1-1-1992

Concurrent Local Search for Fast Proximity Algorithms on Parallel and Vector Architectures

Peter Su
Dartmouth College

CONCURRENT LOCAL SEARCH FOR FAST PROXIMITY ALGORITHMS ON PARALLEL AND VECTOR ARCHITECTURES

Peter Su

Technical Report PCS-TR92-183

October,1992

Concurrent Local Search for Fast Proximity Algorithms on Parallel and Vector Architectures

Peter Su

Dartmouth College and Duke University*

*This work supported in part by DARPA/ISTO Contracts N00014-88-K-0458, N00014-91-J-1985, and N00014-91-C-0114, and by NASA subcontract 550-63 of prime contract NAS5-30428.

Concurrent Local Search

Contact Information:

Peter Su
101 North Building
Dept. of CS
Duke University
Box 90129
Durham, North Carolina 27708-0129
Email: psu@cs.duke.edu
Phone: 919-660-6520

Abstract

This paper presents a fast algorithm for solving the *all-nearest-neighbors* problem:

> Given a set S of n points (or sites) in space, and a set of query points Q, for each $p \in Q$, find the point $s \in S - \{p\}$ that is closest to p.

The algorithm uses a data parallel style of programming which can be efficiently utilized on a variety of parallel and vector architectures [4,21,26]. I have implemented the algorithm in C on one such architecture, the Cray Y-MP. On one Cray CPU, the implementation is about 19 times faster than a fast sequential algorithm running on a Sparc workstation.

The main idea in the algorithm is to divide the plane up into a fixed grid of cells, or buckets. When the points are well distributed, the algorithm processes each query point, q, by searching a small number of cells close to q. Bentley, Weide and Yao first presented this idea for conventional architectures [3], but the technique works equally well on parallel and vector machines, leading to a simple, efficient algorithm.

We can also use the cell technique to solve a wide variety of basic computational problems such as finding closest pairs, sorting, and constructing Voronoi diagrams and Delaunay triangulations.

1. Introduction

On conventional architectures, cell (or bucket) algorithms have been effective in solving a wide variety of computational problems, especially in computational geometry. In this paper, I will show how to implement cell algorithms in a data parallel style on parallel and vector supercomputers. As a case study in using these techniques, I will present an algorithm for solving the all-nearest-neighbors problem and show how well an implementation of the algorithm performs on the Cray Y-MP.

The all-nearest-neighbors problem is easy to state:

> Given a set S of n points in space, and a set query points Q, for each $p \in Q$, find the point $s \in S - \{p\}$ that is closest to p.

This problem is just one in a large class of problems dealing with the proximity relationships of points in space. These problems include finding closest pairs, constructing Delaunay triangulations and constructing Euclidean minimum spanning trees. In addition to many theoretical results, many simple, practical algorithms exist to solve the problems on conventional machines [3,10,14-16,19,24]. In their early paper [3], Bentley, Weide and Yao present a particularly simple algorithm for the all-nearest-neighbors problem when the input is uniformly distributed. The algorithm uses a uniform grid of buckets to solve the problem in linear expected time. The algorithm processes a query point by locating the bucket that the point belongs in and then searching nearby buckets in a outward spiral until the nearest neighbor is found. They also point out that this cell technique can be applied to a wide range of closest point problems, and that it can be adapted to the case where the point set is not so uniform.

The cell-based algorithm is an attractive starting point for a parallel method for this problem since each search is localized, and the search algorithm is not complex. Using this scheme, Levcopoulos, Katajainen, and Lingas [17] present a PRAM algorithm that can solve the all-nearest-neighbors problem in $O(\log n)$ time on $O(n/\log n)$ processors. Mackenzie and Stout [18] show how to improve this to $O(\log\log n)$ time on $O(n/\log\log n)$ processors. While both of these algorithms are based on the cell idea, they use sophisticated techniques to take maximum advantage of the available parallelism available in the PRAM model.

Other known parallel algorithms for this problem concentrate on obtaining good worst-case performance by building the Voronoi diagram of the point set. Thus, each site $s \in S$ must construct a polygon $V(s)$ such that every point in $V(s)$ is closer to s than any other point in S. The algorithm presented by Aggarwal, et al [1]. attempts to parallelize the standard divide and conquer algorithm [14,20]. The merge step proves to be difficult to parallelize effectively, but with some very complicated trickery, the algorithm runs in $O(\log^2 n)$ time on n processors. Cole, Goodrich, and O'Dunlaing [8] improve this to $O(\log^2 n)$ time on $O(n/\log n)$ processors using a more sophisticated merge procedure.

The algorithm of Reif and Sen [22] uses random sampling to create many independent sub-problems. The algorithm uses a recursive sampling scheme to guarantee its worst case performance. This results in an optimal algorithm, running in $O(\log n)$ time on n processors. Ghouse and Goodrich also use probabilistic splitting in their algorithm [13] This algorithm is even more complicated than Reif and Sen's, since it uses some sophisticated techniques to decrease the probability of failure, and also uses Reif and Sen's algorithm as a subroutine.

The trend in theoretical studies of this problem is toward complicated algorithms that are asymptotically "efficient" in the PRAM model. But, such algorithms tend to be difficult to implemented and large hidden constants keep them from being efficient in practice.

