Parallel Algorithms for High-dimensional Proximity Joins

John C. Shafer* Rakesh Agrawal

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We consider the problem of parallelizing high-dimensional proximity joins. We present a parallel multidimensional join algorithm based on an the epsilon-kdB tree and compare it with the more common approach of space partitioning. An evaluation of the algorithms on an IBM SP2 shared-nothing multiprocessor is presented using both synthetic and real-life datasets. We also examine the effectiveness of the algorithms in the context of a specific data-mining problem, that of finding similar time-series. The empirical results show that our algorithm exhibits good performance and scalability, as well an ability to handle data-skew.

1 Introduction

Many emerging applications require efficient processing of proximity joins on high-dimensional points[20]. Typical queries in these applications include:

- Find all pairs of similar images (often as a prelude to clustering the images).
- Retrieve music scores similar to a target music score.
- Discover all stocks with similar price movements.

The work presented in this paper was motivated by the particular data-mining problem of finding similar time-series[1][7]. In [2], an algorithm was proposed that first finds all similar "atomic" subsequences, and then stitches together the atomic subsequence matches to obtain larger similar subsequences. A sliding window of size w is used to create atomic subsequences from each time series. These atomic subsequences are then mapped to points in w-dimensional space. Finding similar atomic subsequences now corresponds to the problem of finding all pairs of w-dimensional points that lie within ϵ-distance of each other, where ϵ is a user-specified parameter.

While parallel algorithms for performing joins on spatial data already exists (e.g. [18], [3], [9]), they have mainly concentrated on joining map data where spaces are typically limited to only two or three dimensions. Furthermore, these algorithms have been designed primarily to perform intersection joins on geometric objects such as polygons and line segments and are not well optimized for handling high-dimensional point data.

We examine the issues in performing proximity joins on high-dimensional points in parallel. We will focus on self joins, although the ideas presented directly apply to non-self joins.

Problem Definition

The proximity self-join problem is defined as follows:

- *Self-join:* Given a set of N w-dimensional points and a distance metric, find all pairs of points that are within ϵ distance of each other.

The distance metric for two w-dimensional points $\vec{X}$ and $\vec{Y}$ that we consider is

$$L_p = \left(\sum_{1}^{w} (X_i - Y_i)^p \right)^{1/p}, \quad 1 \leq p \leq \infty$$

L_2 is the familiar Euclidean distance, L_1 the Manhattan distance, and L_∞ corresponds to the maximum

*Also, Department of Computer Science, University of Wisconsin, Madison.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

distance in any dimension.

Paper organization

The rest of this paper is organized as follows: Section 2 discusses some of the existing serial work on multidimensional joins. Section 3 presents related parallel work as well as our parallel ϵ-kdB join algorithm and a space-partitioning algorithm. A performance study comparing join algorithms along with a sensitivity analysis is given in Section 4. Concluding remarks and possible future directions are in Section 5.

2 Serial Join Algorithms

In this section, we briefly review some of the existing multidimensional join work and discuss how they apply to our problem of performing a proximity join with high-dimensional points. Many methods have been proposed including a variety of index-based methods, as well as space partitioning and multidimensional remapping approaches. However, most of these algorithms are designed for geometric rather than point data with universes typically consisting of only two or three dimensions.

2.1 Non-Index Based

One approach to multidimensional proximity joins is to use space-filling curves to map objects into one-dimensional values. This is done by partitioning the space regularly into N cells. A space-filling curve is drawn through the multidimensional space and cells are numbered in the order visited. Objects to be joined are then examined sequentially and for each cell which an object overlaps, a <cell-number, object-pointer> pair is created. Standard relational indices and techniques for computing joins can now be used on the tuples' one-dimensional cell values. This approach in [17] uses a space-filling curve known as the "Z curve"; assigned cell-numbers are called "Z values". Other space-filling curves include the *Gray code*[6] and the *Hilbert curve*[15]. Of these three, the Hilbert curve has been shown to cluster space better[11]. A shortcoming of space-filling curves is that some proximity information is always lost, so nearby objects may have very different Z values. This complicates the join algorithm. This approach works best when the join condition is that two objects overlap. This reduces the problem to an equi-join on the Z values and allows the traditional relational join algorithms to be used essentially unmodified.

A related approach is to do away with the space-filling curves and one-dimensional mapping and represent each cell with a data bucket. Instead of mapping objects to cell numbers, we store a pointer to the object in each data bucket which is within ϵ-distance. We can then perform a self join by joining each individual data bucket with itself. Non-self joins are handled by using the same partitioning scheme on each dataset and then joining corresponding data buckets. This approach falls within the general framework presented in [14]. In that framework, an algorithm defines *bucket extents* to hold data objects and an *assignment function* that maps data objects to buckets. Bucket extents may or may not be immutable and the assignment function may be one-to-one or many-to one. In the join phase, an algorithm identifies pairs of buckets to be joined (termed *join-bucket pairs*) and effects each join in turn. An example of this framework was presented in [14] where *bootstrap seeding*[13] and sampling was used to obtain the initial bucket extents.

