Efficient Parallel Algorithms for Geometric Clustering and Partitioning Problems*

Amitava Datta †

Abstract

We present efficient parallel algorithms for some geometric clustering and partitioning problems. Our algorithms run in the CREW PRAM model of parallel computation. Given a point set P of n points in two dimensions, the clustering problems are to find a k-point subset such that some measure for this subset is minimized. We consider the problems of finding a k-point subset with minimum L_∞ perimeter and minimum L_∞ diameter. For the L_∞ perimeter case, our algorithm runs in $O(\log^2 n)$ time and $O(n \log^2 n + nk^2 \log^2 k)$ work. For the L_∞ diameter case, our algorithm runs in $O(\log^2 n + \log^2 k \log\log k \log^* k)$ time and $O(n \log^2 n)$ work. We consider partitioning problems of the following nature. Given a planar point set S ($|S| = n$), a measure μ acting on S and a pair of values μ_1 and μ_2, does there exist a bipartition $S = S_1 \cup S_2$ such that $\mu(S_1) \leq \mu_i$ for $i = 1, 2$? We consider several measures like diameter under L_∞ and L_1 metric; area, perimeter of the smallest enclosing axes-parallel rectangle and the side length of the smallest enclosing axes-parallel square. All our parallel algorithms for partitioning problems run in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM. In our algorithms, we use an optimal parallel construction of a range tree. We also show how to perform report mode orthogonal range queries in optimal $O(\log n)$ time using $O(1 + k/\log n)$ processors, where k is the number of points inside the query range. The processor-time product for this range reporting algorithm is optimal.

*Part of this work was done when the author was with Max Planck Institut für Informatik and FernUniversität Hagen.

†Institut für Informatik, Universität Freiburg, Rhein Straße 10-12, D-79104, Freiburg i.Br., Germany. *e-mail:* `datta@informatik.uni-freiburg.de`

1 Introduction

Clustering and *partitioning* problems are important in *pattern recognition* and *cluster analysis* [1, 20]. These methods are useful for collecting subsets of points which are close with respect to some parameter [1, 20].

In this paper, we consider clustering problems of the following type. Given a point set P with n points, the problem is to compute a subset of k points such that some *closeness measure* is minimized. As an example, we may want to minimize the perimeter of the convex hull of the k point subset. This measure was considered by Dobkin *et al* [13]. Aggarwal *et al* [4] considered closeness measures like diameter, side length of the enclosing square, the perimeter of enclosing rectangle, etc. They gave algorithms for these problems on the plane based on higher order Voronoi diagrams. Eppstein and Erickson [14] gave a general framework for solving k point clustering problems based on computing rectilinear nearest neighbours for the points in the set P. The underlying strategy for their algorithms is that a k point subset satisfying any of the closeness measures must be in the set of nearest neighbours for a particular point. The best known sequential algorithms for many of these problems were presented by Datta *et al* [11]. They improved upon the strategy of Eppstein and Erickson [14] by imposing a grid on the point set and solving the problems for smaller subsets of points.

The applications in pattern recognition and cluster analysis typically involve large sets of data. Also, in many cases, it is important to solve such problems fast for real time applications. Hence, the conventional sequential algorithms may not be fast enough for such problems and it is important to study these problems in the domain of parallel algorithms. Currently, no parallel algorithm is known for any of these problems. In this paper, we initiate the investigation of parallel complexity of these problems. Our model of computation is the *Parallel Random Access Machine (PRAM)* [23]. We are interested in the *Concurrent Read Exclusive Write (CREW)* PRAM. In this model, two or more processors can read from the same memory location simultaneously, but only one processor can write to a memory location at a time. For details of this model, see [23]. In this paper, we give efficient parallel algorithms for two of the k-clustering problems. In the first problem, we compute a minimum L_∞ diameter k

