General-Purpose Spatial Decomposition Algorithms: Experimental Results

Stephen R. Tate
and
Ke Xu
University of North Texas

In this paper, we experimentally explore four different hierarchical space decomposition algorithms. Three of these algorithms are known from prior literature, whereas the fourth was derived as a result of initial experimental work with the previous algorithms, and is presented for the first time in this paper. The new algorithm has several variations, and one of them (called TX-AL) is consistently faster than all of the other algorithms, despite the fact that its worst-case asymptotic performance is suboptimal (TX-AL has worst-case complexity $\Theta(n \log n)$, where a couple of the other algorithms/variants have complexity $O(n \log \log n)$). This shows the validity of the bucketing technique introduced by algorithm RT, while making some sacrifices on asymptotic complexity to allow for smaller constants in the running time. We examine the decomposition algorithms by themselves and also in the context of three different applications: closest pair, all nearest neighbors, and n-body force computation.

Other results presented here include: tests that show the important practical benefit of finding tight bounding boxes for regions prior to testing region dependencies and running applications; tests that show how the amount of non-uniformity in input distribution affects decomposition time; and tests that show that this framework of general decomposition plus application is indeed competitive with algorithms that are designed to solve specific problems.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Spatial Decomposition, computational geometry, experimental algorithms

This research was supported in part by Texas Advanced Research Program Grant 1997-003594-019.

Name: Stephen R. Tate
Affiliation: University of North Texas
Address: Department of Computer Science, University of North Texas, P.O. Box 311366, Denton, TX 76208–1366
Name: Ke Xu
Affiliation: University of North Texas
Address: Department of Computer Science, University of North Texas, P.O. Box 311366, Denton, TX 76208–1366

1. INTRODUCTION

Hierarchical space decompositions have proved to be very important for constructing efficient algorithms that solve problems with a geometric basis (references [Bentley 1980; Vaidya 1989; Callahan and Kosaraju 1995] are particularly prominent examples). From simple geometric algorithms like closest pair computation to more complex algorithms such as the tree-based algorithms for n-body force computation, a key component typically is the partitioning of space into regions that can be treated as separate units, and combined to form larger regions when possible. All of the problems we examine in this paper have as input a set of points in d-dimensional space (we restrict d to be 2 or 3 in our experiments), and so dividing space into non-overlapping regions induces a partitioning of the input points.

While this spatial decomposition theme is underlying many algorithms, in many cases the application is ad-hoc. A key contribution of Callahan and Kosaraju [Callahan and Kosaraju 1995] unified these notions under a common framework, defining concepts such as *fair split trees* and *well-separated realizations* that can be applied in many different situations. The basic idea is that given a decomposition tree, you next compute a well-separated realization, which is a list of pairs of regions that can be enclosed within balls of the same diameter which are separated by a distance that is at least a constant fraction of the ball diameter (this constant is the *separation constant*, and is a parameter that can be adjusted for different applications). In many applications, such well-separated regions can be processed as single units, rather than separately processing the potentially large number of points contained in the region. Callahan and Kosaraju go on to show that if the decomposition is a fair split tree, which is true for all decompositions in this paper, the size of the realization (i.e., the number of region pairs) is $O(n)$, and so the applications also typically run in $O(n)$ time. In their work they applied these notions to derive efficient algorithms for the closest pair problem, for the k-nearest neighbors problem (including a fairly simple algorithm for the sub-case of all nearest neighbors), and for n-body potential field computation [Callahan and Kosaraju 1995].

In this paper, we implement and explore four different spatial decomposition techniques that fit into the Callahan and Kosaraju framework, including a new algorithm that is presented for the first time in this paper, and that was derived as a result of experimental experience with the previous algorithms. This new algorithm outperforms the others in practice, but requires modest assumptions about the input. We also consider the decompositions in the context of three different applications: closest pair, all nearest neighbors, and n-body computation.

1.1 The Algorithms

Here we give a brief description of each of the decomposition algorithms that we test.

Algorithm Reg constructs a region-based k-d tree (see, for example, [Goodrich and Tamassia 1998, pp. 597ff]). This is a regular spatial decomposition, where regions are subdivided by taking each dimension in turn and splitting each

non-empty region into two identical size subregions along that dimension. If we consider one round of splitting in each dimension to be one "large split," then we get the well-known quad-tree data structure in two dimensions, and the oct-tree in three dimensions. Many algorithms are based on this style of splitting, including the decomposition used in Greengard's original multipole algorithm for the n-body problem [Greengard and Rokhlin 1987; Carrier et al. 1988]. This algorithm has the benefit that it is simple to implement, and the constants in the running time are quite low. The time complexity depends on the actual distribution of the input data, and when the input is uniformly distributed the expected time is $\Theta(n \log n)$. Unfortunately, as the data becomes more non-uniform the efficiency of Reg decreases, and in fact when the depth of the tree cannot be limited the running time is unbounded. Note that in our experiments, all data meet an additional requirement that input coordinates are represented in fixed point with $O(\log n)$ bits of precision, which does limit the depth of the decomposition tree to $O(\log n)$ and consequently bounds the running time by $O(n \log n)$. Furthermore, while we can still generate non-uniform data under this restriction, our experiments show that Reg is not terribly sensitive to the input distribution, and the simplicity of the algorithm leads to very fast code for a variety of distributions (see the end of Section 2 for more details).

Algorithm CK (from the paper of Callahan and Kosaraju [Callahan and Kosaraju 1995]) bases the region decomposition on the positions of the points, rather than blindly splitting each region in half. This decomposition is build from the top down, much like Reg does, partitioning the input points as regions are split. The chief benefit of this algorithm is that the time complexity is $O(n \log n)$ in the algebraic computation model, regardless of input distribution.

