in some particular order or that the same dimension be used for all nodes at the same level of the tree. A common variation is to use the dimension with the largest span, defined to be the difference between maximum and minimum coordinates of all the points along a given dimension. Another variation is to use internal nodes to store points. With this, one of the points corresponding to the median coordinate is stored in the node and the other points are split into two sets. Such a tree where all the nodes store a point each is called a homogeneous tree. k-d trees where only the leaves are used to store data are called non-homogeneous trees.

In this paper, we focus on efficient parallel construction of balanced, non-homogeneous 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. A coarse-grained parallel computer consists of several relatively powerful processors connected by an interconnection network. Most of the commercially available parallel computers belong to

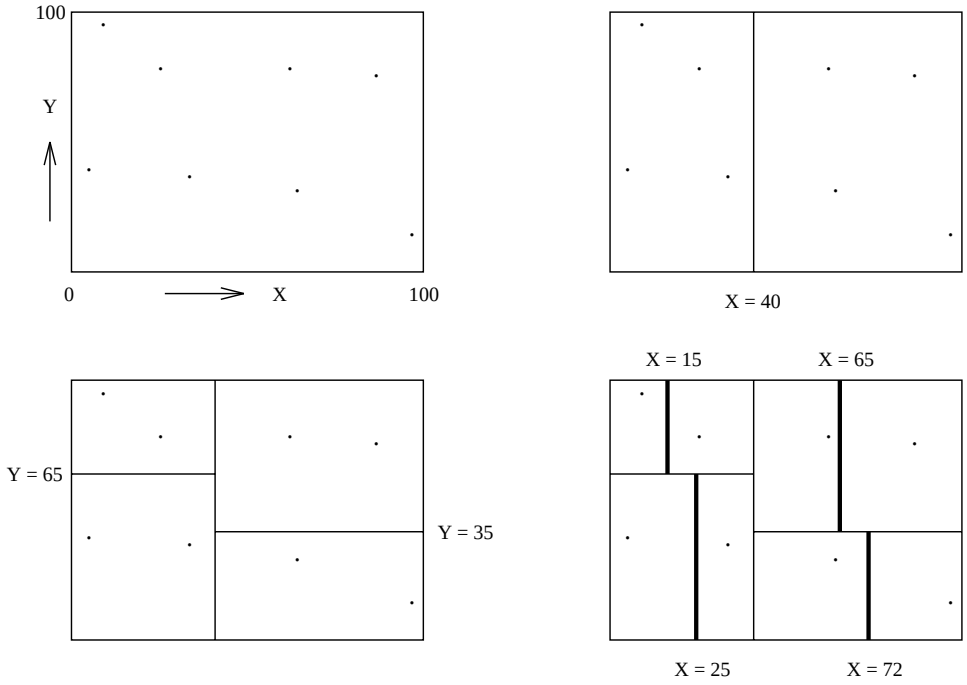

Figure 1: Recursive data partitioning of a two-dimensional data set.

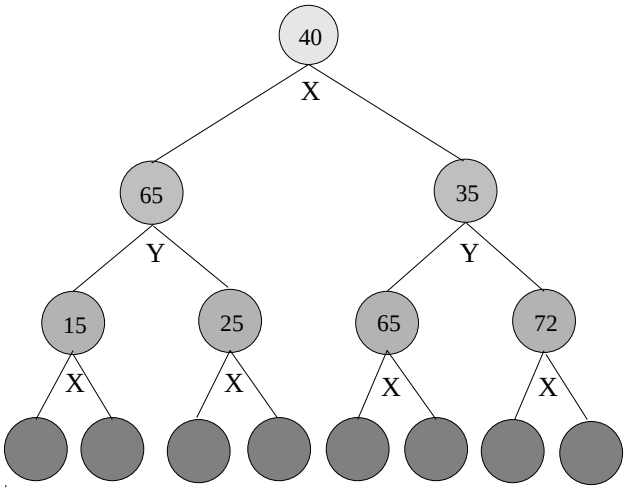

Figure 2: The k-d tree corresponding to the partitioning of data in Figure 1.

this category. Instead of making specific assumptions about the network connecting processors, we describe our algorithms in terms of some basic communication primitives. The running time of our algorithms on a specific interconnection network can be easily derived by substituting the running times of the communication primitives. We provide such an analysis for hypercubes and meshes.

The rest of the paper is organized as follows: In Section 2, we outline several applications that use k-d trees. In Section 3, we describe our model of parallel computation and outline some primitives used by our algorithms. Section 4 describes four different 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 5 presents an algorithm with reduced data movement and Section 6 concludes the paper.

2 Applications

Several applications require the construction of k-d trees only up to a specified number of levels. To motivate the need for parallel construction of k-d trees and to see why partial construction may be useful, we describe the usefulness of k-d trees for the following applications.

Graph Partitioning A large number of scientific and engineering applications involving iterative methods can be represented by computational graphs with nodes representing tasks to be performed in each iteration and edges representing communication between tasks from one iteration to the next. Parallelizing such applications requires graph partitioning such that the partitions have equal computational load and communication is minimized. Computational graphs derived from many applications have nodes corresponding to points in two or three dimensional space with interactions limited to points that are physically proximate. Examples of such applications include finite element methods and PDE solvers. For such graphs, partitioning is often accomplished by recursive coordinate bisection [5]. Recursive coordinate bisection is another name for the process of splitting data that we used to construct k-d trees.

Let p be the number of processors to which the computational graph is to be distributed. Hence, we are interested in partitioning the graph into p partitions. For this application, we are interested in computing the first $\log p$[1] levels of the k-d tree. The p sets corresponding to the internal nodes at level $\log p$ give the desired graph partition.

Hierarchical Applications In hierarchical applications, we study the evolution of a system of n mutually interacting objects where the strength of interaction between two objects decreases with increasing distance between them. For example, the n-body problem is to study the evolution of a

[1] Throughout this paper log refers to $\log_2$

system of n particles under the influence of mutual gravitational forces. To reduce the $O(n^2)$ work involved in computing all pairwise interactions, a clustering scheme is imposed on the objects and the interaction between two clusters sufficiently far from each other is approximated by treating the clusters as individual objects. For this purpose, a clustering scheme that groups physically proximate objects is required. The k-d tree offers such a clustering scheme.

Once the number of objects in a cluster falls below a constant s, the interactions are computed directly. Hence, the k-d tree is built up to $\lceil \log \frac{n}{s} \rceil$ levels.

Databases A database is a collection of records with each record containing k key fields and other data fields that are never used to access data. We can represent each such record by a point in k-dimensional space by assigning a dimension to each key field and specifying a mapping from key values to coordinates. The records can then be organized using the k-d tree representation of the resulting spatial data [2].

Since databases typically consist of large numbers of records, the records are stored on secondary storage devices. In such a situation, non-homogeneous trees offer the advantage that entire records do not have to be read into main memory to make branching decisions at internal nodes when only one key value is required. Since disk accesses dominate database query times, a node is partitioned only if all the records corresponding to that node do not fit in one disk sector. The number of records, r, that can fit in one disk sector depends on the size of the disk sector and the size of the individual records. The k-d tree is built up to a sufficient number of levels such that each leaf has $\leq r$ records and the parent of any leaf corresponds to a set containing $> r$ records.

In this paper, we study efficient parallel construction of k-d trees up to a specified number of levels on coarse-grained distributed memory parallel computers.

3 Model of Parallel Computation

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousand) connected through an 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 (Cray T3D), hypercubes (nCUBE), fat tree (CM5) and hierarchical networks (cedar, DASH).