point subset (or the minimum side-length square). In the second problem, we compute the minimum L_∞ perimeter k point subset (or the minimum perimeter rectangle). The best known sequential algorithms for these two problems were given in [11]. The sequential algorithm for the minimum L_∞ diameter k point subset problem runs in $O(n \log n + n \log^2 k)$ time. We present a parallel algorithm which runs in $O(\log^2 n + \log^2 k \log\log k \log^* k)$ time and $O(n \log^2 n)$ work. So, the work done by our algorithm (processor-time product) matches the time complexity of the sequential algorithm in [11], when $k = O(n)$. For small k also, the work done by our algorithm compares favourably with the time complexity of the sequential algorithm. The sequential algorithm for the minimum L_∞ perimeter problem runs in $O(n \log n + nk^2)$ time. Our parallel algorithm for this problem runs in $O(\log^2 n)$ time and $O(n \log^2 n + nk^2 \log^2 k)$ work. So, for this problem also, the work done by our algorithm compares well with the time complexity of the best sequential algorithm and is away by a factor of $O(\log^2 n)$ when $k = O(n)$.

We solve these problems by non-trivial parallelizations of the algorithms in [11]. For computing the minimum L_∞ perimeter k point subset, we use an optimal parallel construction of the well known geometric data structure of *range tree*. This construction takes $O(\log n)$ time and $O(n)$ processors in the EREW PRAM. Since range searching is an important geometric operation, this construction may be useful for solving other geometric problems in parallel.

The partitioning problems considered in this paper involve partitioning a point set into subsets such that each subset satisfies some optimization criteria with respect to some parameter of the point set. More formally, given a set of points S in the d-dimensional space and a measure μ acting on it, a partitioning problem is to partition S into k subsets S_1, S_2,...,S_k, such that $f(\mu(S_1), \mu(S_2), ..., \mu(S_k)) \leq \mu^*$. Here, μ^* and k are part of the inputs and f is some function of the measures of the subsets. The choice of μ and f depends on actual applications. In most cases, these problems are NP-complete for arbitrary k [17, 21, 22]. So, most of the research has been concentrated on the bipartition problem (i.e., when $k = 2$). The problem of minimizing the maximum diameter for a bipartition was first considered by Avis [2]. Later, an optimal $O(n \log n)$ time algorithm was given by Asano *et al* [3]. In [27], an $O(n^2)$ time algorithm was given for minimizing the sum of the two diameters. In [21], these problems were considered from a different point of view. Instead of imposing a single constraint on the maximum or sum of the parameter, they considered the problem of finding a bipartition where each partition satisfies some constraint. Such a formulation is useful for a finer cluster analysis, since the size of the individual clusters can be controlled. In the formulation of Hersberger and Suri [21], the bipartition problem is the following. Given a set of points S, a measure μ and a pair of values μ_1 and μ_2, decide if there exists a bipartition $S = S_1 \cup S_2$ that simultaneously satisfies $\mu(S_1) \leq \mu_1$ and $\mu(S_2) \leq \mu_2$. The measures considered in [21] are the diameter of a set in the L_∞, L_1 and L_2 metric; the area, perimeter and diagonal of the smallest enclosing rectangle with sides parallel to the coordinate axes (isothetic); the side length of the smallest enclosing isothetic square; and the radius of the smallest enclosing circle. They solved the problem for the diameter, square and rectangle measures in $O(n \log n)$ time and that for the circle measure in $O(n^2 \log n)$ time.

Though partitioning problems have been studied for designing sequential algorithms, the parallel complexity of these problems is mostly unknown. In this paper, we present efficient parallel algorithms for some of these problems in the CREW PRAM model of parallel computation. More specifically, we consider bipartitioning problems under different rectangular and square measures and the diameter bipartitioning problem under the L_∞ and L_1 metrics.

Hershberger and Suri [21] presented $O(n \log n)$ solution for all of these problems, when n is the number of points in the set. The solutions for the diameter partitioning problems under the L_∞ and L_1 metrics have been improved to $O(n)$ in [12, 18]. These two solutions are obviously optimal, but for the other problems, no lower bound is known at present. We solve all of these problems in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM. So, the work done by our algorithm is equal to that of the best known sequential algorithm in the case of rectangular and square measures and away by a factor of $O(\log n)$ in case of diameter measure. We recently came to know [19] that the algorithm in [18] can be parallelised with optimal speed up, i.e., $O(\log n)$ time and $O(n/\log n)$ processors in the CREW PRAM. However, our solutions for all of these partitioning problems are based on a unified framework through parallel range searching. It is not clear whether the technique in [18] can be used to design efficient parallel algorithms for the other partitioning problems considered in the present paper.