My approach will be to solve the all-nearest-neighbors problem by adapting the spiral search idea to a data parallel programming model. While the resulting algorithm is not provably fast in the worst case, and its performance does depend on the distribution of the input, it is easy to implement, and fast in practice.

2. Parallel Algorithms and the Cray

My computational model is based on Blelloch's vector model [5]. This model is based on a large set of vector primitives that fall into three classes of primitive operations: elementwise arithmetic, routing, and scans. My algorithms are designed with these operations in mind, but to make them more concrete I will describe and analyze them in terms of a particular architecture, the Cray Y-MP. In general though, the primitives execute efficiently on a wide variety of architectures such as the Connection Machine, the MasPar MP-1 and the Intel Touchstone [6].

The Cray Y-MP is a vector multiprocessor. Each "head" of the Y-MP is a large, pipelined CPU. Each CPU can execute two kinds of instructions. Scalar instructions correspond to the instruction set of a conventional architecture while vector instructions take maximum advantage of the CPU's pipelined functional units. There are functional units for floating point operations, logical operations and memory operations. The functional units take input from, and write results to, one of eight vector registers, each of which can hold 64 words of data. After a fixed startup cost, the arithmetic and logical functional units return one result per clock tick.

The performance of the memory unit depends on the access pattern of a particular instruction. The Cray memory system is divided into 256 banks. The banks are grouped in four sections each containing eight subsystems that hold eight banks each. Thus, each section has a total of 64 banks of memory. The memory system is interleaved so that consecutive words of memory live in different banks. Thus, as long as an access pattern avoids even strides (and in particular, strides that are multiples of 32), the memory system can deliver one word per clock tick after an initial startup time. In addition, the memory unit supports general routing using scatter/gather instructions. Again, these instructions will be efficient except when the access pattern causes bank conflicts.

We can vectorize each of the parallel instructions in Blelloch's vector model. We consider each vector register element to be one "virtual" processor. A full Cray Y-MP has eight CPUs with vector registers that are 64 elements long. This gives us 512 virtual processors to work with. The machine I used only has four CPUs, for a total of 256 virtual processors. Local arithmetic instructions will vectorize trivially. We can handle routing using scatter/gather instructions. Finally, scans vectorize well if we use the clever algorithm of Blelloch, Chatterjee and Zagha [26].

The pseudocode that I use to describe the algorithm is a mix of conventional code, vectorizable loops and vector primitives. In general, vectorizable loops are marked with `foreach`. The code in these loops can be translated into Blelloch's vector language, but I find the format I have chosen to be more understandable. When appropriate, I make use of some of

Blelloch's vector primitives. In particular, `scan-plus` stands for a prefix-sum computation, and `pack` moves the flagged items in a source array into the front of a destination array and returns the number of items in the destination array [5]. I will use the following symbols to represent various constants in my analysis:

A	=	Cost of arithmetic.
R	=	Cost of routing.
S	=	Cost of scans.
P	=	Number of CPUs available.
L	=	Number of virtual processors available.

For my analysis, I calculate the number of clock ticks needed to process each query point. For large vectors, we then have that $A = 1$, $R \approx 1.8$ (for random permutations) and $S = 2.5$. At this time, my algorithm does not try to take advantage of multiple CPUs, so $P = 1$. However, since the algorithm is almost fully vectorizable, it is likely that it could take advantage of multiple CPUs using the automatic parallelization facilities in the Cray compilers ("multitasking").

3. The Algorithm

Recall that we are given a set S of N points, uniformly distributed in the unit square. For each $s \in S$, we wish to find the point p in $S - \{s\}$ which is closest to s under the Euclidean metric. This corresponds to the original definition of the problem, but with $Q = S$.

My algorithm starts by bucketing the points in S into a uniform grid of $\sqrt{N/c}$ by $\sqrt{N/c}$ cells. We call the parameter c the *cell density*. After the bucketing phase, the expected number of points that fall into a cell will be c. We can implement this step in a simple loop (see figure 1).

We can implement a data parallel version of this loop as long as we first histogram the keys so we know how many list nodes to allocate (see figure 2). In the parallel procedure, the array `counter` keeps track of how many sites have fallen into each bucket. The scan operation computes the first element of `nodes` that falls into each bucket. The second loop then fills `nodes` with pointers to the sites that belong to each bucket. At the end of the second loop, `buckets` will contain the index of the first site in each bucket and `counter` will contain the number of sites in each bucket. Finally, the the points contained in some bucket b are stored in $nodes[buckets[b]]$ to $nodes[buckets[b] + counter[b] - 1]$ (see figure 3).

The main problem with implementing the code in figure 2 is dealing with write collisions on elements of the `counter` array. On the Cray, we can detect such collisions by having each virtual processor first write its processor number into a test array and read it back (see figure 4). We can then mark virtual processors that did not read back what they wrote, and handle those processors sequentially. For uniformly distributed sites, we don't expect too many collisions, so this will not be much of a serial bottleneck.

By carefully counting the vector operations within each `foreach` loop, we can see that the cost of the code in figure 2 is:

$$(S + 14A + 6R)N + 2T_C.$$

Here T_C is the average cost of handling collisions that occur in the main loop of figure 4. For each collision we must do some arithmetic and one memory write which will take roughly 30 cycles on the Cray, so the expression above becomes:

$$28N + 60C \text{ cycles,}$$

where C is the total number of collisions. As long as C is small compared to N, this routine will run in 28 cycles per site.