Algorithm RT (from the paper of Reif and Tate [Reif and Tate 1999]) beats the asymptotic running time of the two previous algorithms by making a mild assumption about the input representation, effectively limiting the number of bits of precision for each input coordinate to $O(\log n)$. This seems to be a reasonable assumption, and by using bucketing ideas similar to those used in radix sort [Knuth 1998] or the van Emde Boas priority queue [van Emde Boas 1977; van Emde Boas et al. 1977], this algorithm achieves a time complexity of $O(n \log \log n)$. Unlike the two algorithms described above, RT builds the decomposition tree in bottom-up phases, where in each phase the points are inserted using bucketing techniques into the leaf nodes of an auxiliary data structure called the "support tree." The tree paths are then discovered by moving from the leaf positions toward the root in the support tree, and the correct decomposition tree nodes and regions are then constructed using the information obtained in this way from the support tree. A careful use of binary search on the tree levels results in the final complexity of $O(n \log \log n)$.

Algorithm TX (introduced in this paper[1]) was designed based on our experience implementing the three algorithms described above, and has two different variants, both of which use bucketing and bottom-up construction. The first

[1]The initials can be viewed as either the authors of this paper, or the state in which the algorithm was invented!

variant, TX-BS, is based on re-mapping the bucketing scheme used by RT, using word-sized bit vector operations in order to keep the time complexity at $O(n \log \log n)$, but with smaller constants so the algorithm is faster in practice. The binary search of levels is retained from RT. The second variant, TX-AL, retains the same flavor of exploiting the input representation and using bucketing, but avoids the binary search of tree levels by explicitly going through all levels from the leaves to the root. The drawback is that TX-AL has time complexity $\Theta(n \log n)$ in the worst case, so the asymptotic complexity is the same as the first two algorithms. Despite this, TX-AL is the fastest algorithm in our experiments, handily beating the other three algorithms and the other variation of this algorithm. The speed comes from a combination of two factors: the bucketing technique introduced by algorithm RT, and the willingness to sacrifice asymptotic complexity for a simpler algorithm with lower constants in the running time.

In addition to examining the practical efficiency of these algorithms, several of the algorithms have parameters which can be adjusted (for example, in the bucket-based algorithms RT and TX the selection of the size of a data structure called the "support tree" can be tuned), and we examine the effect of changing these parameters on both the running time of the decomposition algorithm and on the quality of the decomposition, as measured by the impact on the application using the decomposition.

This study was begun as an experimental study of algorithm RT, to see if the asymptotic improvement of $\Theta(n \log n)$ to $\Theta(n \log \log n)$ was seen in practice. As the difference between $\log n$ and $\log \log n$ is smaller than a factor of 5 for $n < 10^9$, the question of whether the increased intricacy of RT (and the corresponding increase in constant factors) would obliterate the asymptotic improvement, an important practical question, could only be determined through careful implementation and experimentation. The insight that was obtained through these implementations allowed us to design the new TX algorithm, which was an unexpected bonus.

1.2 Experimental Setting and Summary of Results

Our experimental setting was designed to allow great flexibility in exploring decompositions and applications that use these decompositions. We first designed a set of clean C++ class interfaces that captures how applications interact with the decomposition algorithms and resulting decompositions, and then built an experimental environment where different decompositions could be easily "plugged in" to the various applications. For example, in the directory for decomposition Reg, the user can type "`make closestpair2`" to build the closest pair application (in 2 dimensions) with the Reg decomposition, or "`make ann3`" to build the all-nearest-neighbors application (in 3 dimensions) with the Reg decomposition. Conversely, a user working in the directory with the closest pair application could enter the command "`make reg2`" to make that application with the Reg decomposition, or "`make rt2`" to make that application with the RT decomposition. We implemented all four decomposition algorithms, and three applications: closest pair, all-nearest-neighbors, and n-body potential field evaluation (the first two applications work in any number of dimensions, but the potential field evaluation works using the

potential field formulas for two dimensions only — we hope to implement three-dimensional n-body in the near future). All experimental times presented in this paper were obtained on a machine with four 200 MHz Pentium Pro processors with 512k cache each (note however that all our code is sequential for these tests) and 512 MB of shared RAM. This machine was running the Linux 2.2.12 kernel with the egcs 1.1.2 C++ compiler, and tests were run when the machine was otherwise idle (so there was no competition for resources).

All test inputs met the requirements for algorithms RT and TX, so the input consisted of points whose coordinates were integers (or, equivalently, fixed point values) with $O(\log n)$ bits. Inputs were generated by a separate program, which could generate points that were uniformly distributed, distributed as Gaussian or clustered Gaussian, or constructed as several different types of contrived "worst-case" inputs.

The relative running times of the algorithms were fairly consistent across different input distributions, and from slowest to fastest the algorithms were CK, RT, TX-BS, Reg, and TX-AL (see, for example, the main results in Figure 1); however, note that there were some minor re-arrangements in this ordering at extreme distributions and worst-case inputs. The two asymptotically fastest algorithms, RT and TX-BS, were consistently slower than several of the algorithms with worse asymptotic complexity, due in large part to the larger constants in the running time. The fastest algorithm, TX-AL, was significantly faster than the second fastest decomposition, Reg, showing the validity of the bucketing concept, even if the complicated techniques required to reduce the asymptotic complexity to $O(n \log \log n)$ do not seem to be efficient in practice.

Additional tests were performed to determine the practicality of computing tight bounding boxes for all regions prior to computing the well-separated realization, to test the dependence of decomposition time and realization size on the variance of a Gaussian distribution, to measure growth of realization size with input size, and to determine appropriate support tree size for algorithms RT and TX. While the main focus of this project is on the decomposition algorithms, in Section 4 we also examine several issues regarding applications: relative amounts of time spent in decomposition, realization, and application; and dependence of application time on realization size. We also compare times for several application-specific algorithms for closest pair to our general decomposition plus application approach, and show that the general "decomposition toolbox" approach is competitive with specially tuned applications.