Another space partitioning algorithm ($PBSM$) was recently presented in [18] and to some degree, it also fits within the above framework. To address data skew, $PBSM$ partitions the space into many *tiles* such that there are more tiles than data buckets. These tiles are then grouped together using hashing to produce buckets that are relatively consistent in size. Objects are mapped to the buckets depending on which tiles they overlap. For non-self join, objects of the larger relation are mapped to only one bucket — objects of the smaller relation must be replicated in each bucket whose extent is overlapped. Joins between corresponding buckets are then performed using plane sweeping. To perform the join efficiently, the number of buckets is chosen such that two buckets will fit entirely in memory.

2.2 Index Based

Considerable recent work in multidimensional joins has focused on using indices to aid the join. This includes R-trees as used in [4], [3] and [2], PMR quadtrees in [9], and seeded trees in [13]. Whatever the index used, they follow the same schema whereby two sets of multidimensional objects are joined by doing a synchronized depth-first traversal of their indices. Intersection joins are handled by joining any two index buckets whose extents overlap. Likewise, proximity joins are handled by joining any two index buckets whose boundaries are sufficiently near.

Most of these approaches are not well suited to the particular problem of proximity joins on high-dimensional points. The inadequacies include an inability to scale to high-dimensions [20]. For example, the R tree and the kdB tree both use a "minimum bounding rectangle" (MBR) to represent the regions covered by each node in the index. As the number of dimensions gets large, the storage and traversal costs associated with using MBRs increases. Most indices also have substantial build-times. If an index required

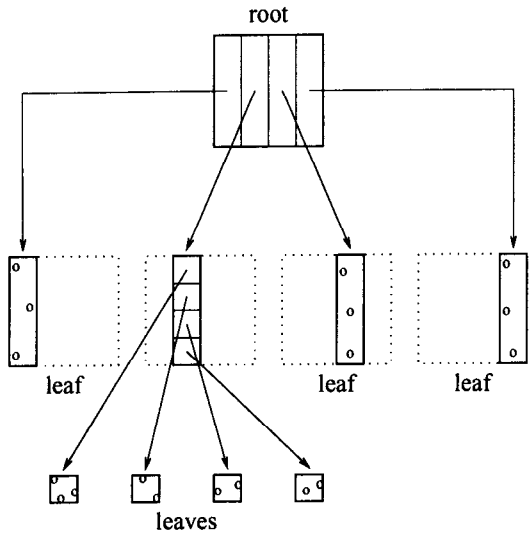

Figure 1: ϵ-kdB Tree

```
procedure self-join(x)
begin
  if leaf-node(x) then
    leaf-self-join(x);
  else begin
    for i = 1 to f−1 do
    begin
      self-join(x[i], x[i]);
      join(x[i], x[i+1]);
    end
    self-join(x[f], x[f]);
  end
end
```

```
procedure join(x, y)
begin
  if leaf(x) and leaf(y) then
    leaf-join(x, y);
  else if leaf(x) then begin
    for i = 1 to f do
      join(x, y[i]);
  end
  else if leaf(y) then begin
    for i = 1 to f do
      join(x[i], y);
  end
  else begin
    for i = 1 to f − 1 do
    begin
      join(x[i], y[i]);
      join(x[i], y[i+1]);
      join(x[i+1], y[i]);
    end
    join(x[f], y[f]);
  end
end
```

Figure 2: ϵ-kdB join algorithm

for a join does not exist, the cost required to build it can often be more than the cost of the join [18]. Many of these algorithms therefore assume that the required index is already available. This is not well-suited to our motivating application of similar time-series since the data to be joined is typically generated "on-the-fly".

Other drawbacks include skew-handling capabilities. In the Grid File[16], skewed data can cause rapid growth in the size of the directory structures. For other indices such as the R tree, skew-handling typically requires maintaining height-balanced trees so that range queries can be efficiently processed. Height-balancing with required updates and possible reinsertions is a major reason for the high cost required to build these indices. A data structure called the ϵ-kdB tree was recently presented in [20] to address the above concerns.

2.3 The ϵ-kdB Tree

The ϵ-kdB tree is an attractive base for our parallel algorithm due to it being specifically designed for performing proximity joins on high-dimensional points. Since the ϵ-kdB tree is not a general-purpose database index, it does not have the many overheads associated with supporting general operations like updates and range queries. Rather than building a persistent structure, the proximity join using the ϵ-kdB tree builds a main-memory data structure on-the-fly for the sole purpose of executing a specific proximity join. This approach works only because building the ϵ-kdB tree dynamically is very fast. Datasets that are too large to fit in memory are handled by partitioning the data and performing an in-memory ϵ-kdB join on each individual partition in turn [20].

Data Structure

In the ϵ-kdB tree, when a leaf node becomes "full" (a parameter that is usually a function of the fanout f), the leaf is split and its data redistributed among its new children. Splitting is performed on a single dimension to create partitions that are either ϵ or slightly larger than ϵ in width. Epsilon, remember, is the user-specified proximity distance. An example ϵ-kdB tree for two dimensions is shown in Figure 1.