Using the "birthday paradox," we can see that the expected number of collisions in each group of L sites will be $L(L-1)/2N$ [9]. On average, the total number of collisions will be

$$C = \frac{L(L-1)}{2N}\frac{N}{L} = L - 1.$$

We conclude that as long as L is small compared to N, our bucketing routine will perform well when the sites are uniformly distributed.

Once we have bucketed the points, we can use Bentley, Weide and Yao's spiral search technique to do nearest neighbor searching (see figure 5). For each query point, q, the spiral search is divided into two phases. In the first phase, we find the bucket that the point lies in and search in an outward spiral until we find a non-empty bucket. We then calculate the distance D between the query point and some point in this bucket. In the second phase, we compute the nearest neighbor of the query point by examining every point that lies within a bucket intersecting a circle of radius D centered at q. In order to make the algorithm simpler, I check any bucket that intersects a square of side $2D$ centered at q. Thus, the algorithm searches D layers of buckets centered at the bucket that q falls in. If q lies in bucket (i, j), then the k^{th} *layer* of buckets away from q is defined as all buckets (l, m) such that either $l = i \pm k$ and $j - k \le m \le j + k$, or $m = j \pm k$ and $i - k \le l \le i + k$.

Bentley, Weide and Yao give an asymptotic analysis of the spiral search algorithm which proves that it will run in $O(N)$ time on average. Below, I will present the vectorized algorithm in detail, and provide the constants hidden by the asymptotic analysis.

The main obstacle to vectorizing this algorithm is that some searches may take much longer than others, so we will have to be careful about load balancing to keep all the virtual processors busy. A straightforward way to do this is to create a virtual "task" for each query point. This task isn't a process in the operating systems sense since it only keeps the state necessary to perform spiral searching. To do the search, the algorithm has each task do one search step, and then check to see if it is finished. It then uses a call to `pack` to delete any finished tasks, and iterates this process until all the tasks have finished. This scheme uses a small amount of overhead to ensure that no search task does any extra work. The parallel algorithm will do no more work than the original sequential algorithm, so we can conclude that the parallel algorithm will do $O(N)$ operations asymptotically.

The load balancing scheme is complicated by the fact that the spiral search loop has a triply nested structure. The outermost loop iterates over the layers of buckets that the spiral search is examining. The second level loop iterates over buckets that lie in each layer, and the inner-most loop iterates over points that lie within each bucket. It is the inner-most loop that does the actual distance computations. I will analyze the cost of each of these loops below.

First, each search task has an outer loop that walks over layers of buckets. The algorithm must be careful to pack away finished tasks whenever it moves to a new layer (see figure 6).

Each time the algorithm moves to a new layer, each task checks to see if it is done and if not, it pre-calculates some information for the bucket loop. The cost of these calculations is about $13A$.

In general, a task survives to layer k only if the first point found by the initial stage of the spiral search algorithm was more than $k-1$ layers away from the query point. This means that $k-1$ layers of buckets must be empty. Since the probability that a given bucket is empty is $(1-c/n)^n < e^{-c}$, the probability that $k-1$ layers of buckets are empty is $O(e^{-k^2})$ [3]. Thus, the algorithm packs away at least one half of the current tasks each time it moves to a new layer, so the total expected time needed for this bookkeeping is $2N(13A+2S+2R)$ or about 44 cycles per point.

The next loop level walks over the buckets within a layer (see figure 7). Each time the algorithm moves to a new bucket, each virtual processor updates its bucket index, does a boundary check, and then participates in a call to `pack`. This takes $3(A+R)+2S$ time. When $c=1$, the algorithm does one distance calculation per bucket, so the expected loop overhead for walking over the buckets is $DN(3A+3R+2S)$ time, where D is the average number of distance calculations that each point performs. I will calculate a value for D below.

The inner loop of the algorithm does a distance calculation between the query point and each member of each bucket (figure 8). Again, after each distance computation, the algorithm looks for tasks which have finished with their bucket, and packs them away.

This phase is trickier to analyze because working out the exact number of distance computations needed for each query point is a tedious exercise. Bentley, Weide and Yao show that this number takes the form of a sum:

$$D = \sum_i e^{-c(i-1)} O(i),$$

where i is the number of cells searched in the first stage. Since the coefficients are exponentially decreasing, it suffices to consider just the first few terms of this sum to work out an accurate approximation. If the first cell is not empty, then we must search the first cell and the eight cells around it. If the first cell is empty, but one of the next eight is not, then we must search the 25 cells in the first two layers around the point. If we cut the sum off here, we are ignoring terms with coefficients that are less than e^{-8}. These terms will not make a significant contribution to the total. The expected number of distance computations will then be

$$D = 9c(1-e^{-c}) + \sum_{1 \le k \le 7} 25ce^{-ck}.$$

For $c=1$, this works out to be about 17 distance calculations per site. For each distance calculation we must increment and check an index, load the appropriate site, perform an inner product calculation and update the running minimum. This takes

$$(2A+2R)+4R+3A+(2A+3R) = 8A+9R.$$

When we add in the overhead for load balancing, the total expected time works out to be:

$$17(2S+11R+8A)N.$$

On the Cray, this will be about 595 cycles per site, or 3.5μs per site. Using this value of D, we can conclude that the outer loop of the algorithm uses about 238 cycles per point.

Putting everything together, we have that the whole algorithm will use about 900 cycles, or 5.4μs per point, on average, to solve the all-nearest-neighbor problem.