2. BASIC DECOMPOSITION TESTS

To get a fair and accurate comparison of the different spatial decomposition techniques on uniform data, we construct a large set of inputs. For two-dimensional data, we begin with a small input of 5000 points, increasing the number of points using a step-size of 5000 until reaching a fairly large input of 500,000 points. For three-dimensional data, we use the same starting size and step size, but the maximum size is only 300,000 points due to memory limitations in our test machine. We also performed some tests with Gaussian and clustered Gaussian distributions, where the clustered Gaussian attempts to generate a natural looking non-uniform distribution of different size clusters of points. Pictures of sample data from these

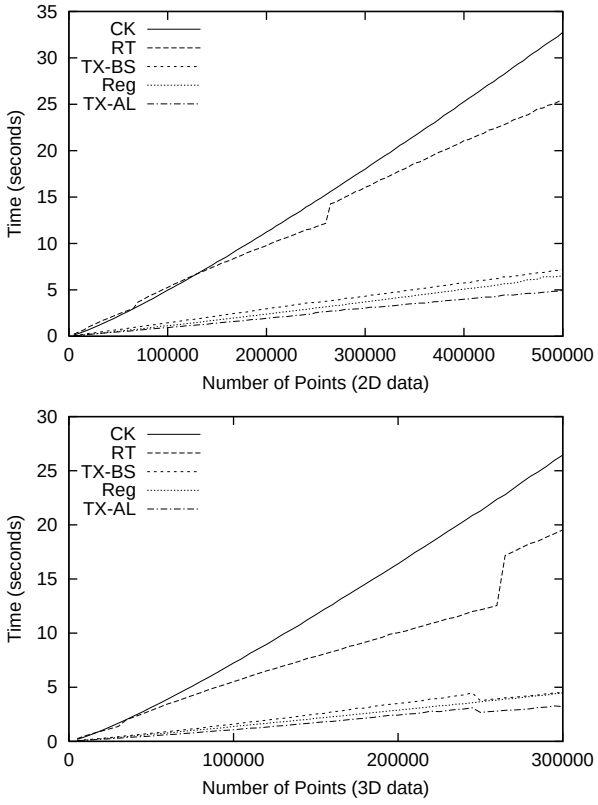

Fig. 1. Running time of the decomposition algorithms (uniform distribution)

distributions are given in the appendix for reference. For each input size, we generate 5 different input instances and run each decomposition 3 times on each instance in an attempt to minimize the effect of timing inaccuracies. The decomposition time is measured and the final result for each input size is the average of the 15 runs. The results of the basic tests on two and three dimensional uniform data is shown in Figure 1; the results from the non-uniform input distributions are in many ways similar, and are included in the appendix for the interested reader (a test specifically exploring the dependence on uniformity is presented later in this section).

The experiment results reflect the significance of the hidden constants in the algorithms' asymptotic complexity. Comparing the algorithms from prior literature, we notice that while the running time of both Reg and CK is $\Theta(n \log n)$, and the running time of RT is $\Theta(n \log \log n)$, for the reasonable range of n values that we tested Reg is much faster than CK and RT on both uniform and the Gaussian-distributed non-uniform data due to its straightforward splitting strategy. Although algorithm CK and RT are able to handle non-uniform inputs more gracefully, they both use complex data structures. In CK, every region to be split maintains for each dimension two doubly linked lists of points (one original and one copy that is manipulated) sorted based on their coordinates in that dimension, and cross-

referenced between dimensions. Splitting a region involves constructing these lists for the subregions as well. In RT, each active node in the support tree, the auxiliary data structure that guides construction of the decomposition tree, maintains a list of its active descendants. The experiments show that these list operations, which require memory allocation and deallocation, are quite inefficient, and while we minimized the effect of this by careful consideration of memory allocation issues (see the next section for details), the added complexity of these data structures results in non-trivial constants in the running time of these algorithms. Our new algorithms TX-BS and TX-AL make several improvements to reduce these constants, and the times for these algorithms are also shown in Figure 1. Further discussion of these algorithms is in Section 3.

2.1 Effect of Non-uniform Inputs

While algorithm Reg has a significant advantage in its simplicity and good performance on uniform data, its biggest drawback is decreased efficiency for non-uniform inputs, with the worst-case in the algebraic computation model (although not with our input restrictions) resulting in unbounded time complexity. Our experiments show that this is a purely theoretical concern for our test data, as Reg is in fact very fast on even non-uniform data in our tests (we tested with both clustered Gaussian data and also an artificially-generated "worst-case" — the results of the latter test are omitted from this extended abstract, but will be included in the full paper). To see to what extent non-uniformity affects the algorithms, we created a set of Gaussian distributed data, each instance containing 300,000 points and with the standard deviation of the instances ranging from 100 to 16,000. The smaller the standard deviation, the more tightly the points are clustered, and at the largest standard deviation of 16,000 the data was visually indistinguishable from uniform data.[2] The results of this test are shown in the upper graph of Figure 2 (the other graph will be discussed in the next section). As can be seen from the graph, the performance of algorithm Reg does get worse as the input becomes more non-uniform, as we suspected, although the degradation is quite mild. Furthermore, CK has only moderate fluctuations with uniformity, and RT and TX get slightly better with non-uniform data (with some anomalous increases over small ranges of standard deviations), which was also expected. However, these minor changes are still not enough to overcome the larger differences in running time constants, and Reg remains the fastest performing algorithm from among those in prior literature.

3. ALGORITHMIC IMPROVEMENTS

After initial implementations and tests, the results and experience with these implementations suggested algorithmic and implementation improvements, which we outline in this section.

All of the decomposition algorithms perform extensive manipulations of lists, especially in the case of CK and RT, whether they are lists of points or lists of tree nodes. Our initial tests showed that a very large amount of time was spent in allocating and deallocating memory for these lists, suggesting that using the

[2]Data that falls outside of our "universe" is wrapped around, which helps large variance distributions become more uniform.