Note that for any node x in the ϵ-kdB tree, the data-points belonging to x will only join with the points belonging to the two adjacent siblings of node x. For example, if x is the ith child of node y (denoted $y[i]$), then x can only join with its two siblings $y[i-1]$ and $y[i+1]$. Thus, determining which leaves can be joined is quite simple and depends only on the path traversed from the root to each leaf. Dimensions are only split once since further partitioning would not separate non-joining data-points any further. The ϵ-kdB tree also uses a global ordering of the dimensions for splitting nodes; all nodes in a given level of the tree use the same dimension for splitting. Global split ordering minimizes the number of neighboring leaf nodes and therefore minimizes the number of leaves that must later be joined[20]. Ideally, the dimension ordering should be chosen to minimize correlations between dimensions, but a random ordering is reasonably effective.

Join Algorithm

To perform a self join of an ϵ-kdB tree, we begin at the root and recursively call the self-join algorithm on each child. We also recursively join each child with its right-adjacent sibling. Self joins of leaves and joins between two leaves are performed by using sort-merge

join. Since it is unlikely that all dimensions will be used for splitting, a non-split dimension is used to sort the data-points in the leaves to be joined. The self-join(x) and join(x,y) algorithms are given as pseudo-code in Figure 2.

To perform a non-self proximity join between two different datasets, we build an ϵ-kdB tree for each dataset using the same global split ordering. This will result in nearly identical tree structures and makes non-self joins no more difficult to execute than self joins. Note that each ϵ-kdB tree is tailored to a specific value of ϵ. However, since the ϵ-kdB tree is cheap to build, generating a new one for each desired ϵ-join is still a win.

3 Parallel Multidimensional Proximity Join

We now examine the problem of performing high-dimensional proximity joins in parallel. As with most parallel algorithms, we must address issues such as workload balance, communication costs and data replication. We assume a shared-nothing parallel environment where each of N processors has private memory and disks. The processors are connected by a communication network and can communicate only by passing messages. Examples of such parallel machines include GAMMA[5] and IBM's SP2[10]. We assume that the data to be joined is distributed equally over the local disks of the multiprocessor.

3.1 Previous Work

Virtually all of the existing work on parallelizing multidimensional joins has focused on joining two-dimensional geometric objects. For example, the authors in [3] use R-trees to join spatial objects in a hybrid shared-nothing/shared-memory architecture where a single data processor services all I/O requests. The authors in [9] compare data-parallel PMR quadtrees with data-parallel R- and R^+-trees for joins and range queries on two-dimensional line segments. Several unusual architectural models were explored in that work in addition to a shared-memory model. However, neither of these approaches deal with a pure shared-nothing architectures or with data spaces larger than two dimensions.

Space partitioning can be parallelized by regularly dividing the data space into bucket extents as before, and then assigning bucket extents to different processors. The parallelization of *PBSM* outlined in [18] follows this approach. After space is partitioned, data is redistributed accordingly and joins are effected independently. As is pointed out in [18], data skew can by addressed by using tiling to fine-partition multidimensional space. The tiles can then be assigned to processors via hashing to balance the load. In general, the larger the data skew, the more finely the space must be partitioned. However, increasing the partitioning granularity can result in a higher degree of data replication.

3.2 Parallel ϵ-kdB Algorithm

Our approach to the parallel proximity join problem, as is explained in more detail below, is to let the ϵ-kdB tree handle the data skew. Regardless of the data distribution, an ϵ-kdB tree produces data buckets with manageable and consistent sizes. Since the ϵ-kdB tree is also cheap to build, we have a performance edge over other index-based parallel join algorithms. On the issue of workload balance, we build an ϵ-kdB tree using the entire dataset before we commit ourselves to or even consider any workload assignment. This allows us to base our workloads on a detailed knowledge of the data distribution. Finally, rather than assign workloads based on data or subspace size, we divide the workload based on join costs, since balancing the workload ultimately depends on the time required to perform each join. We will focus on describing the self-join of a single dataset; however, extending the algorithm for non-self joins of two or more datasets is straightforward.

3.2.1 Building the Tree

To maximize parallelism, we require that each processor independently build an ϵ-kdB tree using all of its local data. We also require that these trees be "identically structured". Our intent is that each processor's tree represents a portion of a common global ϵ-kdB tree; otherwise, we would be forced to treat each structure separately and would face the problem of joining N independent trees with themselves as well as each other. Note that since each processor holds only $1/N$ of all the data-points, the ϵ-kdB trees will be smaller than if we had built a single tree using the entire set of data. To build trees with identical structure, we take advantage of the ϵ-kdB tree's use of a global split ordering. Before processors begin building, they agree on the split ordering they will use. This goal is achieved by having a coordinator who chooses and communicates the split-order, or by having each processor choose the split-order "randomly" but starting with the same random seed.

Since the data on each processor is different, there will be leaves on some processors which on others overflowed and had to be split. An example of this is shown in Figure 3; it shows two processors having built otherwise identical ϵ-kdB trees except that each processor has split a leaf node that the other has not. This structural discrepancy can be resolved by communicating

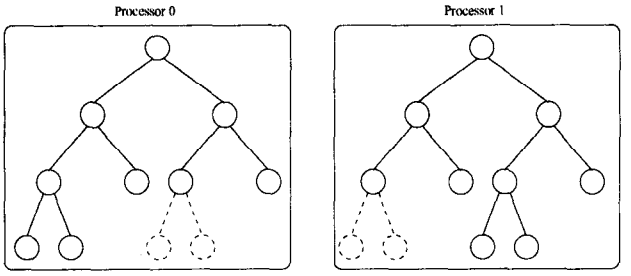

Figure 3: Two-processor ϵ-kdB Trees

tree structure at the end of the build process. Another approach is for processors to periodically broadcast asynchronous messages during the build phase indicating which leaves they have recently split. The latter approach has the advantage in that processors receiving the message will be able to correctly split a node before it becomes full with data. Since messages are sent asynchronously and order does not matter, this communication requires little overhead. Regardless of the approach used, processors perform the specified splits upon receipt of a message. This is illustrated by the dashed subtrees in Figure 3. Since processors build structurally identical trees and work with equal amounts of the data, the build costs remain well balanced.