CGMs have cut-through routed networks which will be the primary thrust of this paper and will be used for modeling the communication cost of our algorithms. For a lightly loaded network, a message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by $T_{comm} = t_s + t_h d + t_w m$, where t_s represents the handshaking costs, t_h represents

the signal propagation and switching delays and t_w represents the inverse bandwidth of the communication network. The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is determined by the link bandwidth. t_w is often higher (an order to two orders of magnitude is typical) than t_c, the time to do a unit computation on data available in the cache. 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. The above expressions adequately model communication time for lightly loaded networks. However, as the network becomes more congested, the finite network capacity becomes a bottleneck. Multiple messages attempting to traverse a particular link on the network are serialized. A good measure of the capacity of the network is its cross-section bandwidth (also referred to as the bisection width). For p processors, the bisection width is $p/2$, $2\sqrt{p}$, and 1 for a hypercube, wraparound mesh and for a shared bus respectively.

Our analysis will be done for the following interconnection networks: 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).

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [7, 8]. The use of collective communication provide a level of architecture independence in the algorithm design. It also allows for precise analysis of an algorithm by replacing the cost of the primitive for the targeted architecture.

In the following, we describe some important parallel primitives that are repeatedly used in our algorithms and implementations. For commonly used primitives, we simply state the operation involved. The analysis of the running time is omitted and the interested reader is referred to [8]. For other primitives, a more detailed explanation is provided. Table 1 describes the collective communication routines used in the development of our algorithms and their time requirements on cut-through routed hypercubes and meshes. In what follows, p refers to the number of processors.

1. **Broadcast.** In a Broadcast operation, one processor has a message of size m to be broadcast to all other processors.

2. **Combine.** Given a vector of size m on each processor and a binary associative operation, the Combine operation computes a resultant vector of size m and stores it on every processor. The i^{th} element of the resultant vector is the result of combining the i^{th} element of the vectors

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	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p)$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$

Table 1: Running times of various parallel primitives on cut-through routed hypercubes and square meshes with p processors.

stored on all the processors using the binary associative operation.

3. **Parallel Prefix.** Suppose that $x_0, x_1, \ldots, x_{p-1}$ are p data elements with processor P_i containing x_i. Let $\otimes$ be a binary associative operation. The Parallel Prefix operation stores the value of $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i.

4. **Gather.** Given a vector of size m on each processor, the Gather operation collects all the data and stores the resulting vector of size mp in one of the processors.

5. **Global Concatenate.** This is the same as Gather except that the collected data should be stored on all the processors.

6. **All-to-All Communication.** In this operation each processor sends a distinct message of size m to every processor.

7. **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 [10]. 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 $O(\frac{r}{p})$.

8. **Order Maintaining Data Movement.** Consider the following data movement problem, an abstraction of the data movement patterns that we repeatedly encounter in k-d tree construction algorithms: Initially, processor P_i maintains 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 were 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. In the order maintaining data movement problem, this global order should be preserved after the distribution of the data.

The algorithm first performs a Parallel Prefix operation on the s_i's to find the position of the elements each processor contains in the global order. Another parallel prefix operation on the r_i's determines the position in the global order of the elements needed by each processor. Using the results of the parallel prefix operations, each processor can figure out the processors to which it should send data and the amount of data to send to each processor. Similarly, each processor can figure out the amount of data it should receive, if any, from each processor. The communication is performed using the transportation primitive. The maximum number of elements sent out by any processor is s_{max}. The maximum number of elements received by any processor is r_{max}.

9. **Non-Order 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.

   For data movements where preserving the order of data is not important, the following modification is done to the algorithm: Every processor retains $min\{s_i, r_i\}$ of its original 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 the order maintaining data movement algorithm.

   The maximum number of outgoing elements at any processor is $max_{i=0}^{p-1}(s_i - r_i)$, which can be as high as s_{max}. The maximum number of incoming elements at any processor is $max_{i=0}^{p-1}(r_i - s_i)$, which can be as high as r_{max}. Therefore, the worst-case running time of this operation is identical to the order maintaining data movement operation. Nevertheless, the non-order maintaining data movement algorithm is expected to perform better in practice.

4 Parallel construction of k-d trees

We consider the task of building a 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.

It is obvious that the construction of a k-d tree involves building a binary tree where the computational task at each internal node is to find the median of all the points in its subtree. Thus, any median finding algorithm can be used for the the construction of k-d trees. To use this strategy, it only remains to identify efficient sequential and parallel median finding algorithms. 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 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.

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 $\log m$ ($m \leq n$).

In the following subsections, we compare the different strategies for local construction and parallel construction of the tree to $\log p$ levels. These strategies are combined to present methods for constructing the tree for more than $\log p$ levels.

4.1 Local Tree Construction

4.1.1 Median-based Method

In this approach, a partition is represented by an unordered set of points in the partition and the median is explicitly computed in order to split the partition. This is the commonly used method to build k-d trees.

Suppose that it is required to find the median of n elements. A deterministic algorithm for selection is to use the median of medians as the estimated median [4]. This ensures that the estimated median will have at least a guaranteed fraction of the number of elements below it and at least a guaranteed fraction of the elements above it. The worst case number of iterations required by this algorithm is $O(\log n)$.

A randomized algorithm takes $O(n)$ time with high probability by using the algorithm of Floyd et. al. [6]. Define the rank of an element to be the number of elements smaller than or equal to it. It is desired to find the median, i.e. the element with rank $k = \lceil \frac{n}{2} \rceil$. In Floyd's algorithm, a random element is estimated to be the median. The elements are split into two subsets S_1 and S_2 of elements smaller than or equal to and greater than the estimated median, respectively. If $|S_1| >= k$, recursively find the element with rank k in S_1. If not, recursively find the element with rank $(k - |S_1|)$ in S_2. Once the number of elements under consideration falls below a constant, the problem is solved directly by sorting and picking the appropriate element. The randomized algorithm has a worst-case run time of $O(n^2)$, an expected run time of only $O(n)$. The number of iterations can be shown to be $O(\log n)$ with high probability. It is known to perform better in practice than its deterministic counterpart due to the low constant associated with the algorithm.

Note that the very process of computing the median of a partition using Floyd's method 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. Since 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.

The expected number of iterations required for finding the median of n points is $O(\log n)$. A different approach can be used to reduce the number of iterations [9]:

(a) randomly sample $\ell = n^{2/3}$ keys from the input;
(b) sort this sample in $O(\ell \log \ell)$ time;
(c) Pick keys from the sample whose ranks (in the sample) are $\lceil k\ell/n \rceil - \sqrt{\ell \log n}$ and $\lceil k\ell/n \rceil + \sqrt{\ell \log n}$ respectively (Call these keys k_1 and k_2);
(d) Drop all the keys in the input that lie outside the range $[k_1, k_2]$;

With high probability [2] , $k_1 < k < k_2$. If so, drop all the keys in the input with ranks that lie outside the range $[k_1, k_2]$;

With high probability, the number of points reduces from n to no more than $\frac{n}{\sqrt{\ell}}\sqrt{\log n}$. Perform an appropriate selection recursively (or otherwise) in the collection of the remaining keys. It can

[2] We say that a randomized algorithm has a resource bound of $O(g(n))$ with high probability if there exists a constant c such that the amount of resource used by the algorithm for input of size n is no more that $c\alpha g(n)$ with probability $\geq 1 - \frac{1}{n^\alpha}$. As an example, if n = 10^6 and $\alpha = 2$, then the probability that the algorithm takes more than $O(n)$ time is equal to 10^{-12}.

be shown that the expected number of iterations of this median finding algorithm is $O(\log \log n)$ with high probability and that the expected number of points decreases geometrically after each iteration with high probability resulting in $O(n)$ expected running time. If $n^{(j)}$ is the number of points at the start of the j^{th} iteration, only a sample of $o(n^{(j)})$ keys is sorted. Thus, the cost of sorting, $o(n^{(j)} \log n^{(j)})$ is dominated by the $O(n^{(j)})$ work involved in scanning the points.