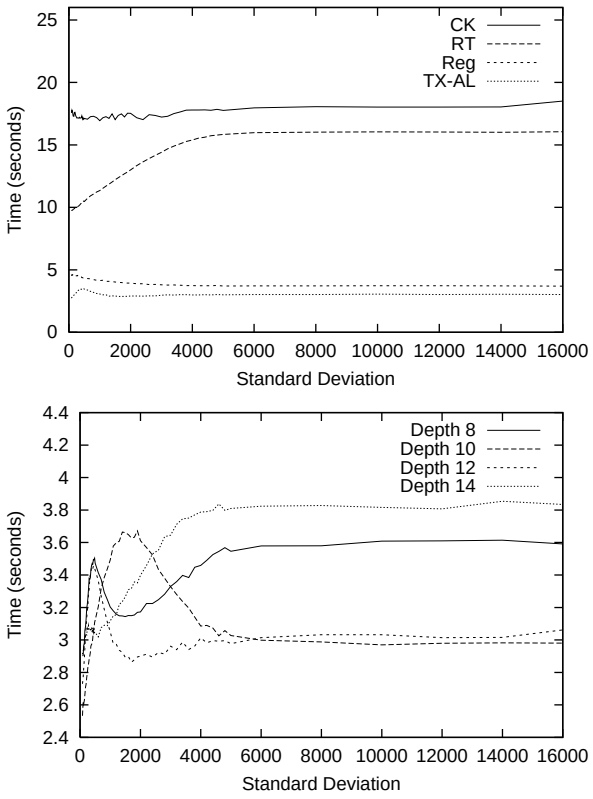

Fig. 2. Dependence of standard deviation (uniformity) on time — 300,000 point inputs

general-purpose `new` and `delete` facilities of C++ for all list node manipulation was inefficient. We changed the allocation strategy for these lists by allocating blocks of list nodes at a time, and maintaining our own free lists for reusing the nodes. This simple data structure change was applied to all decomposition algorithms, and resulted in as much as a 45% speedup.

The next improvement involved changing the bucketing strategy used by algorithm RT — this change is significant enough to consider the result a new algorithm, and this re-mapped algorithm is what we refer to by TX (or TX-BS) in this paper. Algorithm RT introduced the notion of a "support tree", which is essentially a complete region-based k-d tree with a given number of levels. The support tree nodes are stored in an array, and we use bucketing techniques to map points to array indices (or tree nodes). In order to keep the bucketing operation constant time, we map points to array indices by stripping out the appropriate number and position of bits from each coordinate of a point, and then concatenating these bit-strings together with a leading 1 bit. For example, if the bits to be used from dimension 1 are $a_3a_2a_1a_0$ and those from dimension 2 are $b_3b_2b_1b_0$, then the index of this node (which must be on level 8 of the tree since there is a total of 8 bits being used) is the binary number $1a_3a_2a_1a_0b_3b_2b_1b_0$. In d dimensions this takes only $O(d)$ time,

and algorithm RT must do this at most $\log\log n$ times in order to insert a point into the support tree leading to the $O(d \log\log n)$ time bound per point (RT must also overcome several other nontrivial problems to give the overall $O(n \log\log n)$ time bound, but we concentrate here only on the bucketing operation).

While the technique from RT is certainly clean theoretically, the bit extraction and manipulation introduces moderate constants into the running time. Furthermore, simply moving from a node to its parent in the support tree often involves removing a bit from the middle of the node index, and moving up several levels to an ancestor (which RT must do often) involves removing bits from several different locations throughout the node index. The masking and shifting operations required for this can become complex, resulting in a moderate constant overhead for this mapping.

Binary heaps also use a mapping of tree nodes into an array, and while this was a partial inspiration in the development of bucketing for RT, the exact mapping was avoided because points cannot be mapped to arbitrary levels in the tree in constant time. In the bit-string example given above, since the k-d tree we are representing alternates repeatedly among dimension being split, the required tree index for this point in this mapping would be $1a_3b_3a_2b_2a_1b_1a_0b_0$. With $\Theta(\log n)$ bits per coordinate, this seems to require $\Theta(\log n)$ time per mapping in order to interleave all the bits. Despite this apparent increase, some other operations are much simpler. For example, moving up k levels to a node's ancestor only involves shifting the rightmost k bits out of the node index, and finding "representatives" (an important notion from RT that involves clearing certain bits in a node's index) now only involves masking out a certain number of least significant bits. The possibility of simplifying these operations so substantially provided motivation to examine this mapping again to see if something feasible could be obtained.

Since we are considering a word-based computation model, where words contain $\Theta(\log n)$ bits, it turns out we can actually compute this point-to-index mapping in $O(\log\log n)$ time using bit-vector operations on words. The key operation is to "spread out" the bits from each coordinate so they can be combined in an interleaved fashion. We use a divide-and-conquer scheme to spread the bits out: first move the most significant half of the bits up so they start at the correct position, then move up half of each of these halves, then half of each of the resulting quarters, etc. The exact code for this is shown in Algorithm 1 ($<<$ is the "shift left" operator, as in C or C++).

The complexity of this algorithm is clearly $O(\log b)$, and since $b \leq \log n$ we can express the running time as $O(\log\log n)$. Even though this is an improvement over the obvious algorithm for computing these indices, it initially does not seem to be enough. Since RT looks at $\Theta(\log\log n)$ different support tree nodes in order to insert a point, using this algorithm for each indexing operation would increase the time for inserting a point to $\Theta((\log\log n)^2)$. However, careful examination of RT shows that every indexing operation except for the very first involves moving up the tree to an ancestor of the "current node." As described above, this is now a very simple constant-time operation (a right shift), so the process of inserting a point now involves an initial $\Theta(\log\log n)$ time indexing operation, followed by $\Theta(\log\log n)$ additional constant-time indexing operations; therefore, the asymptotic time bound is maintained. Due to page limits for this extended abstract, we omit further details,

Algorithm 1 Bit spreading algorithm: x is input coordinate, b is the number of bits in x, and d is the number of dimensions. Effectively puts $d-1$ zeros between each bit in the input x.

```
s ← 2^⌈lg b⌉ {Shift amount: smallest power of 2 that is ≥ b}
m ← (1 << s) − 1 {Mask: set least significant s bits to 1}
for i ← 1 to d − 1 do
  m ← (m << 2s) OR m {repeat bits throughout mask}
end for
while s > 1 do
  s ← s/2
  m ← m XOR (m << s)
  x ← ((x AND NOT m) << (s(d − 1))) OR (x AND m)
end while
```

which will be available in the full paper, but summarize in the following theorem.