4. Measurements

I have implemented the above algorithm in C, mixed with calls to Blelloch, Chatterjee, and Zagha's [7] assembly language routines for `pack` and `scan-plus`. The code also runs on my Sun workstation, which proves to be an invaluable aid for modifying and debugging off-line from the Cray.

In addition, I have implemented a C version of Bentley, Wiede and Yao's original algorithm. By comparing the vector algorithm to the sequential, I obtain a more realistic idea of how efficient the vector algorithm is.

Since the algorithm spends most of its time doing distance computations, the first logical parameter to check is the number of distance calculations that each algorithm performs. Figure 9 shows the number of distance computations per point that each of the implementations performed as a function of the size of the problem. The graph shows a summary of several runs at each inputs size. The dots in the graph represent the median values of the trials, while the vertical lines connect the first and third quartile values to the minimum and maximum values respectively.

Figure 9 shows that the average number of distance calculations performed per point is bounded by a constant near 15. More importantly, my analysis predicted a value for this constant that is within 20% of the actual value. The sequential algorithm performed fewer calculations because it never computes the distance between a point and itself, while the vectorized algorithm avoids the extra conditional check at the expense of a few extra computations. On the Cray, this is a good tradeoff since conditionals are expensive inside vectorized loops.

Figure 10 shows that runtime of the scalar algorithm on a fast workstation. The timings were taken on a Sparcstation 2 with 64MB of memory, using `gcc -O2`. The graph shows that the average run time of the algorithm tops out at about 85μs per point.

Figure 11 shows the performance of both the scalar and vector algorithms on the Cray Y-MP. The code was compiled using the Cray standard C compiler using the highest available scalar and vector optimization levels, but no multitasking. The Cray run times are given in units of 6ns clock cycles. Thus, figure 11a shows that the scalar code on the Cray uses about 3690 cycles, or 22.1μs per point.

Figure 11b shows the performance of the vectorized algorithm on one processor of the Cray Y-MP. Least squares regression indicates that the asymptotic run time of the algorithm is about 773 cycles per point, or 4.6μs per point. This is about 4.77 times faster than the scalar time on the Cray, and 18.91 times faster than the run time on the workstation. Finally, for a sense of history, this run time is 600 times faster than the time originally reported by Bentley, Weide and Yao for this algorithm when implemented on a PDP-10 [3].

The actual run time of the algorithm is about 15% better than the analysis predicted. This discrepancy occurs because in order to make my analysis simpler, I assumed that a call to `pack` would cost one scan and one routing operation, or about 4 cycles per element. In fact, the assembly language routine for `pack` overlaps the scan with the routing operation and only costs about 2.5 cycles per element. If we re-calculate the cost of the algorithm with this new value for calls to `pack`, we get about 800 cycles per point, which is within 5% of the actual run time for large problems.

My simple analysis also does not account for some programming techniques such as loop unrolling and chaining which can improve the performance of the vector units. These techniques are used by the Cray C compiler to speed up some simple loops by a large amount, making parts of the resulting program somewhat faster than my analysis predicted.

Finally, the Cray C compiler didn't really vectorize all of the bucketing code. It could not fully vectorize the last loop, even though it is clear the loop can be vectorized using the collision resolution technique shown in figure 4. This makes the bucketing routine run somewhat slower than the analysis predicted, but it is a small effect since this routine is not a large percentage of the total run time.

4.1. Discussion

The performance of my implementation is encouraging. It shows that my analysis was accurate, which means that the computational model was accurate. This is satisfying because the model is not tied to the Cray architecture, so by reassigning the costs of the primitives, we can easily redesign the algorithm for other machines. For example, to move the algorithm to an architecture like the MasPar MP-1, we might note that doing global packs would slow the algorithm down, since global routing on that architecture is slow. But, we can achieve a similar effect using local indirect addressing and occasional calls to a global pack, so the algorithm will still show good performance.

The critical assumption that is necessary for my algorithm to peform well is that the size of the input is much larger than the amount of parallelism available in a given machine. This is critical to keeping all processors busy, and for guaranteeing good load balance. Because of this, the algorithm needs large per-processor memories to hold problems big enough to be effective. This is not really much of a problem, since the current generation of parallel architectures all have per-processor memories that are large enough to handle big problems. In fact, I have just started to investigate the performance of this algorithm on the MasPar MP-1. Early results are encouraging.

My experience also shows the importance of designing algorithms that are practical enough to be implemented. Being able to do the implementation pointed out many details that my original analysis had missed. Also, being able to collect performance data from the running program can provide insight that is not available through mathematics alone. This kind of evaluation and analysis is not possible for many theoretical algorithms that are too complicated to implement on current architectures. My original analysis missed two aspects of the algorithm which became apparent only after writing the code. First, managing the nested loops in the algorithm was much more complex than I originally predicted. Until writing the code, I had only considered the load balancing scheme in singly nested loops so early analysis of the algorithm ignored much of the cost involved in dealing with many levels of loop nesting.

Second, my original analysis of the algorithm did not fully account for the indirection that is present in my data structures. The result of this was an overly optimistic estimate for the performance of the inner loop. After writing the program, I updated my analysis and found that I had misjudged the cost of the inner loop by almost a factor of two.