All the experimental results presented in this paper are limited to the direct approach. However, this algorithm can be substituted without affecting the total time for computation of the median significantly.

4.1.2 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. To begin with, we maintain k sorted arrays $A_1, A_2, \ldots, A_k$ where A_l contains all the points sorted according to dimension d_l. At any stage of the algorithm, we have a partition and k arrays storing the points of the partition sorted according to each dimension. Without loss of generality, assume that the partition should be split along d_1. The median value of the coordinates along d_1 is easily obtained by picking the middle element of A_1. To split the partition, we need to compute the sorted arrays corresponding to the subpartitions. This is extremely simple along d_1 since the array is split into two parts around the middle element. To create the arrays along any other dimension d_l, each element of A_l is scanned and copied to the appropriate array of the subpartition depending upon the coordinate of the point along d_1.

Instead of maintaining k arrays for every partition considered in the k-d tree, we can get away with just maintaining one set of k arrays. Any partition corresponding to a node in the k-d tree 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 sub-partitions. 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 elements when necessary (as can be done in the 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. Note that since the points needs not be distinct, 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 sorting the n points along each of the k dimensions before the actual tree construction. We refer to this as the preprocessing time. The time for preprocessing is $O(kn \log n)$. 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, the data may already be present in sorted order. For instance, if the sort-based method is used to construct the first $\log p$ levels of the tree in parallel, each processor will already have the sorted data for local tree construction.

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 from 1 to 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 most practical applications such as graph partitioning and hierarchical methods.

4.1.3 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 $kn \log n$. Thus, 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 take $O(n)$ 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 can be accomplished using binary search in $O(\log b)$ time. 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 $O(b)$ 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 $O(\log b)$ time. 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 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.

4.1.4 Experimental Results

In this section, we present experimental results for the three algorithms. The computation time required for local tree construction, as summarized in Table 2, can be decomposed into four major parts:

1. Preprocessing time (X) - This is the time required for preprocessing for the different strategies before the actual tree construction. The largest cost is associated with sorting. The cost associated with median finding is zero. Bucketing requires an intermediate cost.

Method	Tree construction upto $\log m$ levels		
	Preprocessing(X)	Median Finding(m)	Data Movement(s)
Median-based	-	$O(n \log m)$	$O(n \log m)$
Sort-based	$O(kn \log n)$	$O(\log m)$	$O(kn \log m)$
Bucket-based	$O(kn \log b)$	$O(\frac{n}{b})$	$O(kn \log m)$

Table 2: Computation time for local tree construction for the different strategies.

2. Cost of finding the median (m) - The cost of finding median at every level for each sublist using the sort-based method is $O(1)$. The cost for bucket-based method is $O(\frac{l}{b} + \log b)$ for a list of size l divided into b buckets. The sum of the sizes of all the b sublists in the bucket-based method is equal to n, the number of elements. Median-based method takes $O(l)$ for a list of size l.

3. Data movement time per element (s) - At each level the lists are decomposed into two sublists based on the median. This requires movement of data. 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. This requires going over the data elements twice: one to count the number of elements for the two sublists and other to actually move the data. Median-based method does not require the first step resulting in a significantly lower value of s. Additionally, the number of lists for which data has to be moved is substantially higher for the sort-based and bucket-based strategy as compared to the median-based strategy ($k - 1$ versus 1).

4. Overhead per list (r) - There is an overhead attached with maintenance and processing of sublists at every level. The number of partitions doubles as the number of levels of the tree increase. The overhead is due to maintaining pointers and appropriate indices for the data in the partition. This overhead is the smallest for sorting, slightly larger for median-method and highest for bucketing. The cost for bucketing is proportional to the number of buckets. This overhead occurs on a per list basis and 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 quick sort has been shown to work well in practice).

Based on the above analysis and the following reasons we decided to limit our experimental results to $k = 2$:

1. In many practical situations the value of k is limited to 2 or 3.

2. For larger k, we do not expect the sort or bucket-based strategy to work better due to large value of s.

3. Our goal is to write optimized software for fixed record sizes and determine if the improvements achieved for $k = 2$ are significant using preprocessing methods.

The following discussion is specific to $k = 2$, although much of the discussion should be applicable for larger values of k also.

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, much larger values of r and m. We experimented with different bucket sizes for the bucketing strategy. There is a tradeoff between r and m. 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 increase exponentially with the increase in the number of levels. However, we found that the values of r and m are sufficiently large that sort-based method was better than the bucket-based method except when the number of levels were very small (less than 4). However, for these cases the median-based approach is expected to work better.

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 m as seen in Table 2 and verified by our experimental results. One would expect that the median-based approach 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 Figure 3. These figures give the time required for sort-based method, median-based method and sort-based method (without the cost of sorting). These results show that median-based approach is better than the sort-based methods except when the number of levels is close to $\log n$. For larger levels the increase due to higher m 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.

The median-based strategy was found to be much better than the bucket-based strategy. Although we do not present any experimental results here, the median-based strategy was much better than the bucket-based strategy for small levels even ignoring the time required for bucketing. The improvement in running time due to a reduced value of m is offset by an increase in the value of s.

We conclude the following from our experimental results for $k = 2$:

1. If information is available about the sorted order along both the dimensions, using a sort-based strategy is always preferable.

2. For unsorted data, using a median-based strategy is the best unless the number of levels are close to $\log n$ 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 or better than the other strategies unless preprocessing information used for sorting method is already available.