THEOREM 1. *Given m points (a subset of the initial n-point input), TX-BS builds a support tree that is isomorphic to that produced by Algorithm RT, and does so in $O(m \log\log n)$ worst-case time.*

This re-mapping change alone improved the running time by approximately 50%, but also allowed a further improvement: the new mapping made possible simple array-based lists to keep track of non-empty (or "active") support tree nodes, removing still more of the penalties imposed by allocating and freeing linked list nodes. Changing to the array-based lists improved running time by approximately an additional 5%.

Our final practical improvement involves using bucketing with the mapping just described, but we give up on the notion of binary search within the levels of the support tree. Instead, we first place all points into the appropriate leaf nodes of the support tree, and then propagate the non-empty nodes up the tree, joining nodes under a common parent when paths merge. We call this algorithm TX-AL (for "all levels"). Since the support tree has $\Theta(\log n)$ levels, the worst-case time to insert m points becomes $\Theta(m \log n)$. However, it's also important to notice that in a dense support tree (one in which a constant fraction of the leaf nodes are non-empty), the asymptotic complexity of this algorithm is actually $O(m)$, beating out all the previous algorithms! Unfortunately, we cannot guarantee that all support trees used in a decomposition will be dense, so this observation does not help the worst-case asymptotic complexity of the complete decomposition algorithm.

3.1 Support Tree Size

One key parameter that we experimented with in the RT and TX algorithms was the size of the support tree. Any support tree with size $O(n)$ and $\Omega(n^c)$ for some $c > 0$ can achieve the $O(n \log\log n)$ time bound for the asymptotically fastest algorithms. In the theoretical presentation of these algorithms, the support tree is typically $\Theta(n)$ in size, but our experiments show that smaller support tree sizes work better for our input distributions in practice. In particular, the theoretical presentation of algorithm RT [Reif and Tate 1999] gives a support tree with $d\lfloor \frac{1}{d} \log_2 n \rfloor$ levels; however, our experiments with the improved version TX showed that in fact for

uniformly distributed 2d inputs the optimal support tree size grows more closely to the function $d\lfloor\frac{2}{d}\log_{10}(4n)\rfloor$, giving a support tree size of approximately $n^{0.6}$. We do note however, that in other distributions and dimensions this is no longer optimal.

In particular, the discontinuities evident in Figure 1 (particularly the 3d data) are due to non-optimal support tree size selection. This became apparent in tests in which we manually selected the support tree size: For example, our formula says that in 3 dimensions we should switch from 9 levels to 12 levels at $n = 250,000$, but forcing a support tree with 12 levels for $n = 240,000$ improved the running time, showing that while the support tree size formula is optimized for 2 dimensions, it simply switches "too late" in 3 dimensions. Similar experimental exploration shows that other discontinuities in the running time graphs have similar explanations. In addition, the distribution of the input also affects what size support tree is best to use, as revealed by the bottom graph of Figure 2. This graph shows the running time as a function of standard deviation (as explained in the previous section), but with 4 plots in which the depth of the support tree is fixed at 8, 10, 12, and 14. For high variance data (more-or-less uniform), the performance of depth 10 and depth 12 support trees is very close, but as the standard deviation decreases, the depth 12 tree becomes significantly more efficient. At the largest difference, the depth 10 tree is over 25% slower than the depth 12 tree. As the standard deviation decreases further, the depth 14 tree is temporarily the fastest, and then for very small standard deviation the depth 10 becomes best. Unfortunately, at this point, we do not know how to automatically adjust the support tree in such situations. An interesting open problem which we are now investigating involves selecting the proper support tree based on not only the size of the input, but measurable distribution characteristics as well.

4. REALIZATION AND APPLICATION TESTS

While the main focus of our work is on the algorithms for computing the spatial decomposition, we also consider these algorithms in the context of larger applications of the decompositions. After the decomposition tree is built, two more phases are performed: the well-separated realization is computed as described in the Introduction, and then the application is run using this realization. The applications we studied have traditionally been solved in such a manner, although in some previous algorithms the realization is sometimes not explicitly computed. For example, in his original paper on the multipole algorithm for n-body force evaluation, Greengard uses a quad-tree decomposition with the realization implied by the regular geometric structure of this tree (in fact, at the time Greengard developed his algorithm the notion of realizations hadn't been introduced yet).

Our first test using applications was to determine what amount of time is spent in each of the three phases of a decomposition-realization-application combination. Figure 3 shows the results of this test on 2-d uniform data for the best-performing decomposition (TX-AL) and the three different applications. For each of the three applications, the percentage of time spent in each of the three phases is remarkably consistent across all input sizes, and is shown below.