Our algorithms for solving partitioning problems use a different technique from that in [21] and are based on a variant of parallel range searching. We also use this technique for solving one of the clustering problems. Though range searching is an important tool in computational geometry, it is surprising that not much attention has been paid to the parallel solution of this problem. We briefly review the known results for parallel range searching. Katz and Volper [25] have given an algorithm for retrieving the sum of values in a region on a two dimensional grid in $O(\log n)$ time, using $O(n^{\frac{1}{3}})$ processors. In [29], a parallel algorithm has been presented for range searching in orthogonal domains using a distributed data structure. But it is assumed that a range tree is available and then the search is done on this range tree. No algorithm is given for construction of a range tree. In this paper, we give an optimal parallel construction of the well known data structure of *range tree*. Our construction takes $O(\log n)$ time and $O(n)$ processors in the EREW PRAM model of parallel computation. A single processor can perform a count mode range query on this data structure in $O(\log n)$ time. In a count mode query, given an orthogonal query range, we have to count the number of points inside this query range. If multiple queries are to be serviced simultaneously, we need the CREW PRAM model. We also show how a report mode range query can be performed in a simple way optimally in parallel through our construction of the range tree. In a report mode query, we have to enumerate all the points which fall inside the orthogonal query range. We can perform a report mode query in $O(\log n)$ time, using $O(1+k/\log n)$ processors, where k is the number of points to be reported. The work done by the range reporting algorithm is optimal [28].

The rest of the paper is organized as follows. In section 2, we present the results related to parallel range searching since these methods are used later in the paper. We present our parallel algorithms for clustering problems in sections *3-6*. In sections 3, we discuss the general strategy for solving the geometric clustering problems in parallel. In section 4, we discuss how to construct a degraded δ-grid in parallel. In section 5, we present the algorithm for finding the minimum L_∞ perimeter k point subset. And in section 6, we present a parallel algorithm for computing the minimum L_∞ diameter k point subset. We present our parallel solutions for geometric partitioning problems in sections *7-8*. In section 7, we consider the partitioning problems under different measures like area/perimeter of an enclosing rectangle, the diagonal length of an enclosing square or rectangle and the side length of an enclosing square. In section 8, we consider the diameter bipartitioning problem in L_1 and L_∞ measures. Finally, we conclude with some comments and open problems in section 9. Preliminary versions of portions of this paper have appeared in [9, 10].

2 An optimal parallel algorithm for range searching

In this section, we present an optimal parallel algorithm for constructing a range tree. We also show how to perform count and report mode range queries on this data structure optimally in parallel. We first briefly review the sequential version of the range tree [28]. First the points are sorted according to x coordinate and a skeletal *segment tree* [28] is built. Then for every internal node in this segment tree, the points in its subtree are sorted according to y coordinate. The array of points in the subtree of the node u is represented by $Y[u]$. The points in $Y[u]$ are sorted according to decreasing y coordinates. Consider an internal node u and its left and right children v and w. Consider a point $p_i \in u$. Notice that, p_i will occur either in $Y[v]$ or in $Y[w]$. We assume without loss of generality that p_i is a member of $Y[v]$. A pointer is kept from the occurance of p_i in $Y[u]$ to its occurance in $Y[v]$. Following the notation of [28], we call such a pointer as $LBRIDGE$. A similar pointer is kept to the element immediately above p_i in $Y[w]$. This pointer is called as $RBRIDGE$. Such pointers are maintained for every element in every internal node of the tree T. Finally, all the points in the root are kept in a balanced binary tree sorted according to y coordinates. We call this tree as $T[root]$. For details of this structure, see [28]. Suppose, the query is a rectangle R. We denote the four sides of R as *top*, *bottom*, *left* and *right*. We also denote the height of R as the interval $[top, bottom]$ and the width of R as $[left, right]$.