The performance of the algorithm is also encouraging when evaluated with various other statistics. It is almost 100% vectorizable, it generates very few bank conlicts, and thus makes nearly optimal use of the Cray memory system, and the vectorized C code runs more than 60

times faster than the same code running on a workstation. But, the most important performance metric for a parallel algorithm is how well it compares to an optimal sequential algorithm. The performance of my algorithm by this metric is good, but could be better. My analysis, and experimental data provided by the Cray performance measurement tools show that the inner loop is memory bound, doing nine indirect memory operations in order to perform eight arithmetic operations. Thus, the algorithm uses too much memory bandwidth for each unit of "useful" work. The key to speeding up the algorithm even more is to, if possible, simplify the data structure so that it uses less indirection, or to simplify the structure of the parallel loops to remove as much overhead as possible.

5. Extensions and Applications

Although the current algorithm performs well on uniformly distributed point sets, its performance on non-uniform sets is suspect. Bentley, Weide and Yao show that their sequential implementation still performs well even when the points are non-uniform. To check their results, and examine the effect of non-uniformity on the vectorized algorithm, I ran another set of trials with clustered inputs. The clusters were generated using the formulas $x = p_x^i + o_x$ and $y = p_y^i + o_y$ where o_x and o_y are normally distributed and p_x^i and p_y^i are chosen at random for $i = 1, 2, \cdots 10$. I set the scale on the normal to .1 to model mild clustering.

Figure 12 shows the results of these experiments. These figures indicate that mild clustering slows the algorithm down by a factor of two, which is consistent with the earlier results reported by Bentley [3].

For point sets that are extremely non-uniform, we can add a pre-processing phase to the algorithm that attempts to adapt the bucket grid to the distribution of the input. This phase would use a small sample of the input to define a non-uniform bucket grid, and then would further subdivide these buckets in a uniform way. If the sample is a good predictor for the real distribution of the points, then the distribution of points within the new bucket structure will be smooth. Weide shows how to use this technique in the context of conventional algorithms [3] and many parallel algorithms have used random sampling in a similar way [4,13,23].

The cell data structure can also be used to solve many other basic computational problems. First, we can use it to design an algorithm for constructing the Voronoi diagram of a planar point set [2]. In practice, it is easier to consider algorithms for constructing the dual of the Voronoi diagram, the Delaunay triangulation. Many simple algorithms exist to solve this problem on conventional machines, while relatively few exist for vector and parallel machines [11,12,14,15,19,24]. The algorithms of Dwyer [11] and Maus [19] both use buckets to speed up a gift-wrapping like scheme for constructing the triangulation. On uniformly distributed point sets, both of these algorithms run in linear expected time. These algorithms add valid edges to the diagram incrementally using a search scheme that is similar to spiral search. It is not difficult to imagine an algorithm that performs many of these searches in parallel. Given the performance of the parallel spiral search algorithm, I expect that adapting these schemes will also lead to efficient parallel algorithms.

We can also use the cell technique to solve problems that are not related to proximity. For example, the bucketing algorithm in figure 2 could be used as the first pass for a bucket sort algorithm. In his thesis, Weide presents experimental results that show that when combined with the simple sampling strategy outlined above, bucket sort outperformed quicksort on a

PDP-10 [25]. Using the same technique on a vector multiprocessor could result in an algorithm that is potentially faster than Blelloch and Zagha's radix sort algorithm [26] on some classes of input sets.

6. Conclusions

I have designed a simple parallel algorithm for solving the all-nearest-neighbor problem that is efficient on vector architectures. The algorithm shows that cell-based search structures lead to simple and practical parallel algorithms. These algorithms should also translate well to other architectures that can support a data parallel style of programming.

Further research will evaluate the usefulness of these techniques in solving many other problems, including sorting, and constructing Delaunay triangulations. I am optimistic that this work will result in efficient algorithms for these problems on a wide range of architectures.

7. Acknowledgements

I'd like to thank the North Carolina Supercomputer Center for letting me use their Cray Y-MP. I also must thank my two advisors, Scot Drysdale and John Reif, who have never lost faith in the fact that I would eventually stumble on something good after spending a long time watching me look. Finally, Mike Landis and Owen Astrachan read early drafts of the text and provided many helpful comments.

8. References

1. A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. O'DUNLAING, AND C. YAP, "Parallel Computational Geometry," *Algorithmica*, 3, pp. 293-327, New York (1988).
2. F. AURENHAMMER, "Voronoi Diagrams-A Survey of a Fundemantal Geometric Data Structure," *Computing Surveys*, 23, 5, pp. 345-405 (1991).
3. J. L. BENTLEY, B. W. WEIDE, AND A. C. YAO, "Optimal Expected Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software*, 6, 4, pp. 563-580 (1980).
4. G. BLELLOCH, C. LEISERSON, B. MAGGS, G. PLAXTON, S. SMITH, AND M. ZAGHA, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," *Annual ACM Symposium on Parallel Algorithms and Architectures* (1991).
5. G. BLELLOCH, *Vector Models for Data-Parallel Computing*, MIT Press (1990).
6. G. BLELLOCH AND S. CHATTERJEE, "VCODE: A Data-Parallel Intermediate Language," *Frontiers of Massively Parallel Programming*, pp. 471-480 (1990).
7. S. CHATTERJEE, G. BLELLOCH, AND M. ZAGHA, "Scan Primitives for Vector Computers," *Proceedings of Supercomputing '90*, pp. 666-675 (1990).
8. R. COLE, M.T. GOODRICH, AND C. O'DUNLAING, "Merging Free trees in Parallel for Efficient Voronoi Diagram Construction," *LNCS 443*, pp. 432-445 (1990).
9. T. CORMEN, C. LEISERSON, AND R. RIVEST, *Introduction to Algorithms*, MIT Press/McGraw Hill (1990).
10. R. A. DWYER, *Average-case Analysis of Algorithms for Convex Hulls and Voronoi Diagrams*, Ph.D. Thesis, CMU, Pittsburgh, PA (1988).