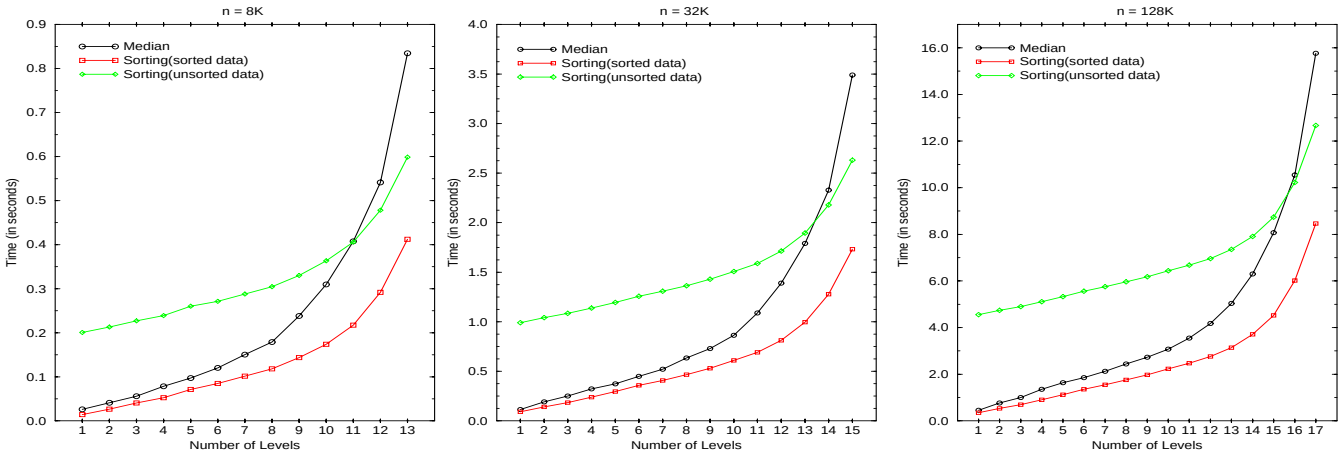

Figure 3: Local tree construction for random data of sizes 8K, 32K and 128K

4.2 Parallel Tree Construction for $\log p$ levels

4.2.1 Median-based Method

Each processor initially has $\frac{N}{p}$ points. The median of the N points along dimension d_1 is found by a parallel median finding algorithm using all the p processors. Each processor locally scans its $\frac{N}{p}$ points and separates them into two sets containing points less than or equal to and greater than the median, respectively. All the points less than or equal to the median are moved to processors $P_0, \ldots, P_{\frac{p}{2}-1}$ and the other points are moved to processor $P_{\frac{p}{2}}, \ldots, P_{p-1}$ such that each processor again has $\frac{N}{p}$ points. We recursively solve the two problems of finding k-d trees of $\frac{N}{2}$ points on $\frac{p}{2}$ processors in parallel. This process is used to build the k-d tree up to $\log p$ levels. The local part of the k-d tree is now built using a sequential median finding algorithm, for which we use a randomized algorithm described in Section 4.1.1. To complete the description of the method, it only remains to describe the parallel median finding algorithms used.

Parallelization of deterministic median finding is not work optimal for arbitrary data distributions. The worst case running time for deterministic median finding can be as large as $O(\frac{N}{p} \log N)$ time without load balancing at every step. This is because the data can be distributed such that, at every iteration the size of the local data on one processor does not decrease. Load balancing is only feasible for a machine with a fast network (small value of t_w) and a cross section bandwidth which is $O(p)$. Otherwise one step of load balancing will dominate the overall cost of median finding. Further, the constants involved with sequential deterministic algorithms are typically an order of magnitude larger. A direct parallel implementation of the simple randomized algorithm without load balancing given in Section 4.1.1 also results in worst case "expected performance" of $O(\frac{N}{p})$ on

p processors due to the same reasons. For this case using a randomized algorithm which chooses a sample of size n^ϵ is preferable, although still suboptimal, as it results in $O(\log\log N)$ iterations resulting in worst case time of $O(\frac{N}{p}\log\log N)$.

A comparison of different deterministic and randomized parallel selection algorithms for coarse grained machines is given in [1]. These show that the randomized approaches have considerably better performance than the deterministic algorithms for arbitrary distributions of data.

For the following algorithms, we assume that $N \geq p^2$ points are randomly distributed among the p processors. If the data is not distributed randomly this can be achieved by using a transportation primitive initially. The cost of this randomization will be considerably less than the cost of construction of the k-d tree, which is lower bounded by the transportation primitive. The reason for this is that in the worst case, construction of the k-d tree may require movement of all the local data assigned to a given processor to all the remaining processors. Thus, the analysis of all the algorithms described below is independent of the initial distribution of data. Using the results given in Appendix A, the following can be shown:

1. Let $n = \frac{N}{p}$. With a high probability the number of points per processor is given by $n + o(n)$. For convenience in the description as well as practical situations we will assume that all these points are equally distributed among all the processors, i.e. each processor has n points.

2. Consider any subset S of these points such that $|S| \geq p\log p$. It can we shown with high probability that the maximum number of points which are mapped to a given processor is given by $O(\frac{|S|}{p})$.

The above property results in parallelization of median finding at every level such that the number of elements mapped to any processor is approximately equal.

We have found that the randomized median finding algorithm, which is a straightforward parallelization of Floyd's algorithm results in the best performance when the data is distributed randomly [1]. The method used is as follows: All processors use the same random number generator with the same seed so that they can produce identical random numbers. Let N be the number of elements and p be the number of processors. Let $N_i^{(j)}$ be the number of elements in processor P_i at the beginning of iteration j. let $N^{(j)} = \sum_{i=0}^{p-1} N_i^{(j)}$. Let $k^{(j)}$ be the rank of the element we need to identify among these $N^{(j)}$ elements.

Consider the behavior of the algorithm in iteration j. First, a parallel prefix operation is performed on the $N_i^{(j)}$'s. All processors generate an identical random number between 1 and $N^{(j)}$ to pick an element at random, which is taken to be the estimate 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 containing elements less than or equal to and greater than the estimated median, respectively. 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 j^{th} iteration. Since the points are mapped randomly among all the processors, it can be shown that the number of remaining points left after each iterations are mapped equally among all the processors with high probability (i.e. the maximum is close to mean) unless the number of remaining points are very small. Thus, the total expected time spent in computation is $O(N/p)$ [3] Another option is to ensure that a load balancing is done after every iteration. However, such a load balancing always resulted in an increase in the running time and that the data is reasonably balanced without any load balancing if the input distribution is random [1].

As discussed in section 4.1.1, the number of iterations required for N points is $O(\log N)$ with high probability. The communication involved in every iteration is one Parallel Prefix, one Broadcast and one Combine operation. Therefore, the expected running time of parallel median finding, is $O(\frac{N}{p} + (t_s + t_w)\log p \log N)$ on the hypercube and $O(\frac{N}{p} + (t_s + t_w)\log p \log N + t_h\sqrt{p}\log N)$ 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. Note that the very process of finding the median splits each local list into two sublists containing elements less than or equal to and greater than the median. 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 moving this data movement results in a random distribution of the two lists to the two subsets of processors.

Since each point has k coordinates, a record of size $O(k)$ is required to store a point. In the median finding algorithm, we repeatedly compare two points based on one coordinate and swap them if necessary. To save work when moving points within an array, we only keep one copy of the records and store pointers in arrays instead of entire records. However, when data movement across processors is required, all the points to be moved have to be copied to an array before communication.

Building the first $\log p$ levels of the tree on the hypercube requires $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s + t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + k\frac{N}{p} + t_s\frac{p}{2^i} + t_w\frac{kN}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(\frac{kN}{p}\log p + t_s(p + \log^2 p \log N) + t_w(k\frac{N}{p}\log p + \log^2 p \log N) + t_h p \log p)$ time. The time required on the mesh is $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s+t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + t_h\sqrt{\frac{p}{2^i}}\log\frac{N}{2^i} + k\frac{N}{p} + (t_s + t_w\frac{kN}{p})\sqrt{\frac{p}{2^i}} + t_h\sqrt{\frac{p}{2^i}}) = O(\frac{kN}{p}\log p + t_s(\sqrt{p} + \log^2 p \log N) + t_w(\frac{kN}{\sqrt{p}} + \log^2 p \log N) + t_h\sqrt{p}\log N)$.