3.2.2 Performing the Join

Once processors have built the ϵ-kdB structures, we must create and assign workloads. For that, each processor must first know the global size of each leaf in the ϵ-kdB tree so that join-cost estimates can be calculated. Processors therefore walk their local tree in depth-first order and copy leaf sizes into an array. Since the trees have identical structure, global leaf sizes can be determined by performing a vector sum of these arrays.

Once the global size of each leaf is known, each processor executes the ϵ-kdB self-join algorithm on its local tree. However, instead of joining the leaves, the processors simply note the join to be performed and its cost. In our implementation, this is done by creating a small object that identifies the two leaves R and S being joined, and then storing this join object with a linked-list on leaf R. The cost of the join is estimated by the potential result size: the self-join of a single leaf R is assigned a cost of $\frac{|R|*(|R|+1)}{2}$. Joins between two different leaves R and S are assigned a cost of $|R|*|S|$. While enumerating joins, we also keep a running total of all the join costs.

Once join enumeration is complete, we are left with the task of assigning the joins to the different processors. As is explained further below, each processor receives a disjoint subset of the enumerated joins with a cost that is roughly $1/N$ of the computed total cost. We then redistribute the data so that the joins can actually be carried out. For example, if processor P_0 is assigned the join $R \bowtie S$, then each processor will send its local data for leaf R and leaf S to P_0 so it can effect the join. Data replication occurs whenever an ϵ-kdB leaf is required by more than one processor's workload. Note that instead of directly assigning data or subspaces to each processor, we are assigning joins. This will ensure that the amount of work and time required for each assignment is as equal as possible.

Workload Creation

When deciding how to partition the joins among the processors, we want to minimize replication and any associated communication costs. One way to achieve this is to "cluster the joins" such that joins involving the same set of data buckets are assigned to a single processor. We could do this by analyzing the join list, but instead we can again exploit the ϵ-kdB tree to our advantage. Recall that a node in an ϵ-kdB tree can only join with the adjacent siblings of itself and its ancestors. So, we once more walk the tree in depth-first order, and for each leaf visited, we assign the stored join objects to one processor. Once we have created a full assignment, we continue walking and assign joins to the next processor. This should achieve a reasonable clustering of data buckets with very little effort thereby keeping replication and communication low.

As we are creating assignments, we also note which data leaves are needed for each processor. Note that each processor is performing this same workload-creation algorithm so that they will each know which leaves to send to which processors. However, aside from keeping a running count of assignment costs and noting which processor requires which set of leaves, a processor will only keep its own assignment; join objects that are assigned to other processors are deleted. Thus joins are "assigned" to the local processor by leaving them on the linked list of the ϵ-kdB node.

Workload Execution

Once the assignments have been created, processors begin redistributing data asynchronously. Since we do not want processors to flood the network by sending their entire dataset out at once, we use flow control and have processors send data to each recipient in depth-first order. This ensures that processors do not wait long between receipt of the two halves of a join. This serves to minimize the length of time each leaf's data must be kept in memory, and thereby minimizes the total memory consumed at any given time. Once a leaf's data has been redistributed, it is deleted by the

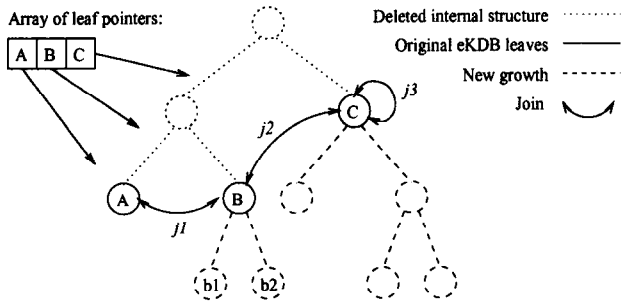

Figure 4: Second Growth Phase

sending processor if it itself does not require the leaf for its join workload.

When all the data belonging to a leaf has been received, a processor executes as many of the joins involving that leaf as possible. This is done by examining the leaf's linked list of joins and for each join, checking if the other half has finished receiving its data. If so, the join object is deleted and the join executed. Otherwise, we remove the join object and append it to the linked list of the other leaf. When we are later signaled that that leaf has finished collecting data, we will see the join object and execute the join then. When all joins involving a leaf have been executed, that leaf and its data are deleted.

3.2.3 Performance Considerations

Beyond the algorithm details described above, there are several performance considerations that an efficient implementation needs to take into account. First, we note that since we have explicitly enumerated all the joins, we no longer need to retain the internal ϵ-kdB structure. If we collect pointers to all of the leaves into an array, we can delete the rest of the ϵ-kdB tree and free up memory. This is illustrated in Figure 4.