11. R. A. DWYER, "A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations," *Algorithmica*, 2, pp. 137-151 (1987).
12. S. FORTUNE, "A Sweepline Algorithm for Voronoi Diagrams," *Algorithmica*, 2, pp. 153-174 (1987).
13. M. GHOUSE AND M. GOODRICH, "In-Place Techniques for Parallel Convex Hull Algorithms," *Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 192-201 (1991).
14. L. GUIBAS AND J. STOLFI, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Transactions on Graphics*, 4, 2, pp. 75-123 (1985).
15. L. GUIBAS, D. KNUTH, AND M. SHARIR, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams," *ICALP*, pp. 414-431 (1990).
16. J. KATAJAINEN AND M. KOPPINEN, *Constructing Delaunay Triangulations by Merging Buckets in Quad-tree Order*, p. Unpublished manuscript (1987).
17. C. LEVCOPOULOS, J. KATAJAINEN, AND A. LINGAS, "An Optimal Expected-Time Parallel Algorithm for Voronoi Diagrams," *SWAT*, pp. 190-198 (1988).
18. P. D. MACKENZIE AND Q. STOUT, "Ultra-Fast Expected Time Parallel Algorithms," *ACM-SIAM Symposium On Discrete Algorithms*, pp. 414-423 (1990).
19. A. MAUS, "Delaunay Triangulation and the Convex Hull of n points in expected linear time," *BIT*, 24, pp. 151-163 (1984).
20. F. PREPERATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, New York (1985).
21. J. PRINS, W. HIGHTOWER, AND J. REIF, "Implementations of Randomized Sorting on Large Parallel Machines," *Annual ACM Symposium On Parallel Algorithms And Architectures* (1992).
22. J. REIF AND S. SEN, *Optimal Parallel Randomized Algorithms for 3-D Convex Hulls and Related Problems*, Tech Report, Duke University (1990).
23. J. REIF AND S. SEN, "Polling: A New Randomized Sampling Technique for Computational Geometry," *Symposium on Theory of Computation*, 21 (1989).
24. M. SHARIR AND E. YANIV, "Randomized Incremental Construction of Delaunay Diagrams: Theory and Practice," *Annual ACM Symposium on Computational Geometry* (1991). Submitted.
25. B. WEIDE, *Statistical Methods for the Analysis of Algorithms*, Ph.D. Thesis, CMU (1978).
26. M. ZAGHA AND G. E. BLELLOCH, "Radix Sort for Vector Multiprocessors," *Proceedings Supercomputing'91*, pp. 712-721, IEEE, Albuquerque, New Mexico (November 1991).

Figures

```
% sites[0..N-1] = array of sites
% nodes[0..N-1] = array of list nodes
% buckets[0..N/c-1] = array of buckets

for i := 0 to N-1 do
begin
  bindex := bucket index of sites[i];
  node := newnode();
  nodes[node].site := i;
  nodes[node].next := buckets[bindex];
  buckets[bindex] := node;
end
```

Figure 1: Sequential loop to bucket points.

```
% sites[0..N-1] = array of sites
% nodes[0..N-1] = array of list nodes
% bindex[0..N-1] = bucket index
% buckets[0..N/c-1] = array of buckets
% counter[0..N/c-1] = histogram

foreach i = 0 to N-1
begin
  bindex[i] := index of sites[i];
  counter[bindex[i]]++
end

buckets := counter := scan-plus(counter)

% The following loop assumes that write
% collisions on the counter array are taken
% care of so all processors get a correct index
% into nodes.

foreach i = 0 to N-1
begin
  nodes[counter[bindex[i]++] := i
end
```

Figure 2: Parallel loop to bucket points.

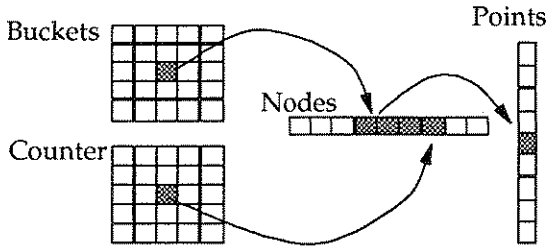

Figure 3: The bucket data structure.

```
% dest[bindex[i]] := dest[bindex[i]]+1
% L virtual processors numbered from 0 to L-1

foreach i=0 to L-1 pnum[i] := i

for offset := 0 to N-1 begin
  foreach i = 0 to L-1 begin
    tindex[i] := bindex[i+offset]
    dest[tindex[i]] := pnum[i]
    test[i] := (dest[tindex[i]] = pnum[i])
  end
  foreach i = 0 to L-1 begin
    dest[tindex[i]] := dest[tindex[i]]+1
  end
  if some element of test is zero begin
    for i := 0 to  L-1 begin
      if (not test[i]) begin
        dest[tindex[i]] := dest[tindex[i]]+1
      end
    end
   end
  offset := offset + L
end
```

Figure 4: Handling write collisions.