[3] This can be converted into an algorithm which completes with high probability using the median finding approach described in Section 4.1.1

4.2.2 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 [12]: Suppose we want to sort the points along d_1. First, sort the $\frac{N}{p}$ points on each processor locally. This takes $O(\frac{N}{p}\log\frac{N}{p})$ time. Select p evenly spaced points from the local sorted array of each processor. Gather the selected $p(p-1)$ elements on one processor and sort them locally in $O(p^2\log p)$ time. Using $p-1$ evenly spaced elements in this array, the range of the input points is partitioned into p ranges called buckets. It is guaranteed that no more than $\frac{2N}{p}$ points belong to any bucket [12]. The buckets are specified by the endpoints of the corresponding ranges, also called splitters. We need $p-1$ splitters to specify the p buckets. The $p-1$ splitters are broadcast to every processor. Each processor locates the $p-1$ splitters in its sorted array of $\frac{N}{p}$ points using binary search in $O(p\log\frac{N}{p})$ time. This splits the local array on each processor into p sorted subarrays, one belonging to each bucket. The subarrays are distributed to the processors such that P_i gets all subarrays that belong to bucket i. This is done using the transportation primitive since there can be high variance in the individual message sizes. Each processor then merges the p subarrays it receives. This takes $O(\frac{N}{p}\log p)$ time. Finally, an Order-maintaining data movement operation is used to ensure that each processor has exactly $\frac{N}{p}$ points.

The total computation time required is $O(\frac{N}{p}\log\frac{N}{p} + p^2\log p + p\log\frac{N}{p} + \frac{N}{p}\log p)$. For $N \geq p^2$, this reduces to $O(\frac{N}{p}\log N + p^2\log p)$. The communication time depends on the topology and can be easily found by substituting the running times of the communication primitives used. The total time required for sorting is $O(\frac{N}{p}\log N + p^2\log p + t_s p + t_w(\frac{N}{p} + p^2) + t_h p\log p)$ on a hypercube and $O(\frac{N}{p}\log N + p^2\log p + t_s\sqrt{p} + t_w(\frac{N}{\sqrt{p}} + p^2) + t_h\sqrt{p})$ on a mesh.

Once we preprocess the data by sorting it along each dimension to create k sorted arrays, the work of finding 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 $\sum_{i=0}^{\log p-1} O(k^2\frac{N}{p} + t_s\frac{p}{2^i} + t_w k^2\frac{N}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(k^2\frac{N}{p}\log p + t_s p + t_w k^2\frac{N}{p}\log p + t_h p\log p)$. The corresponding time on a mesh is $\sum_{i=0}^{\log p-1} O(k^2\frac{N}{p} + t_s\sqrt{\frac{p}{2^i}} + t_w k^2\frac{N}{p}\sqrt{\frac{p}{2^i}} + t_h\sqrt{\frac{p}{2^i}}) = O(k^2\frac{N}{p}\log p + t_s\sqrt{p} + t_w k^2\frac{N}{\sqrt{p}} + t_h\sqrt{p})$.

Unlike the sequential sort-based method, the above scheme cannot be easily modified to reduce the computational work at every level from $O(kn)$ to $O(n)$. This is because the method to reduce computational work used elements of an array L to access elements of an array A_l and vice versa. Since these arrays are distributed across processors, the resulting communication generated makes this method impracticable.

4.2.3 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 $O(\frac{N^\epsilon \log N^\epsilon}{p} + \frac{N^\epsilon}{p}\log^2 p + (t_s + t_w\frac{N^\epsilon}{p})\log^2 p)$ time on a hypercube and $O(\frac{N^\epsilon \log N^\epsilon}{p} + \frac{N^\epsilon}{p}\log^2 p + (t_s + t_w\frac{N^\epsilon}{p})\sqrt{p})$ 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

Method	Parallel tree construction upto $\log p$ levels			
	Preprocessing (X)	Median finding (m)	Local processing (s)	Communication due to data movement(T)
Median-based	-	$O(\frac{N}{p}\log p + (t_s + t_w)\log^2 p \log N)$	$O(k\frac{N}{p}\log p)$	$O(t_s p + t_w(k\frac{N}{p}\log p))$
Sort-based	$O(\frac{N}{p}\log N + p^2 \log p + t_s p + t_w(\frac{N}{p} + p^2))$	$O(\log p)$	$O(k^2\frac{N}{p}\log p)$	$O(t_s p + t_w(k^2\frac{N}{p}\log p))$
Bucket-based	$O(\frac{N}{p}(k + \log p) + t_s p + t_w(k\frac{N}{p}))$	$O(\frac{N}{p})$	$O(k^2\frac{N}{p}\log p)$	$O(t_s p + t_w(k^2\frac{N}{p}\log p))$

Table 3: Time for tree construction upto $\log p$ levels on p processors for different strategies on a hypercube.

for splitting the buckets along each dimension. The first $\log p$ levels of the tree can be built in $\sum_{i=0}^{\log p-1} O(k^2\frac{N}{p} + t_s\frac{p}{2^i} + t_w k^2\frac{N}{p} + t_h\frac{p}{2^i}\log\frac{p}{2^i}) = O(k^2\frac{N}{p}\log p + t_s p + t_w k^2\frac{N}{p}\log p + t_h p\log p)$ time on a hypercube and $\sum_{i=0}^{\log p-1} O(k^2\frac{N}{p} + t_s\sqrt{\frac{p}{2^i}} + t_w k^2\frac{N}{\sqrt{p}} + t_h\sqrt{\frac{p}{2^i}}) = O(k^2\frac{N}{p}\log p + t_s\sqrt{p} + t_w k^2\frac{N}{\sqrt{p}}\log p + t_h\sqrt{p})$ time on a mesh.

4.2.4 Experimental Results

In this section, we compare the three algorithms for tree construction for $\log p$ levels experimentally. Our implementations are on the CM-5 for which most of the analysis presented for the hypercube is applicable. The relative communication overheads for a mesh also compare similarly as presented in this section. Our goal is to compare the software overheads in implementing these strategies in a relatively machine independent manner. For all the experiments we assume that the distribution of data is random. Whenever, this not the case a preprocessing step can be added to perform this randomization. As discussed earlier, the cost of this randomization is small enough that it should not affect the overall timings and conclusions significantly.

The execution time required for parallel tree construction for $\log p$ levels can be decomposed into the following parts as summarized in Table 3:

1. Preprocessing time (X) - This is the time required for preprocessing for the different strategies before the actual tree construction. The largest cost is associated with the sort-based method as k lists, each containing N elements needs to be sorted across p processors. The cost associated with median finding is zero. Bucketing requires an intermediate cost since the data is partially ordered. This includes communication costs involved in the primitive operations used here.

2. Cost of finding the median at every level (m) - The cost of finding median at every level for each sublist using the sort-based is $O(1)$ and requires little communication and local pro-

cessing. The bucket-based approach requires slightly larger communication and higher local processing costs (proportional to size of the bucket which contains the median). The median-based approach requires the maximum amount of communication (number of broadcasts and combines required is $O(\log \frac{N}{2^i})$ for level i). However this cost is independent of the number of local data items ($O(N/p)$). The local preprocessing costs are proportional to $O(N/p)$.

3. Local processing time (s) - At each level the lists are decomposed into two sublists based on the median. This requires local processing in identifying elements for the appropriate partition. 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. This requires going over the data elements twice: one to count the number of elements for the two sublists and other to actually move the data. Median-based method does not require the first step resulting in a significantly lower value of s. Additionally, the number of lists for which data has to be moved is substantially higher for the sort-based and bucket-based strategy as compared to the median-based strategy ($k-1$ versus 1).

4. Communication due to data movement (T) - All of these methods require the application of transportation primitive. This cost is expected to be higher for the bucket and sort-based methods as compared to the median-based approach which require the ordering in the data to be maintained when data is moved. Further, the number of lists for which data has to be communicated is substantially larger for the sorting and bucket-based strategy as compared to the median-based strategy ($k-1$ versus 1).