Another performance concern is the size of the ϵ-kdB leaves after data redistribution. Each leaf assigned to a processor will have to accommodate the redistributed data of the corresponding leaves of $N-1$ other processors. This will result in ϵ-kdB leaves holding up to N-times as much as those of a serially-grown ϵ-kdB tree. This can result in considerable join time since the processors will be executing joins that are N-times as large as those on a serial processor. To solve this problem, processors continue to grow the tree as data is redistributed. For example, node B in Figure 4 is a leaf before redistribution. However, as data for leaf B is received from other processors, it becomes full and is split in this "second-growth" phase, resulting in the new leaves $b1$ and $b2$. This is exactly what would have happened if this tree had been grown on a serial processor. Thus, the joins that are finally executed by the parallel algorithm are exactly the joins that would be executed by the serial algorithm working with the same dataset.

A detailed description of these performance considerations and how they impact the implementation can be found in [19].

3.3 Parallel Space Partitioning

For comparison purposes, we have also implemented a parallel space-partitioning algorithm for performing proximity joins. Our implementation fits within the hash-join framework[14] in that we divide space into a regular multidimensional grid and join corresponding partitions. Join work is distributed across the multiprocessor by dividing the set of bucket extents equally among the N processors. We also employ a separate set of buckets for holding replicated datapoints; this allows us to avoid generating duplicates in the join results which some algorithms must later filter out. The number of data buckets M is chosen to be fairly large compared to the size of the multiprocessor. This not only ensures smaller and more efficient joins, but also allows us to balance the workload similar to how tiling is used in *PBSM* [18].

3.3.1 Implementation Details

Since we will be partitioning the data space into M subspaces (where $M > N$), each processor allocates an array of M/N data buckets for storing data points. Processors also allocate an additional array of M/N buckets for storing data-points that have been replicated across subspace boundaries. Subspaces are assigned to processors using round-robin. Subspace i is therefore represented by data bucket $B[i \ / \ N]$ on processor $P_{i \bmod N}$; any data-point that is within ϵ-distance of subspace i will be inserted into the replica bucket $R[i \ / \ N]$. A two-dimensional example is shown in Figure 5. Separating the replicated data-points from the originals avoids the generation of duplicate matches.

The self-join algorithm proceeds as follows. As processors scan their local dataset, each data-point is examined to determine to which subspace i the point belongs. The data-point is then sent to the processor responsible for that subspace, where the point is then inserted into bucket $B[i \ / \ N]$. The data-point is also sent to to any processor responsible for a subspace j that is within ϵ-distance of the data-point. These processors insert the data-point into the corresponding replica bucket $R[j \bmod N]$. Note that it is possible for the original subspace and/or multiple neighboring subspaces to belong to a single processor. For example, the point $x1$ in Figure 5 resides in data bucket $B[0]$ and is replicated in replica bucket $R[2]$; both of these buckets reside on processor P_0. Although a data-point is not replicated for each instance, a pointer to

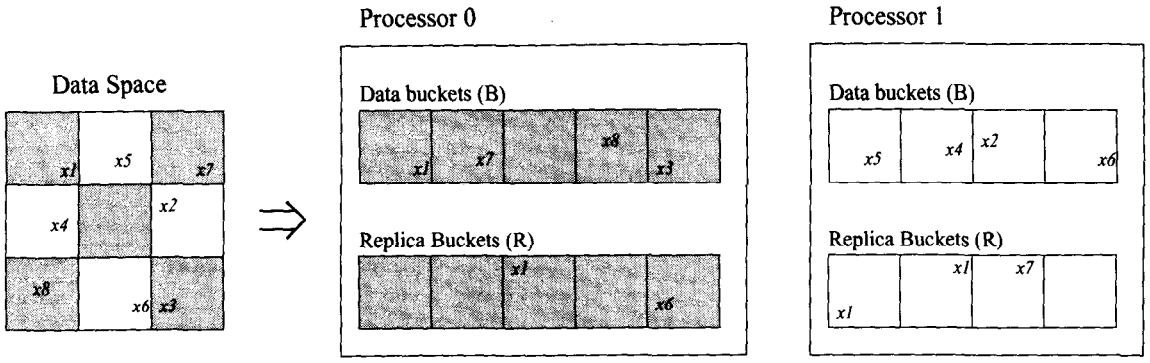

Figure 5: Space Partitioning: 2D example for 2 processors

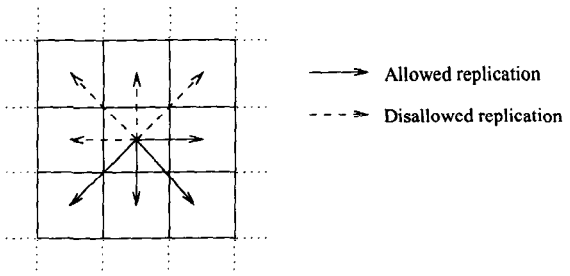

Figure 6: Replication directions in 2D

the data-point must still be inserted into each required bucket. After the data has been redistributed, processors self join each of their data buckets $B[i]$. Proximity joins across subspace boundaries are taken care of by joining each bucket $B[i]$ with its corresponding bucket of replicas $R[i]$.