The range searching is done in the following way. A search is done in the primary segment tree with the interval $[left, right]$. At most $O(\log n)$ nodes are selected during this search. Initially, a search is done with the two coordinates *top* and *bottom* in $T[root]$ to locate the two points p_j and p_k immediately below *top* and above *bottom* respectively. While doing the search with $[left, right]$, we follow the $LBRIDGE$ and $RBRIDGE$ pointers from p_j and p_k to locate the elements in the arrays of the nodes of the segment tree which are selected during the search. Suppose, w is a selected node. We can find in $O(1)$ time, the two points in $Y[w]$ which are just below *top* and above *bottom*. Suppose, these two points have indices m and n $(m > n)$ in $Y[w]$. In the count mode, the difference $(m - n)$ is reported and in the report mode, all the points in between and including these two indices are reported. The net effect is that, we do only one binary search in $T[root]$ and then follow the pointers in constant time during the search. The count mode query can be performed in $O(\log n)$ time and the report mode in $O(\log n + k)$ time, where k is the number of points in the query rectangle R.

2.1 Optimal parallel construction of range tree

In our parallel construction of the range tree, we follow a strategy based on Cole's [6] optimal parallel merge sort algorithm. We first sort the points according to increasing x coordinate and establish them in the leaves of a complete binary tree T from left to right. We assume for simplicity that $n = 2^k$, for some $k > 1$. In the general case, the algorithm can be easily modified by introducing dummy leaves. Now, we build the primary segment tree according to x coordinates by the method of construction of plane sweep tree as described in [7, 23]. This construction takes $O(\log n)$ time and $O(n)$ processors in the CREW PRAM. The next step is to sort the points according to y coordinate. This sorting is done through a merge sort procedure as in [6] and described below. In Cole's algorithm, the merging in an internal node u is done by cross ranking the elements in its two children v and w, where v and w are respectively the left and right child of u. An element $p_i \in v$ is ranked in the array of w through the elements in their parent u. The merging is done through sampling in three phases and once all the elements in the array at u are sorted, the cross ranking pointers are destroyed. Notice that, at the time when all the elements in the array of u are sorted, the cross ranking pointers are exactly the pointers $LBRIDGE$ and $RBRIDGE$ in our above description. Suppose, an element $p_i \in v$ is ranked between two elements $p_j, p_k \in u$ and $y_j > y_k$. If p_j does not appear in $Y[v]$, then the element just below p_j in $Y[v]$ is p_i. So, this is the $LBRIDGE$ pointer in the reverse direction. We can easily keep a pointer in the other direction. So, in the y sorting stage, instead of destroying the cross ranking pointers after a node becomes saturated, we maintain them throughout the construction of the algorithm. See Figure 1 for an example of this pointer structure. At the end, we get a layered structure as in the sequential case. The time and processor requirement is exactly the same as in Cole's algorithm, i.e., $O(\log n)$ and $O(n)$ respectively. This construction can be done in the EREW PRAM. The space requirement is $O(n \log n)$ as we do not destroy the arrays in the internal nodes, but maintain them even after an internal node becomes saturated. If we want to do multiple range searching on this tree in parallel, we need the CREW PRAM model of parallel computation. Notice that, a single processor can answer an orthogonal count-mode range query in $O(\log n)$ time as in the sequential case.

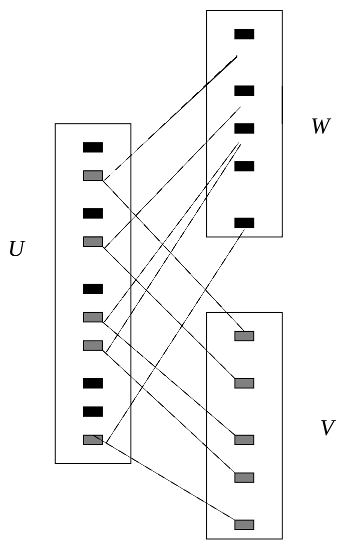

Figure 1: Illustration for cross-ranking pointers when the node *U* is saturated. Only pointers for the elements in *V* are shown. The elements of *Y[V]* are cross-ranked in *Y[W]* through the elements of *Y[U]*.

2.2 Report mode range searching in parallel

In the *count* mode query, we have to report all the points in the set P within the query rectangle R. In this case, we assign one processor to perform the query in $O(\log n)$ time as in the sequential case. But the servicing of a report mode query is considerably different in our case. We describe this in details below. We first mention some details about our processor allocation scheme. Our scheme is similar to that used by Goodrich in [16]. At every step of the algorithm where new processors are to be allocated, our algorithm computes the required number of processors on the fly. Also, the processor allocation is global in the sense that, if a group of new processors start working at step $(i+1)$, at step i, a global array is created with pointers to the tasks to be performed by the new group of processors. As a result, the processor allocation scheme is much cleaner and can be implemented quite easily. For more details of this scheme, see [16].