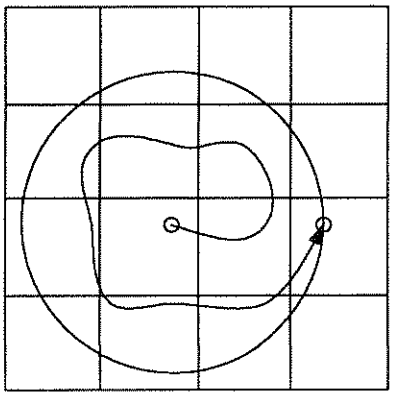

Figure 5: Spiral search to find a near neighbor.

```
layer := 0
numtasks := N
while (numtasks > 0)
  foreach i = 0 to numtasks-1 begin
    tasknum[i] := i
    <Prepare for this layer>
  end
  doOneLayer(layer, numtasks, tasknum)
  foreach i = 0 to numtasks-1 begin
    done[i] := <Task is finished>
  end
  numtasks := pack(tasknum, tasknum, not done)
end
```

Figure 6: The outer loop of the spiral search algorithm.

```
doOneLayer(layer, numtasks, tasknum)
  for each bucket in this layer
    foreach i = 0 to numtasks-1
      temp[i] := tasknum[i]
      bindex[i] := bucket(tasknum[i])
      done[i] := counter[bindex[i]] = 0
    end
    pack(bindex, bindex, not done)
    pack(temp, temp, not done)
    doOneBucket(numtasks, temp, bindex)
  end
end
```

Figure 7: The second outer loop of the spiral search algorithm.

```
doOneBucket(numtasks, tasknum, bindex)
  bi := 0
  while (numtasks > 0)
    foreach i = 0 to numtasks-1
    Check nodes[buckets[bindex[i]]]+bi]
    done[i] := (bi > counter[bindex[i]]
  end
  pack(bindex, bindex, not done)
  numtasks := pack(tasknum, tasknum, not done)
  bi++
end
```

Figure 8: The inner loop of the spiral search algorithm.

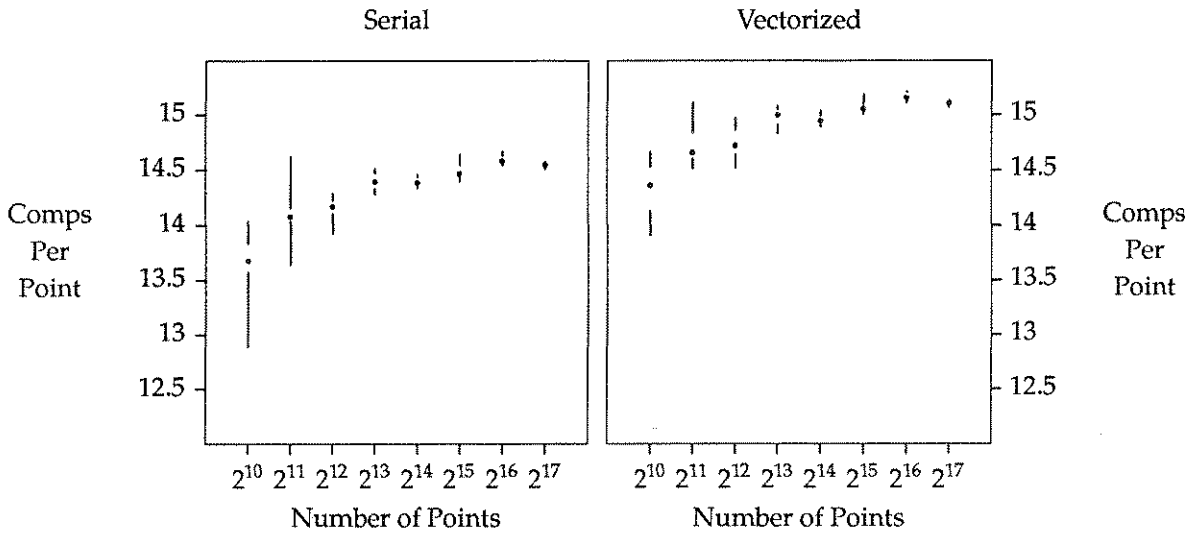

Figure 9: Distance computations per site for each algorithm.