3.3.2 Performance Considerations

As with the ϵ-kdB algorithm, there are several performance considerations that should be taken into account. Since a data-point is never replicated more than once on any single processor, care must be taken to avoid sending a data-point more than once to any single processor. This is done during the build phase by enumerating all ϵ-near subspaces of a data-point before it is redistributed. If the data-point is destined for multiple buckets on any single processor, the data-point is sent once along with a list of all the buckets that should reference the data-point. Redistributed data is packed into larger messages so that we do not incur a communication call for each data-point. Messages are also sent and received asynchronously so that processors do not spend time waiting.

Finally, if we actually replicate a data-point into every subspace that is within ϵ-distance, we will generate duplicate matches (as well as extra work). Figure 5 shows an example of this where points $x3$ and $x6$ can each potentially be replicated into the other's subspace. To avoid this, we consistently replicate in only half of all possible directions. This is illustrated for two-dimensions in Figure 6, where the solid lines indicate allowed directions for replication and the dashed lines indicated disallowed directions. This results in data-point $x6$ being replicated across the boundary, but not $x3$.

4 Performance Evaluation

We have implemented both the parallel ϵ-kdB and space-partitioning proximity join algorithms on an IBM SP2 [10] using the MPI-standard communication primitives[8]. The use of MPI allows our implementation to be portable to other shared-nothing parallel architectures, including workstation clusters. Experiments were conducted on a 16-node IBM SP2 Model 302. Each node in the multiprocessor is a Thin Node 2 consisting of a POWER2 processor running at 66.7MHZ with 256MB of real memory. Attached to each node is a 1GB disk. The processors run AIX level 4.1 and communicate with each other through the High-Performance Switch with HPS-tb3 adapters. See [10] for SP2 hardware details.

To study the algorithms' sensitivity to different sized inputs, we generated synthetic datasets with both uniform and Gaussian distributions. Data-points were generated with eight dimensions with the values in each dimension ranging from -1.0 to 1.0. The Gaussian mean and standard deviation were set at 0.0 and 0.25 respectively. All experiments use an ϵ value of 0.1 unless otherwise noted. Further experimental results studying the performance characteristics of the parallel ϵ-kdB algorithm can be found in an expanded version of this paper ([19]).

4.1 Algorithm Comparison

In this section, we compare the performance of the parallel ϵ-kdB algorithm with that of space-partitioning. Due to the ϵ-kdB tree's ability to dynamically adjust to data skew present in a data set, we expect it to be a more robust algorithm than space-partitioning. While the space-partitioning approach should do well