We first assign one processor to locate the nodes in the primary segment tree which are covered by the query range. We denote these nodes as $u_1, u_2, ..., u_k$, where $k = O(\log n)$. This processor also counts the number of points to be reported in these nodes. We distinguish between two types of nodes, those which contribute number of points larger than $O(\log n)$ in the query range and those which contribute a number less than $O(\log n)$. We call the former as a *large* node and the latter as a *small* node. We also denote the number of points to be reported at a node u_i as $count(u_i)$. The processor assigned initially does the following. If a node u_i is a small node, it writes $count(u_i)$ in a global array $\mathcal{R}$. This is written in the consecutive location of the array $\mathcal{R}$ as the processor traverses the covered nodes in the range tree. Suppose, the number of entries in the array $\mathcal{R}$ is k_1. In other words, there are k_1 small nodes covered by the query range. We assign $k_1/\log k_1$ processors and do a partial sum on the elements of the array $\mathcal{R}$ in $O(\log k_1)$ time [23]. Now, we group the elements of $\mathcal{R}$ in groups of $\log n$ and detect the elements where the partial sums are multiples of $\log n$. This can be done by a single processor assigned to each group. Then, we allocate processors globally at these boundaries (i.e., where the partial sums are multiples of $\log n$) to enumerate the points within the query range. Note that, the boundaries actually correspond to the covered nodes in the segment tree. A particular block of $O(\log n)$ elements may be present in several nodes of the range tree. But it is easy for a single processor to enumerate these elements since the addresses of these nodes are available in the array $\mathcal{R}$. If the total sum of the numbers in the array $\mathcal{R}$ is k_2 (note that, $k_2 \geq k_1$), the enumeration of all the points can be done by $O(k_2/\log n)$ processors in $O(\log n)$ time.

We now discuss the enumeration of points in the *large* nodes. Remember that, the number of points to be reported in any such node is greater than $\log n$. Consider one such node u. We know the two indices i and j such that the points in $Y[u]$ in between these two indices are to be reported. We assume that $|j - i| = k_3$. We assign $O(k_3/\log n)$ processors globally. The m^{th} processor in this group starts enumerating a group of $O(\log n)$ elements from the index $i + m\log n$. If any fragment less than $O(\log n)$ is left at the end, this is enumerated by the last processor in additional $O(\log n)$ time. So, the overall processor and time requirement for the *large* nodes is $O(k_4/\log n)$ and $O(\log n)$ respectively, where the total number of elements to be reported in all the large nodes is k_4. In case of both *small* and *large* nodes, the processor allocation is global and similar to that in [16] as discussed earlier. This concludes our description of the range searching problem. We state the result in the following

Theorem 2.1 *The range tree data structure can be built in $O(\log n)$ time by $O(n)$ processors in the EREW PRAM. The space requirement is $O(n\log n)$. A count mode query can be performed by a single processor in $O(\log n)$ time and the report mode query can be performed in $O(\log n)$ time by $O(1+k/\log n)$ processors, where k is the number of points to be reported. If multiple queries are to be performed simultaneously, concurrent read capability is required. The work done by the algorithm is optimal.*

We also need the following additional information from an orthogonal range search. Given an orthogonal query range R, we want to find out the maximum x coordinate point in this range. This is the well-known range-maxima query. An algorithm is given in [23, pp.131-136] for preprocessing an array A for range-maxima queries. This takes $O(\log n)$ time and $O(n/\log n)$ processors in the CREW PRAM. After this preprocessing, given any two indices i,j in the array A, a single processor can find the maximum element in the subarray $A[i], \ldots, A[j]$ in $O(1)$ time. We preprocess the arrays present at every internal node of our range tree by using this algorithm. Since every element occurs in $O(\log n)$ nodes of the range tree, the overall processor and time requirements are $O(n)$ and $O(\log n)$ respectively. After this, an orthogonal range maximum query is answered in the following way. Given a query range, as before we visit the covered nodes of the primary segment tree. Suppose, at a node n_i, all the elements between the indices j and k fall inside the query range. Instead of counting the number of such elements, we perform a range-maximum query in $O(1)$ time. It is easy to see that the maximum element in the complete query range can be found in $O(\log n)$ time by a single processor.