5. Overhead per list (r) - There is an overhead attached with maintenance and processing of sublists at every level. The number of partitions doubles as the number of levels of the tree increase. The overhead is due to maintaining pointers and appropriate indices for the data in the partition. This overhead is the smallest for sorting, slightly larger for median-method and highest for bucketing. The cost for bucketing is proportional to the number of buckets. This overhead occurs on a per list basis and grows exponentially with the increase in number of levels.

The t_h term is insignificant in the overall communication time and hence it is ignored in the table. For reasons similar to one given for local tree construction, we limit our experimental results to only $k = 2$.

A comparison of sorting and median finding approaches show that sorting has much higher value of X, larger value of s, a smaller values of r and a higher value of T. We divide the comparison into two parts:

1. Small values of N/p: For this case median finding should not parallelize well. The time is dominated by the communication cost in m, which increases with increase in number

of processors. One would expect the median-based approach to still work better for small number of processors, due to the large value of X in sorting. Otherwise sorting may be preferable since the higher communication costs in median finding will offset the effect of X in the sort-based approach.

2. Large values of N/p: For large values of N/p, median finding should parallelize reasonably well. The communication cost in median finding should not dominate the overall cost and should be significantly lower than the cost of T required for both strategies. The value of T should be smaller for the median-based approach as compared to the sort-based approach because it does not require maintaining the order in the data movement. Thus, median-based approach should have lower overall communication costs when compared to the sort-based approach at every level. One would expect median-based approach to perform better than sorting except for very large number of processors.

A comparison of bucket-based and median finding approaches show that bucket-based approach has larger value of X, larger value of s, a larger value of r and T. We will again divide the comparison into two parts:

1. Small values of N/p: For this case, median finding does not parallelize well. The time is dominated by the communication cost in m which increases with increase in number of processors. One would expect that median-based approach to perform worse than the bucket-based approach which has small communication overhead for median finding unless the number of processors are small and the value of X is large for bucket-based approach.

2. Large values of N/p: For large values of N/p median finding parallelizes reasonably well. The communication cost in median finding should not dominate the overall cost and should be significantly lower than the cost of T required for both strategies at every level. The value of T should be smaller for the median-based approach as compared to the bucket-based approach because it does not required maintaining the order in the data movement. Thus, median-based approach should have lower overall communication costs than bucket-based approach at every level. One would expect median-based approach to be better than bucket-based approach except for a very large number of processors.

A comparison of bucket-based and sort-based approaches show major differences in the preprocessing time (X) and the local processing time in finding the median. One would expect that bucketing would work better than sorting when the difference in preprocessing time time is larger than the added time in local preprocessing for finding the median. The cost of the latter decreases at every level (as the sizes of each of the buckets at an average decreases with increase in number of levels).

The experimental results for different values of N/p for constructing a tree up to $logp$ levels, are

presented in Figure 4. These show that the sort-based strategy is never better than the bucket-based method for any value of N/p . The median-based approach is the best approach for large values of N/p (greater than 8K per processor) while the bucket-based approach is the best for small values of 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. The crossover point (of k) for which the median-based strategy would be better for small values of N/p would be machine specific.

We would like to emphasize that the conclusions reached above are for a randomized median-based strategy. The constants involved for a deterministic algorithm for median finding may make sort-based and bucket-based methods better than the median based algorithms for small values of k and for a wide range of N/p.

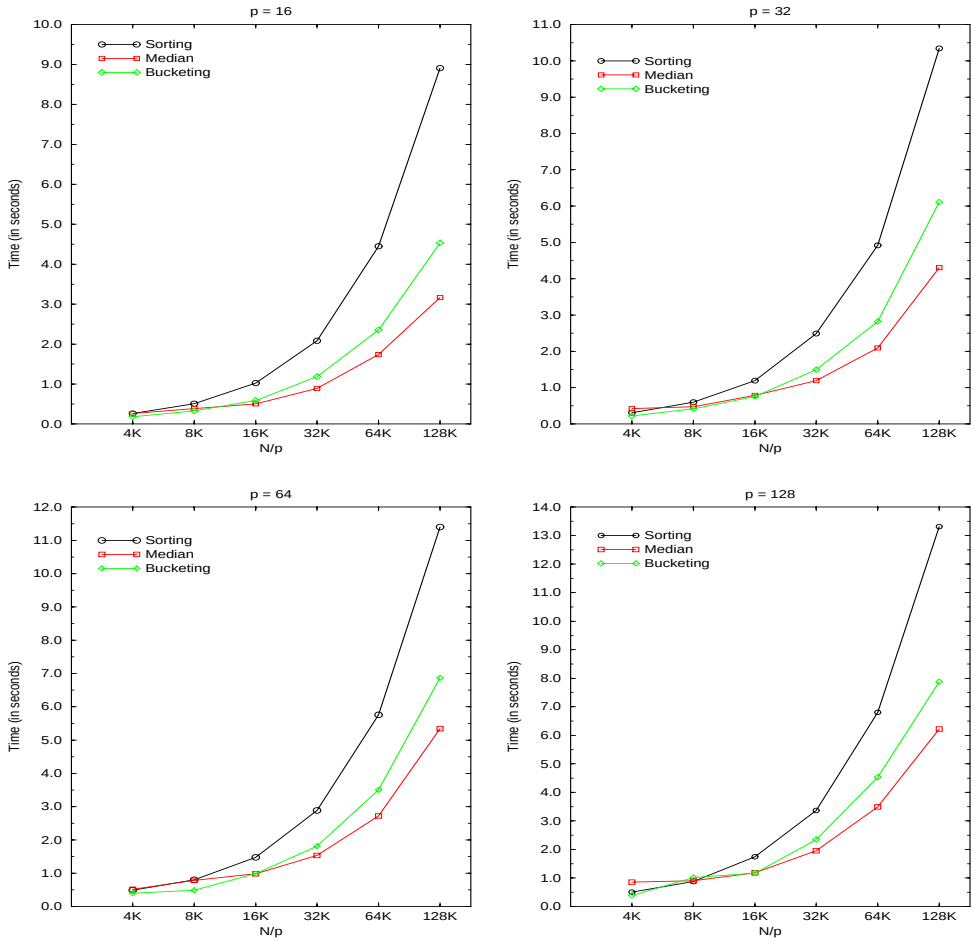

Figure 4: Tree construction to log p levels for p = 16, 32, 64 and 128 using random data

4.3 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. Potentially a different strategy 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 different number of levels ($\log p$ to $\log N$) for different number of processors (8, 32, 128) for different values of N/p (4K, 16K, and 128K) (see Figure 5). 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.

For larger values of k, we would expect that one of the median or bucket-based strategies should be used to construct the tree till $\log p$ levels. This would depend on the value of N/p and the target architecture. 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.

5 Reducing the Data Movement

The algorithms described for the parallel construction of the first $\log p$ levels of the tree require massive data movement using transportation primitive at every level of the tree. The large number of points per processor available allows for reducing the cost significantly by using a different approach.