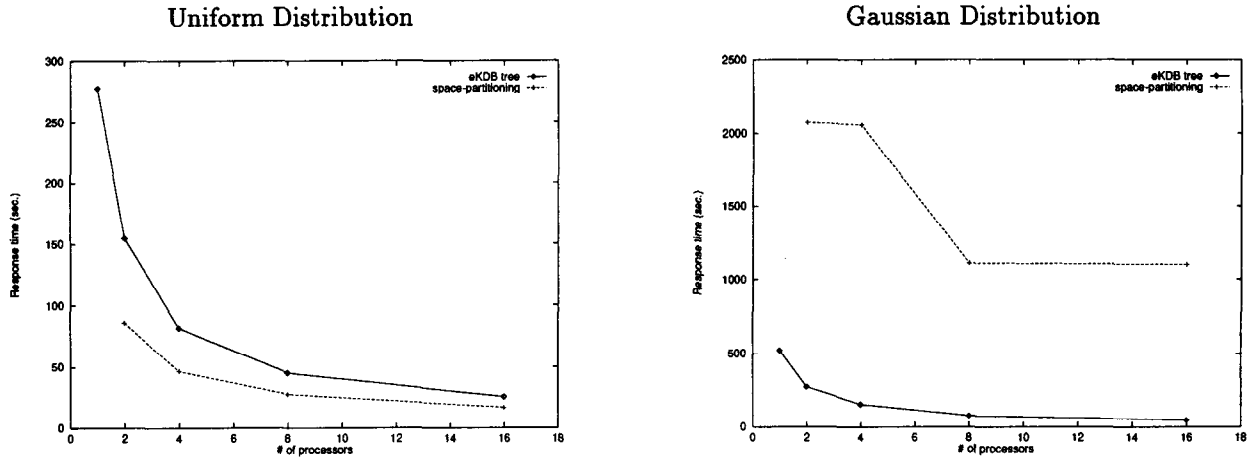

Figure 7: Speedup of ϵ-kdB and space-partitioning algorithms

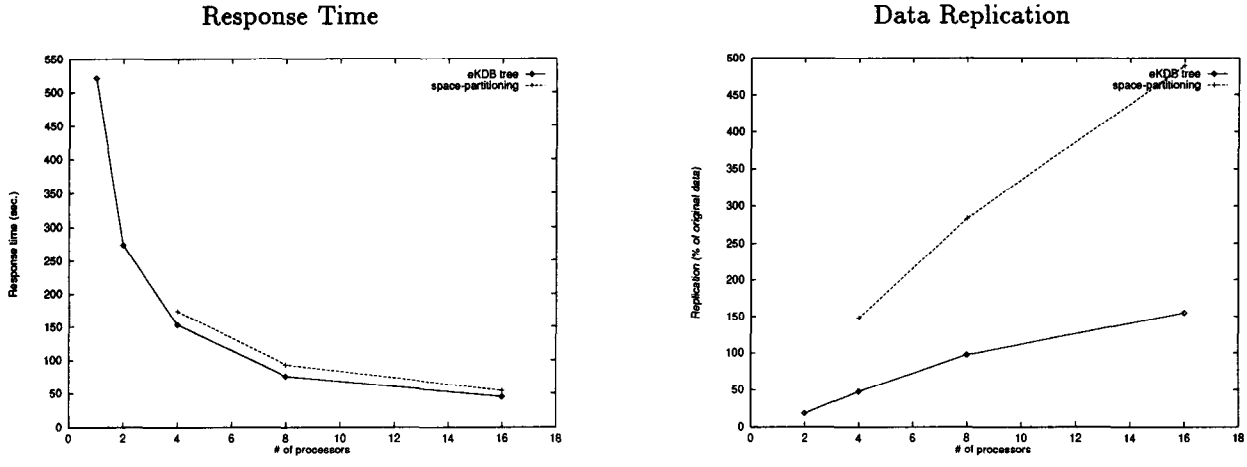

Figure 8: Speedup on Gaussian data with finer partitions

when the data has a uniform distribution, we expect its performance on skewed datasets to be sensitive to the partitioning granularity. Furthermore, determining what is an appropriate partitioning granularity for a given dataset is a hard problem. For our experiments, we chose to divide the data space for the space-partitioning algorithm such that the number of data buckets is roughly equal to the number of leaves used by the ϵ-kdB tree. Data space is divided regularly in all dimensions.

Since join results generally grow $O(N^2)$ with the size of the input dataset, we fixed the dataset to be joined and changed the size of the multiprocessor. Since we did not implement a serial version of the space-partitioning algorithm, performance numbers for that algorithm begin at two processors. Graphs show total response time, which includes the time required to load, build and join a dataset. Costs associated with each algorithm's build and join phases are not directly comparable as the ϵ-kdB algorithm redistributes data during the join phase, whereas the space-partitioning algorithm redistributes data during its "build" phase.

For our first experiment, we ran both algorithms on a synthetic dataset of 500,000 data-points with uniform distribution (see Figure 7). We then varied the multiprocessor size from 1 to 16 processors. As expected, space partitioning does well when the data to be joined is uniformly distributed. Both algorithms scale almost perfectly with the ϵ-kdB algorithm trailing slightly due to it being a more complex algorithm. We then ran experiments on a synthetic dataset of 500,000 points with Gaussian distribution. With this skewed dataset, the inflexibility of the space-partitioning approach becomes apparent. On 16 processors, the variation in each processor's total response time varied from about six seconds to almost eleven-hundred. This severe workload imbalance forces the space-partitioning approach to run several times longer than even the serial version of the ϵ-kdB algorithm. In contrast, the robustness of parallel ϵ-kdB algorithm is readily apparent in not only overall

execution times, but in individual processor response times that varied by at most 15 seconds. Furthermore, the ϵ-kdB algorithm continues to exhibit near-perfect speedup on this highly skewed dataset.

Further analysis of the results revealed that the problem with the space-partitioning algorithm was that the data space was not partitioned finely enough for round-robin assignment to create balanced workloads. This leaves a handful of processors to do most of the work. To verify this hypothesis, we increased the number of subspaces from 15,000 (the number automatically created by the ϵ-kdB algorithm) to over 1.5 million. Figure 8 shows the new speedup results on the same Gaussian dataset (the original ϵ-kdB numbers are repeated for comparison). While the response times are now comparable, space-partitioning pays a huge penalty in replication costs. When executing parallel joins with such large epsilon values, we expect to see a fair amount of replicated data; however, the replication associated with space-partitioning in Figure 8 is extremely high. Furthermore, this graph represents only replicated data objects; with space-partitioning, a processor may have many buckets containing pointers that all reference a single in-memory data-point. In this experiment, the total number of pointers present on all processors was over 9 million. The problem with high replication is that workloads may become too large to fit in a processor's memory. In Figure 8, the 2-processor configuration aborted because of insufficient memory. Thus, we cannot solve the sensitivity of space-partitioning by always running with a fine partitioning — doing so may prevent the algorithm from executing at all.

To summarize, the problem with space-partitioning is that its performance depends critically on the parameter used for data-space partitioning. If we are lucky to have chosen the right parameter for a given data set, we will have good performance. On the other hand, if we partition the space too coarsely, we can have a large performance penalty due to work load imbalance. Conversely, if we partition the space too finely, we can overwhelm system resources with too much replicated data. In contrast, the parallel ϵ-kdB algorithm is robust, as it has built-in capability for skew handling.

4.2 Sample Application: Similar Time Series

For our last set of experiments, we return to the problem that originally motivated us — discovering similar time-series[2]. As discussed in the Section 1, a significant part of this data-mining problem is proximity joining points in w-dimensional space. We can perform this step in parallel by using our ϵ-kdB proximity-join algorithm. Note that the second step of matching can be easily parallelized by distributing the atomic matches equally among the processors.