Lemma 2.2 *A range tree can be preprocessed in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM, such that given an orthogonal query range R, we can find the maximum element with respect to x coordinate inside R in $O(\log n)$ time, using a single processor.*

3 The general strategy for solving clustering problems

Let P be a set of n points on the plane. The points in the set P are represented by $p_1, p_2, \ldots, p_n$. The x (resp. y) coordinate of the point p_i is represented by x_i (resp. y_i). We use some terminology from [11]. Let k be an integer such that $1 \leq k \leq n$. By a *box*, we mean a 2 dimensional axes-parallel rectangle of the form $\Pi_{i=1}^{2}[a_i : b_i)$. If $b_i = a_i + \delta$ for $i = 1, 2$, then we call the box as a δ-box. The closure of a box, i.e., the product of 2 closed intervals $[a_i : b_i]$, $i = 1, 2$ is called a *closed box*.

- μ denotes a function which maps a set V of points in 2-space to a real number $\mu(V)$.
- $S(P, k)$ denotes the problem of finding a subset of P of size k whose measure is minimal among all k-point subsets.
- $\mu_{opt}(P)$ denotes the minimal measure.
- P_{opt} denotes a k- point subset of P such that $\mu(P_{opt}) = \mu_{opt}(P)$.

- $\mathcal{A}$ denotes a CREW PRAM algorithm that solves problem $S(P', k)$, where $P' \subset P$ and $|P'| = O(k)$.
- $T(n, k)$ (resp. $W(n, k)$) denotes the time (resp. work) complexity of algorithm $\mathcal{A}$.

For example, if $\mu(B)$ is the L_∞ diameter of set B, then $S(B, k)$ is the problem of finding a subset of size k whose L_∞ diameter is minimum among all k point subsets. We make the following assumptions about the measure μ which we prove later for each of the problems.

Assumption 3.1 *There exists a closed $\mu_{opt}(P)$ box that contains the optimal solution P_{opt}.*

Assumption 3.2 *There exists an integer constant c such that for any $\delta < \mu_{opt}(P)/c$, any closed δ box contains less than k points of P.*

The constant c will depend on the particular problem and we will fix this constant later for each problem. For example, c is 4 for the problem of finding the minimum L_∞ perimeter k point subset. Our algorithms are based on the following lemma

Lemma 3.1 *Let δ be a real number. Assume there exists a closed δ box that contains at least k points of P. Then $\mu_{opt}(P) \leq c\delta$ and there exists a closed $(c\delta)$ box that contains the optimal solution P_{opt}.*

Proof: By Assumption 3.1, we know that the optimal solution is contained in a closed μ_{opt} box. From Assumption 3.2, we know if $c\delta < \mu_{opt}(P)$, any closed δ-box contains less than k points of P. Hence, the optimal solution cannot be found in such a closed δ-box. This proves that $\mu_{opt} \leq c\delta$. □

Our algorithms work in the following way. We reduce problem $S(P, k)$ to $O(n/k)$ subproblems $S(P', k)$ for subsets $P' \subset P$ of size $O(k)$. All the subproblems are solved simultaneously in parallel. Each subproblem is solved by a CREW PRAM algorithm $\mathcal{A}$. Before discussing the decomposition of our problem, we need some definitions.

Definition 3.2 *Let δ be a positive real number, let $\alpha \leq \beta$ be positive integers, and let $\mathcal{R}$ be a collection of δ-boxes such that*

1. *each box in $\mathcal{R}$ contains at least one point of P,*

2. *each point of P is contained in exactly one box of $\mathcal{R}$,*

3. *there is a box in $\mathcal{R}$ that contains at least α points of P,*

4. *each box in $\mathcal{R}$ contains at most β points of P.*

Then $\mathcal{R}$ is called an $(\alpha, \beta; \delta)$-covering of P.

Now, we give the generic algorithm $\mathcal{G}$ for both of our problems.