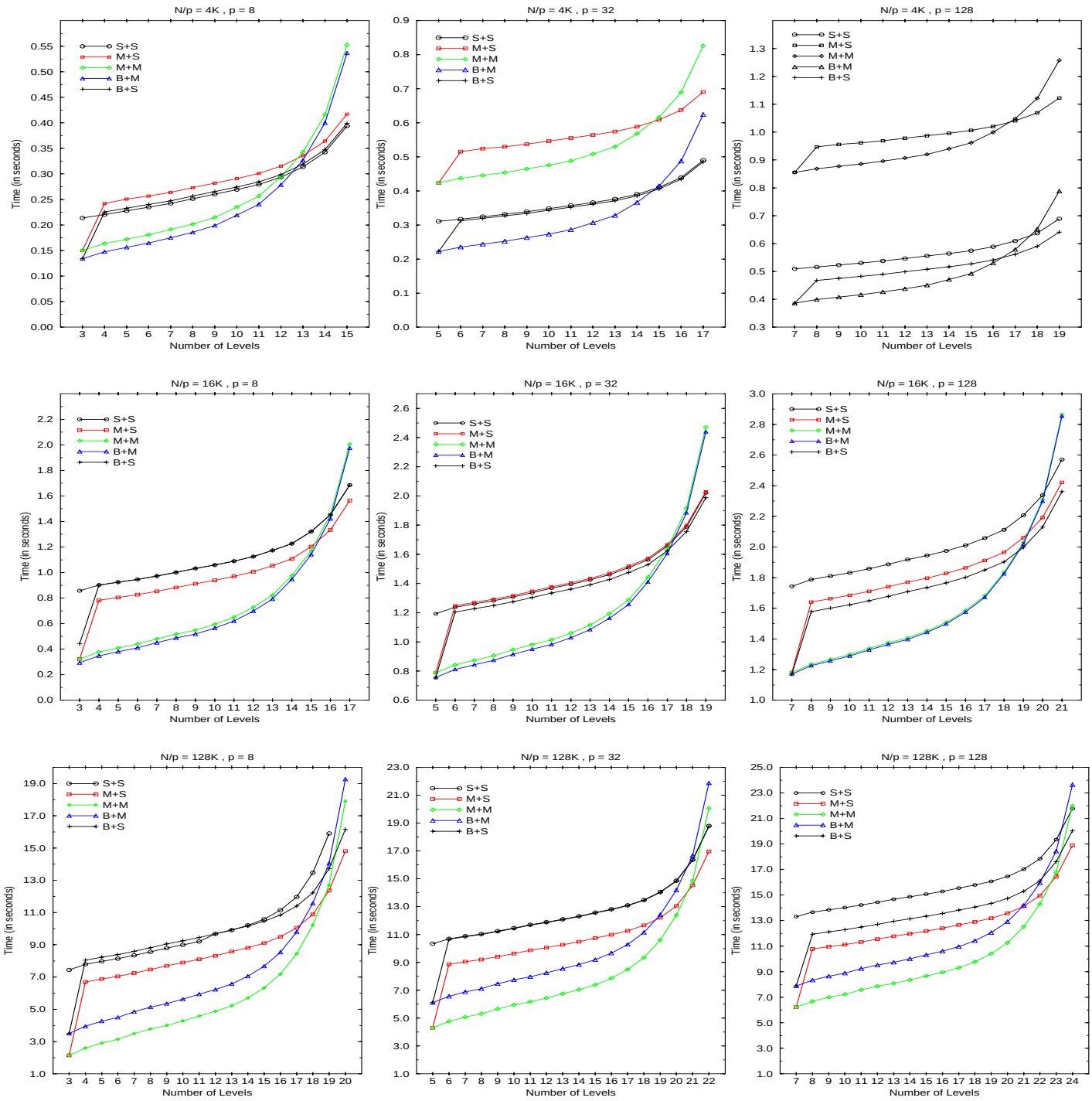

Figure 5: Global tree construction for random data of size $\frac{N}{p}$ = 4K, 16K, 128K on p= 8, 32, 128. On the X-axis level i represents the tree is built to i levels (a tree having 2^i leaves)

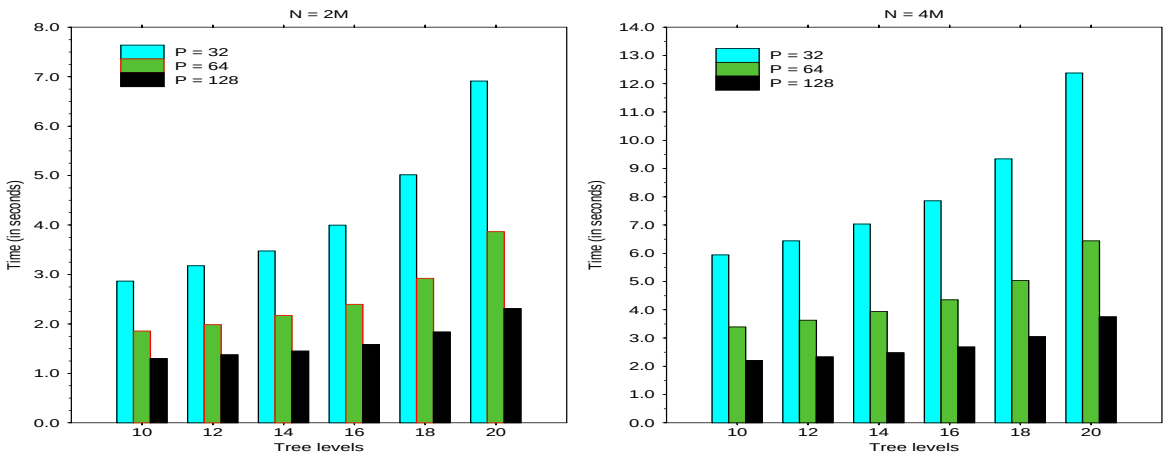

Figure 6: Total time for tree construction on 32, 64 and 128 processors up to the specified level on 2M and 4M random data

Consider the median-based method for constructing the first $\log p$ levels of the tree. Initially, all the N points belong to one partition and are distributed uniformly on all the p processors. After finding the median, the local data is divided into two sublists (typically of unequal size), each belonging to one of the subpartitions. Rather than 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. Each processor potentially has different number of points from each of the two subpartitions. A median can be found for each of the subpartitions in parallel by combining the communication (median calculation using randomized methods involves broadcasts and prefix calculation) and computation for both the lists. The local computation for each subpartition is proportional to the number of points assigned to a given processor. Since the sum of the points from each of the subpartitions assigned to a given processor is equal to $\frac{N}{p}$, the local work involved would be balanced among all the processors. This approach can be repeatedly applied until the number of subpartitions is equal to p. At this stage, the data can be distributed among the processors by using a transportation primitive such that each processor has all the points of one of the p subpartitions. The local tree can then be constructed on each processor without any communication. Although the communication required for finding the median for each of the last $\log p - 2$ stages will be larger than if the subpartitions were decomposed among subsets of processors, the overall time for communication would be less because data movement costs would be significantly reduced.

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. Note that a processor may contain estimated medians of any number of partitions between 0 and 2^i. We need to broadcast the estimated medians to all the processor. To avoid 2^i broadcasts, we adopt the following strategy: Each processor has

an array of size 2^i corresponding to the 2^i subpartitions, with all elements initialized to zero. If a processor has the estimated median for a subpartition, it fills the corresponding element of the array with the estimated median. 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 $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N)$ on a mesh. Constructing the first $\log p$ levels of the tree on a hypercube requires $\sum_{i=0}^{\log p - 1} O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ 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 $O(\frac{N}{p}(\log p + k) + t_s(p + \log^2 p \log N) + t_w(\frac{kN}{p} + p \log p \log N) + t_h p \log p)$. The corresponding time on the mesh is $\sum_{i=0}^{\log p - 1} O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N + t_h \sqrt{p} \log N) + O(\frac{kN}{p} + t_s \sqrt{p} + t_w \frac{kN}{\sqrt{p}}) = O(\frac{N}{p}(\log p + k) + t_s(\sqrt{p} + \log^2 p \log N) + t_w(\frac{kN}{\sqrt{p}} + p \log p \log N) + t_h \sqrt{p} \log p \log N)$.