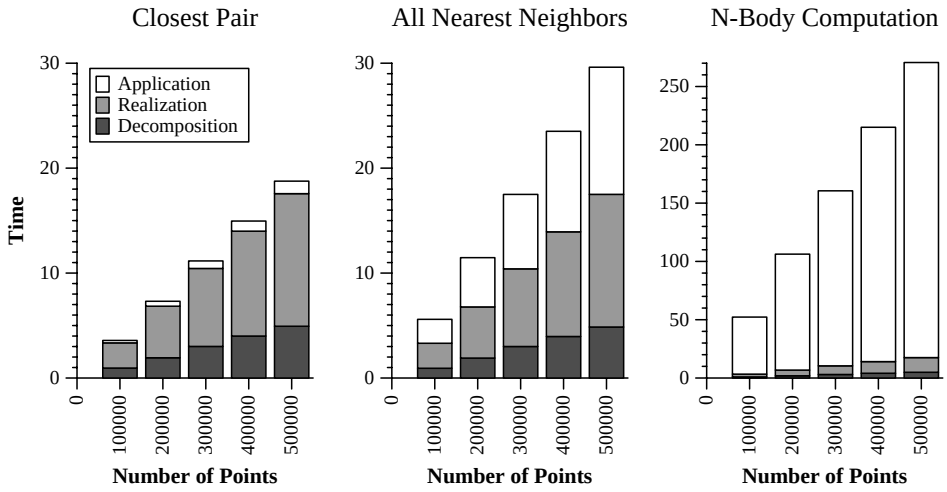

Fig. 3. Running time of different algorithm phases (2-d uniform data)

Application	Decomposition	Realization	Application
Closest Pair	27%	67%	6%
All Nearest Neighbors	17%	42%	41%
N-body Computation	2%	5%	93%

The applications show an interesting range of complexities, from very simple to fairly complex. This also gives a measure of how improvements in the decomposition algorithm get reflected in the overall application time: closest pair can be greatly improved by a faster decomposition, but N-body depends more on fast application code and a faster decomposition has relatively little effect. Note that we only used the fastest decomposition in these tests in order to determine the impact of *future* improvements — the improvement from older and slightly slower algorithms is more significant than indicated by these figures. It is also interesting to note that the realization computation takes up a relatively large amount of time. As the realization is one of the concepts that makes this decomposition framework applicable to a variety of problems, and yet it takes a large amount of the time, we plan to investigate ways of reducing the time required for this phase in the future.

The next test we performed regarding the realization was to simply determine the actual realization size for various inputs. Callahan and Kosaraju proved that the realization contains at most

$$2(n-1)(3(s\sqrt{d}+2\sqrt{d}+1)+2)^d \tag{1}$$

pairs, where s is the separation constant, which we fix to $s = 2.1$ in these tests. The leading term on this bound becomes $1003n$ and $36,400n$ for two and three dimensions, respectively. While this is sufficient to show that the realization is $O(n)$ size (when d is considered a constant), the constants are quite large and the analysis suggests that this bound might not be the tightest possible.

Figure 4 shows the actual realization sizes as produced by both Reg and CK, for both uniform and clustered inputs, with the upper graph being for two dimensions

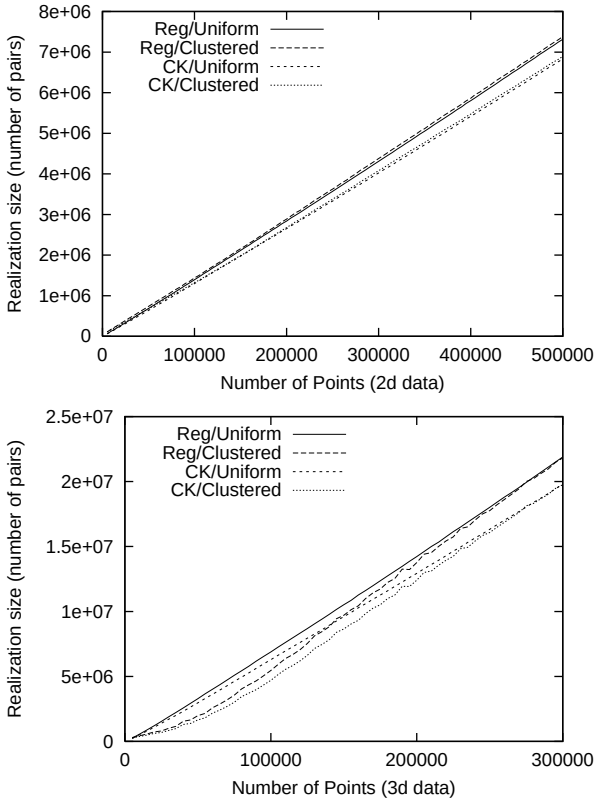

Fig. 4. Realization size of different algorithms and input distributions

and the lower graph being for three dimensions. Note that RT and TX produce the exact same tree (and hence realization) as Reg, so only one such tree is represented in the graph. The point-based decomposition CK produces a slightly smaller realization (consistently around 6.7% smaller in two dimensions), and the distribution has surprisingly little effect on the realization size. Furthermore, the measured realization size grows as $13.7n$ and $66n$ for CK in two and three dimensions, respectively, and at the slightly faster $14.6n$ and $73n$ for Reg in two and three dimensions. This demonstrates that for uniform data, and this particular clustered distribution, the bound (1) is in fact several orders of magnitude larger than necessary.

In our next test, we performed the following experiment: in the traditional top-down regular decomposition (as used by Greengard, for example), the regions in the decomposition tree are defined by the region splitting of the decomposition, with no regard for the points actually contained in the region. Since the realization benefits from regions being well-separated (which, recall, is a function of the region size and region boundaries), it is a sensible extension of these algorithms to compute a tight "bounding box" for the points in each region. These tight boxes are then used to compute the realization. The CK algorithm computes these tight bounding boxes as a matter of course in the decomposition, but computing such bounding