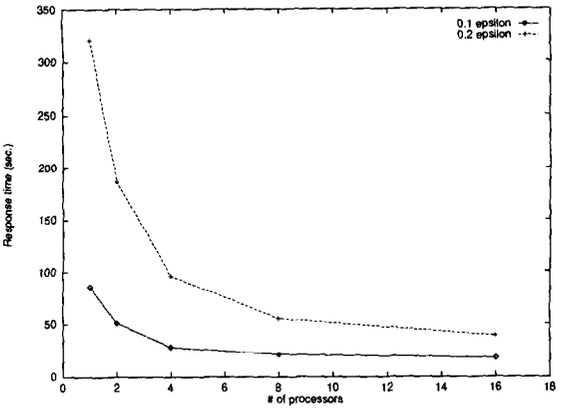

Figure 9: Speedup of ϵ-kdB algorithm on mutual fund dataset

For our time-series data, we used a set of the daily closing prices of 800 U.S. mutual funds for the dates Jan 4, 1993 through March 3, 1995. Using a sliding window of size 8 to generate the atomic subsequences, the resulting dataset to be joined consists of about 320,000 data-points. The data was obtained from the MIT AI Laboratories' Experimental Stock Market Data Server (now StockMaster at www.stockmaster.com). A speedup graph showing response times for the proximity join is given in Figure 9. Speedups using two different values of ϵ are shown. It should be noted that the data is highly skewed, as each data-point is independently scaled to have a minimum and maximum value of -1 and 1 in any dimension. The speedups, however, remain close to ideal.

5 Conclusions and Future Work

We have presented a new parallel algorithm for performing proximity joins on high-dimensional points. Its use of the ϵ-kdB tree makes it a fast and robust algorithm that automatically handles high-degrees of data skew while maintaining near-ideal scalability. We compared the ϵ-kdB algorithm to a space-partitioning implementation and showed that response times were comparable to or better than those of space-partitioning without that approach's sensitivity to dataset distributions. We also confirmed the performance of the parallel ϵ-kdB algorithm on real-life datasets from a data-mining application.

For future, it might be worthwhile to explore the post-build workload partitioning approach used by the parallel ϵ-kdB algorithm in the context of space partitioning. As in the ϵ-kdB algorithm, processors could collect local data-points into data buckets without performing redistribution. After the initial load, processors would exchange data bucket information and then

partition the workload based on join-cost estimations. Since workload balancing is performed after the data has been fully examined, we can use a more sophisticated assignment algorithm than round-robin to create the join workloads. A good approach might be to use space-filling curves such as the Hilbert curve[15] to create a total ordering of the data buckets. The buckets would then be assigned to different processors by partitioning the ordering into contiguous ranges. This could take advantage of the clustering capabilities of space-filling curves and help minimize the amount of data replication. Of course, the problem of deciding space partitioning granularity still remains, although the algorithm's sensitivity to it should be reduced due to the explicit use of workload balancing. It may also be advantageous to split just a few of the dimensions into ϵ-width partitions instead of dividing the multi-dimensional space regularly. However, this introduces the additional question of choosing which dimensions to split. Ultimately, the parallel ϵ-kdB algorithm is likely to retain the advantage since partitioning of space in that algorithm is dynamic and automatic.

Recently, another serial spatial-join algorithm (the *Size Separation Spatial Join*) was presented in [12]. It is a space-partitioning algorithm but differs in that it uses multiple levels of partitioning with increasing degrees of granularity. The algorithm appears to perform well on two-dimensional point data — even when that dataset is skewed. It would be interesting to see how well this algorithm extends to higher dimensions and how well it can be parallelized, and then to compare it to the parallel ϵ-kdB tree.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth Int'l Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[2] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, pages 490–501, Zurich, Switzerland, September 1995.

[3] T. Brinkhoff, H. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-Trees. In *Proc. of 12th Int'l Conference on Data Engineering*, New Orleans, USA, February 1996.

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using *R*-trees. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, Washington, D.C., May 1993.

[5] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. In *IEEE Transactions on Knowledge and Data Engineering*, pages 44–62, March 1990.

[6] C. Faloutsos. Multiattribute hashing using gray codes. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, May 1992.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.

[8] M. P. I. Forum. MPI: A Message-Passing Interface Standard, May 1994.

[9] E. G. Hoel and H. Samet. Algorithms for Data-Parallel spatial operations. Technical Report CS-TR-3230, University of Maryland, February 1994.

[10] International Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995.

[11] H. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, May 1990.

[12] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1997.

[13] M. Lo and C. V. Ravishankar. Generating seeded trees from data sets. In *Proc. of the Fourth International Symposium on Large Spatial Databases*, Portland, ME, August 1995.

[14] M. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, May 1996.

[15] B. Moon, H. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of hilbert space-filling curve. In *IEEE Transactions on Knowledge and Data Engineering*, March 1996.

[16] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[17] J. A. Orenstein and T. Merrett. A class of data structures for associative searching. In *Proc. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984.

[18] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[19] J. C. Shafer and R. Agrawal. Parallel Algorithms for High-dimensional Proximity Joins. Research Report, IBM Almaden Research Center, San Jose, California, 1997. Available from `http://www.almaden.ibm.com/cs/quest`.

[20] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U.K., April 1997.