Compare the equations describing the run-time of the median-based method without and with reduced data movement. For both hypercubes and meshes, we note that 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, the amount of data transferred reduces by a factor of $\log p$ from $\frac{kN}{p} \log p$ to $\frac{kN}{p}$. The same analysis holds true for permutation networks. However, on the mesh, the data transferred improves only by a small constant factor.

A similar strategy can be used to reduce the data movement for sorting as well as bucketing method.

5.1 Experimental Results

We limited ourselves to applying this strategy to the median-based approach only. Figure 7 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 strategy gives significant improvements due to lower data movement. 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 insignificant for higher dimensional data. Thus, the new strategy should give improved performance as the number of dimensions is increased.

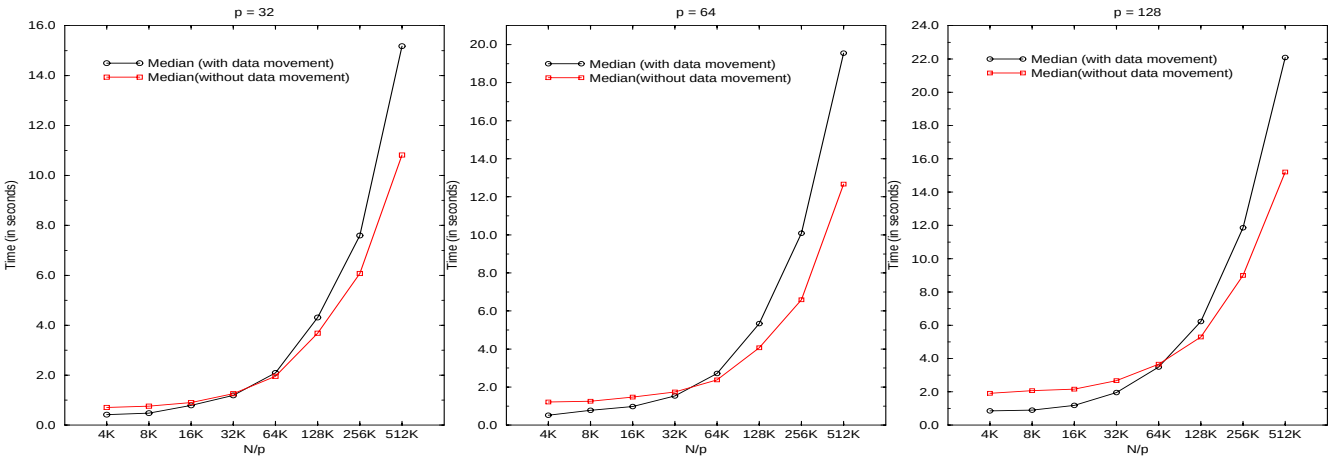

Figure 7: Comparison of the two approaches for $\log p$ levels for different values of N/p.

6 Conclusions

In this paper, we have looked at various strategies for the parallel construction of multidimensional binary search trees. We have designed algorithms that use preprocessing to eliminate or ease the task of median finding at every internal node of the tree to investigate if such strategies lead to faster algorithms than traditional median finding approaches.

For two dimensional point sets, the median-based strategy is faster than the other strategies for large values of $\frac{N}{p}$. It also exhibits good scaling as can be seen by the run-time analysis and the experimental results. This is true mainly because of the performance of the randomized median finding on random data sets. For arbitrary data sets, a randomization step can be performed without much additional cost for the task of constructing a k-d tree. The methods using preprocessing do not perform well for arbitrary levels mainly because the cost of preprocessing is high and can be effectively amortized only if the tree is built almost completely. The sort-based method works better than the median-based method if the tree is built with a single element per leaf. Note that the conclusions we derive are based on the use of randomized algorithms for median finding. In our investigation of various algorithms for median finding [1], we found that deterministic algorithms are slower by an order of magnitude. Using deterministic median finding algorithms would lead to entirely different conclusions. Bucket-based strategy proposed in the paper is found to be useful for applications such as graph partitioning when the number of points per processor is small.

Even though our experiments were conducted for two dimensional point sets, they have certain implications for higher dimensional point sets. 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. Theoretical as well as experimental analysis both suggest that using data parallelism is preferable upto $\log p$ levels followed by running the tasks sequentially in each of the processors.

7 Acknowledgements

Ibraheem Al-furaih was supported by a scholarship from King AbdulAziz City for Science and Technology (KACST), Riyadh, Saudi Arabia. Sanjay Goil is supported in part by NASA under subcontract #1057L0013-94 issued by the LANL. Sanjay Ranka is supported in part by NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

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, Technical Report No. SCCS-743, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY.

[2] J.L. Bentley, Multidimensional binary search trees in database applications, *IEEE Transactions on Software Engineering, SE-5* (1979) 333-340.

[3] J.L. Bentley, Multidimensional binary search trees used for associative search, *Comm. of ACM, 19* (1975) 509-517.

[4] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest and R.E. Tarjan, Time bounds for selection, *J. Comp. Sys. Sci., 7* (1972) 448-461.

[5] F. Ercal, *Heuristic approaches to task allocation for parallel computing*, Ph.D. Thesis, Ohio State University, 1988.

[6] R.W. Floyd and R.L. Rivest, Expected time bounds for selection, *Comm. of ACM, 18* (1975) 165-172.

[7] G. Fox, et. al. *Solving Problems on Concurrent Processors: Volume I - General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ. (1988).

[8] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings Publishing Company, California, 1994.

[9] S. Rajasekharan, W. Chen and S. Yooseph, Unifying themes for parallel selection, *Proc. 5^{th} International Symposium on Algorithms and Computation*, Beijing, China, 1994, Springer-Verlag Lecture Notes in CS 834, 92-100.

[10] S. Ranka, R.V. Shankar and K.A. Alsabti, Many-to-many communication with bounded traffic, *Proc. Frontiers of Massively Parallel Computation* (1995), to appear.

[11] A. Schonhage, M.S. Paterson and N. Pippenger, Finding the median, *J. Comp. Sys. Sci., 13* (1976) 184-199.

[12] H. Shi and J. Schaeffer, Parallel Sorting by Regular Sampling, *J. Parallel and Distributed Computing, 14* (1992), 361-370.

Appendix A

A Bernoulli trial has two outcomes namely *success* and *failure*, the probability of success being p. A binomial distribution with parameters n and p, denoted as $B(n,p)$, is the number of successes in n independent Bernoulli trials.

Let X be a binomial random variable whose distribution is $B(n,p)$. If m is any integer $> np$, then the following are true:

$$Prob.[X > m] \leq \left(\frac{np}{m}\right)^m e^{m-np};$$

$$Prob.[X > (1+\delta)np] \leq e^{-\delta^2 np/3}; \text{ and}$$

$$Prob.[X < (1-\delta)np] \leq e^{-\delta^2 np/2}$$

for any $0 < \delta < 1$.

We say that a randomized algorithm has a resource bound of $O(g(n))$ with high probability if there exists a constant c such that the amount of resource used by the algorithm for input of size n is no more that $c\alpha g(n)$ with probability $\geq 1 - \frac{1}{n^\alpha}$.