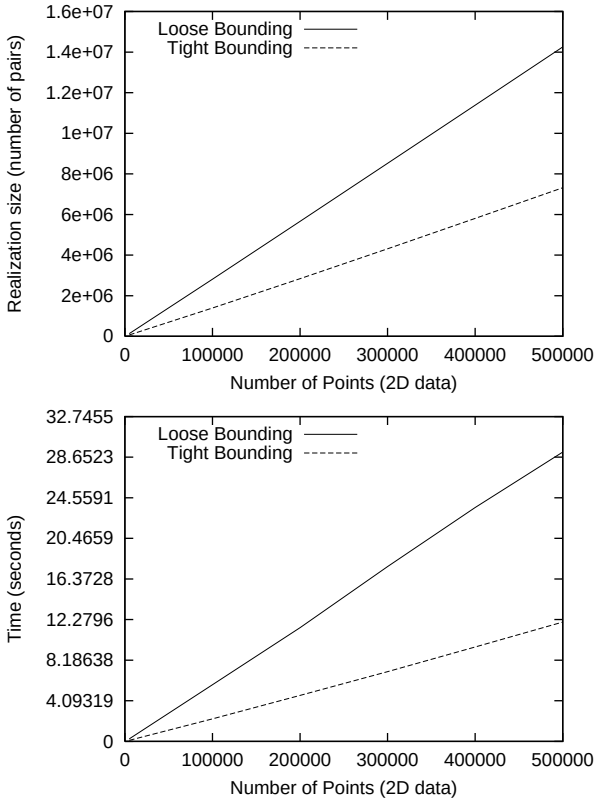

Fig. 5. Tight bounding effect on realization size and application time (all nearest neighbors)

boxes in the top-down Reg algorithm during the decomposition would be somewhat inefficient since it would require scanning the list of points for each internal node for minimum and maximum values in each dimension. We avoid this problem by computing the bounding boxes *after* the decomposition, working from the bottom up and merging minimal regions. This is a fast $O(n)$ time operation.

The results of this bounding box test are shown in Figure 5, where the graph on the top shows the realization size for both tight and non-tight (traditional) bounding boxes on uniformly distributed data in two dimensions. The tight bounding boxes consistently halved the size of the realization. The bottom graph in Figure 5 shows how the running time of one particular application (application phase only), all nearest neighbors, is affected by the tight bounding box. The graph is almost identical in shape to the realization size graph above, showing the linear dependence of the application's time on the realization size. Thus, improvements in the quality of the decomposition (such as using tight bounding boxes) are reflected with faster application times. We pay for the time to compute the tight bounding boxes of course, but tests show that this seems to always be a clear win: for example, computing tight bounding boxes for 500,000 points took 0.8 seconds, but since the realization was smaller it actually took 9.4 seconds less time to compute the

realization! This is a net savings of 8.6 seconds, even before any improvement in the application time. We note that without explicitly computing the realization this notion of a tight bounding box is not applicable, so in particular using the original Greengard multipole algorithm there was no way to get the kind of gains that are possible by using this technique with the decomposition/realization framework.

4.1 Some final tests

In a final test, we compared the running time of our decomposition-based implementation of closest pair and all-nearest neighbors with the running time of a couple of other implemented algorithms for this problem. In particular, we use a simple sweepline algorithm with $\Theta(n^2)$ worst-case complexity but good average case complexity, and for closest pair we also use the publicly available closest pair implementation distributed with LEDA (the Library of Efficient Data types and Algorithms) [Mehlhorn et al. 1999]. We also include experimental (and analytically extrapolated) times for the naive $\Theta(n^2)$ algorithms as an interesting contrast, although this is clearly an unfair comparison.

We also tried the following experiment: since computing and storing the full realization seems wasteful for a single run of a simple application like closest pair, we also included a modified implementation in which we change the realization construction in two ways: we run the closest pair application while the realization is being computed, and we don't store the realization for future use (since it is no longer needed afterwards). We call this modification "Special TX-AL." The times in the table below summarize our findings, where the times are in seconds, and the times for the decomposition-based algorithms reflect the cumulative time for all three phases of the algorithm.

	Closest Pair			All-Nearest-Neighbors		
	100,000	300,000	500,000	100,000	300,000	500,000
$\Theta(n^2)$ search	1140	10,400	28,900	2560	22,700	63,000
Sweepline	0.7	2.8	5.9	5.7	29.8	64.3
LEDA	1.3	5.7	10.9	N/A	N/A	N/A
Special TX-AL	3.5	11.1	18.7	N/A	N/A	N/A
TX-AL w/app	4.1	12.0	20.0	6.2	18.4	31.0

From these results we see evidence of the unfortunate reality that, particularly for simple problems such as closest pair, more general solutions are often slower than special-purpose solutions. In particular, the techniques used in these algorithms get more and more general as you go down the rows of the table (ignoring the first row), and the times get consistently slower. Still, it is somewhat encouraging that we can get within a reasonable factor of the best special-purpose algorithms with the general-purpose decomposition/realization/application framework, and beat the special-purpose algorithm in the case of large inputs for all nearest neighbors.

We would like to compare the other applications with special-purpose code as well, but only implementations of closest pair algorithms were readily available. We will do further studies if other implementations are discovered.

5. CONCLUSIONS

This study began as a test of the practicality of the asymptotically superior algorithm RT, as compared to other previous algorithms. Guided by initial results, this allowed us to develop a new algorithm TX that was faster than the other algorithms we tested, and we believe this to be the fastest decomposition algorithm known at this time. Even though the TX-AL variant is not asymptotically optimal in the worst-case, the lower constants and the efficiency of the bucketing techniques (as introduced in RT) result in an algorithm that is almost twice as fast as the next fastest one in our tests.

Since these algorithmic improvements would not have been made without working with implementations (there is no theoretical improvement over existing algorithms), we believe this speaks very strongly for experimental algorithms work. In particular, while large asymptotic improvements are probably the most important part of good algorithm design, even improvements that do not affect the asymptotic complexity can be worthwhile (and in fact, small asymptotic improvements such as the $\Theta(n \log n)$ to $\Theta(n \log \log n)$ improvements examined in this paper may actually be harmful in practice!).