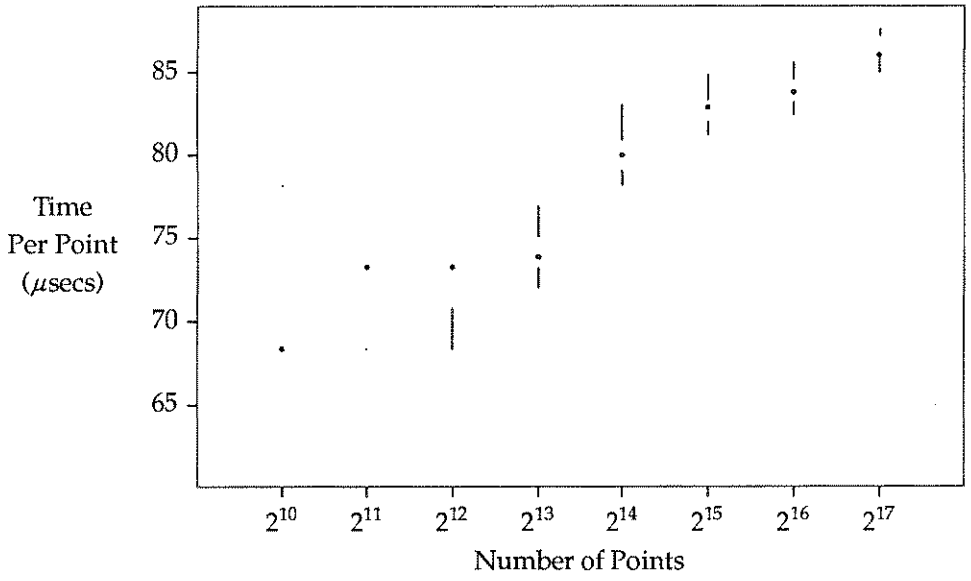

Figure 10: Time per point for the spiral search algorithm on a Sparcstation 2 (microseconds).

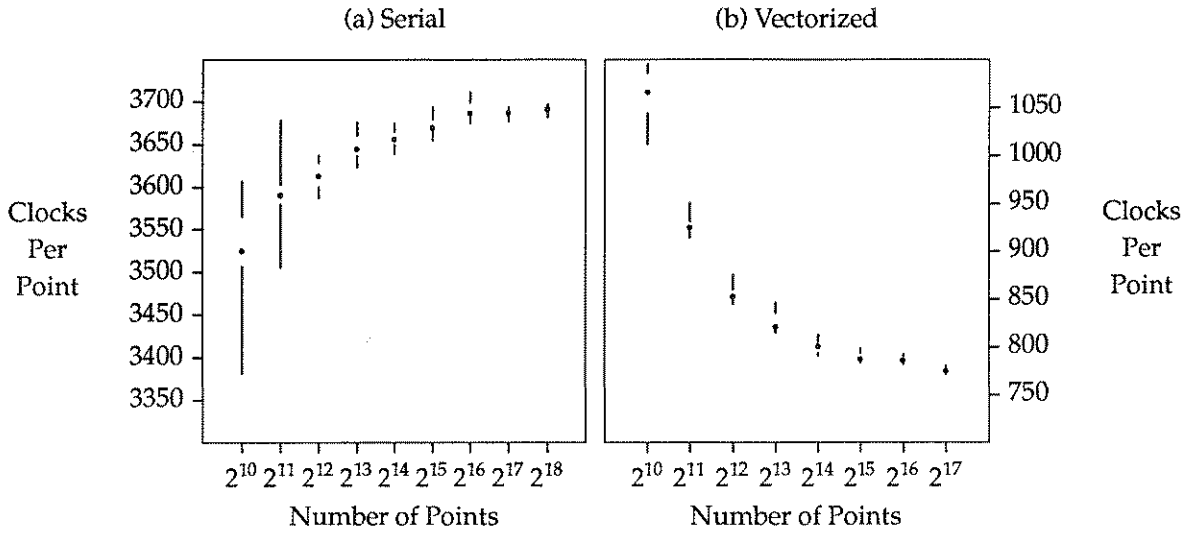

Figure 11: Time per point for the spiral search algorithm on the Cray (6ns clocks).

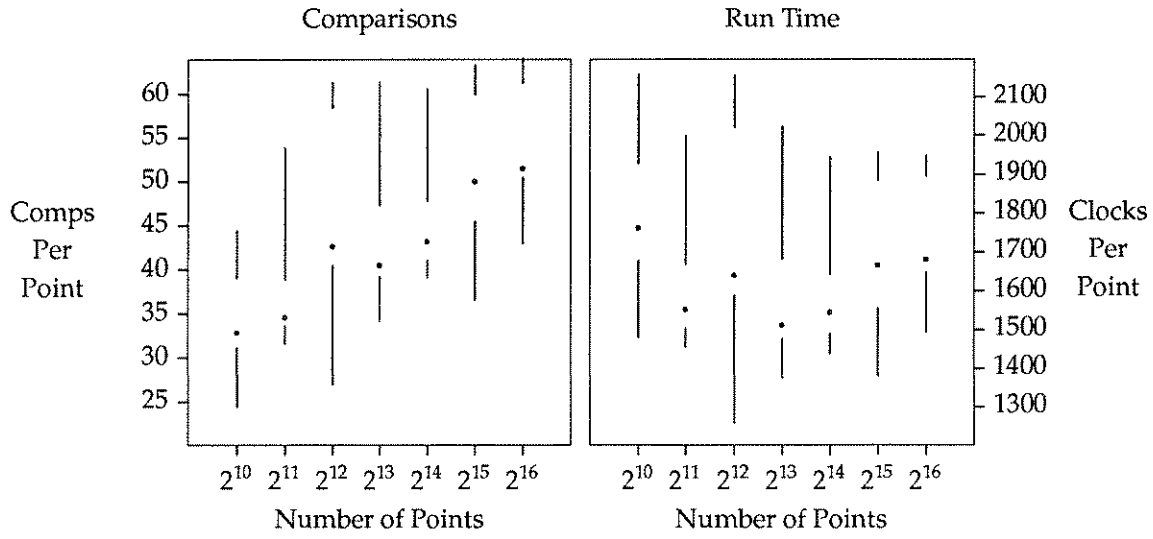

Figure 12: Performance of the vectorized algorithm on clustered inputs.