There are several issues that we are currently investigating, and some results that were omitted due to page bounds for this extended abstract. In particular, we are exploring ways of more effectively determining the support tree size to be used based on measurable data properties, parallel versions of these algorithms, integrating some of these ideas (in particular the realization and tight bounding box ideas) into other existing applications such as the FastCap capacitance extraction tool [Nabors and White 1991], and the effect of changing the separation constant on overall application time.

ACKNOWLEDGMENTS

The authors would like to thank Ajay Nemarugommula and Sandeep Dhopate for coding a large portion of the Reg and CK algorithms, and Xiaobo Peng for coding and investigating the tight bounding box question.

REFERENCES

Bentley, J. L. 1980. Multidimensional divide-and-conquer. *Commun. ACM 23*, 4 (April), 214–229.

Callahan, P. B. and Kosaraju, S. R. 1995. A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM 42*, 1 (Jan.), 67–90.

Carrier, J., Greengard, L., and Rokhlin, V. 1988. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal of Scientific and Statistical Computing 9*, 4, 669–686.

Goodrich, M. T. and Tamassia, R. 1998. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., New York.

Greengard, L. and Rokhlin, V. 1987. A fast algorithm for particle simulations. *Journal of Computational Physics 73*, 325–348.

Knuth, D. E. 1998. *The Art of Computer Programming; Volume 3: Sorting and Searching* (Second ed.). Addison Wesley, Reading, Mass.

Mehlhorn, K., Näher, S., Seel, M., and Uhrig, C. 1999. The LEDA user manual, version 3.8. For more information on the LEDA sofware library, see

`http://www.mpi-sb.mpg.de/LEDA/leda.html`.

Nabors, K. and White, J. 1991. Fastcap: A multipole accelerated 3-D capacitance extraction program. Technical report, MIT Department of Electrical Engineering and Computer Science.

Reif, J. H. and Tate, S. R. 1999. Fast spatial decomposition and closest pair computation for limited precision input. *Algorithmica*. To appear. Currently available as *UNT Department of Computer Science Technical Report N-96-001*, downloadable as `http://www.cs.unt.edu/~srt/papers/decompose.ps`.

Vaidya, P. M. 1989. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Disc. and Comput. Geom. 4*.

van Emde Boas, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6*, 3 (June), 80–82.

van Emde Boas, P., Kaas, R., and Zijlstra, E. 1977. Design and implementation of an efficient priority queue. *Math. Sys. Theory 10*, 99–127.

APPENDIX

A. EXAMPLE INPUTS AND ADDITIONAL GRAPHS

Figures 6 and 7 show example inputs in uniform, Gaussian, and clustered Gaussian distributions, while Figure 8 shows decomposition running times for clustered Gaussian inputs in two and three dimensions.

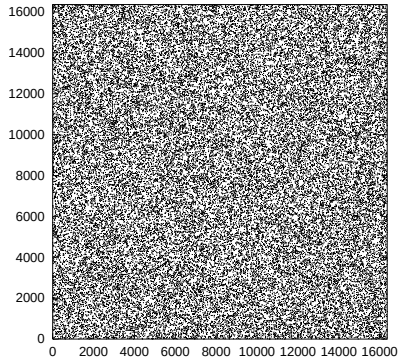

Fig. 6. Uniform distribution

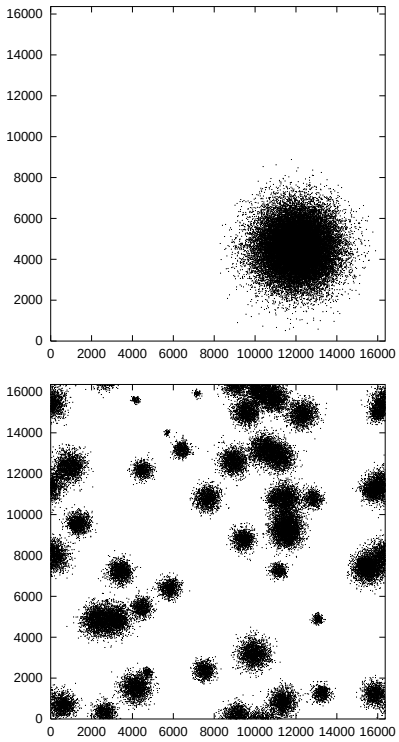

Fig. 7. Gaussian and Clustered Gaussian distribution

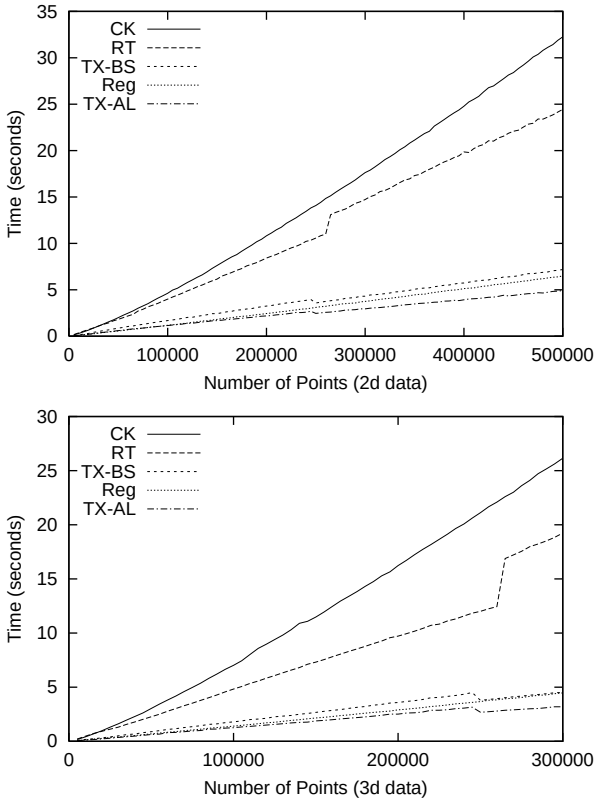

Fig. 8. Decomposition times on clustered Gaussian distribution