Theorem 3.3 *The algorithm correctly solves the problem $S(P, k)$. Moreover, there is a constant c' such that the algorithm takes $O(\log^2 n + T(c'k, k))$ time and $O(n \log^2 n + (n/k)W(c'k, k))$ work.*

Proof: By Lemma 3.1, there is a closed $(c\delta)$-box that contains the optimal solution. It is clear that this box must be contained in the $(2c+1)\delta$-box that is centered at some box of the data structure $\mathcal{R}$. The algorithm checks all these $(2c+1)\delta$-boxes. If there are less than k points in such a box, then it does not contain the optimal solution. Hence, the correctness of the algorithm follows.

Each box in $\mathcal{R}$ contains at most $4k$ points. Moreover, the point location queries in Step 2(a) find at most $(2c+1)^2$ boxes of $\mathcal{R}$. Therefore, the set P' in step 2(b) has size at most $(2c+1)^2 4k$.

There are at most $(2c+1)^2(n/k)$ boxes $B \in \mathcal{R}$ that give rise to a subset P' of size at least k. Hence the algorithm $\mathcal{A}$ is applied in parallel to at most $(2c+1)^2(n/k)$ different subsets.

As mentioned already, we will show in section 3 that the real number δ and the covering $\mathcal{R}$ can be computed in $O(\log^2 n)$ time using $O(n)$ processors. Moreover, in $O(\log n)$ time and using $O(n)$ processors, $\mathcal{R}$ can be stored in a data structure such that a point location query can be answered by a single processor in $O(\log n)$ time. In Step 3, we have to compute the maximum of $O(n/k)$ quantities. This can be easily done in $O(\log n)$ time using $O(n)$ processors [23] i.e., $O(n \log n)$ work. Also, μ_{opt} and P_{opt} can be reported within this time and work.

So, the running time of our algorithm is bounded by :

$$O(\log^2 n + T((2c+1)^2 4k, k)) \tag{1}$$

1. Compute a positive real number δ together with a $(k, 4k; \delta)$ -covering $\mathcal{R}$ of P.

   In the next section, we show that such a δ and such a covering $\mathcal{R}$ exist. We compute this covering in $O(\log^2 n)$ time using $O(n)$ processors in the CREW PRAM. We will also store this collection in a data structure of size $O(n)$ such that point location queries can be answered in $O(\log n)$ time by a single processor. We will build this data structure also in $O(\log n)$ time, using $O(n)$ processors.

2. For each box $B \in \mathcal{R}$, do the following two substeps:

   (a) Find all boxes in $\mathcal{R}$ that overlap the $(2c+1)\delta$-box that is centered at B. These boxes are found as follows:

   Let (b_1, b_2) be the lower-left corner of B. Then in the data structure for $\mathcal{R}$, we locate the $(2c+1)^2$ points

   $$(b_1 + \epsilon_1\delta, b_2 + \epsilon_2\delta),$$

   $$\epsilon_i \in \{-c, -c+1, \ldots, c-1, c\}, i = 1, 2.$$

   (b) Let P' be the subset of points of P that are contained in the boxes that are found in the previous step. If $|P'| \geq k$, solve problem $S(P', k)$ using algorithm $\mathcal{A}$.

3. Find the optimal solution out of $O(n/k)$ solutions found in Step **2**. Output μ_{opt} and P_{opt}.

Algorithm 1: The main steps in our generic algorithm $\mathcal{G}$.

and the work done by the algorithm is :

$$O(n \log^2 n + (n/k)(2c+1)^2 W((2c+1)^2 4k, k)) \tag{2}$$

$\square$

4 Parallel algorithm for computing a $(k, 4k; \delta)$ covering

In the previous section, we have seen that we need a real number δ and a $(k, 4k; \delta)$ covering $\mathcal{R}$ for the point set P. We also need a data structure for these boxes that supports point location queries. We first discuss what we mean by a *degraded grid*. Our definition of the degraded grid is similar to that in [11]. Suppose, the value of δ and a δ-box containing at least k points are known. Then we can take a grid containing this box and having mesh size δ. Then our grid $\mathcal{R}$ is the collection of all non-empty grid cells. The problem with this approach is that we need the *floor function* for distributing the points among the grid cells. If we follow this approach, our algorithm will fall outside the *algebraic decision tree* model of computation. Instead of a perfect grid like this, we construct a degraded grid which supports all the required operations without using the floor function.

4.1 Degraded grids

In a standard δ-grid, we divide the plane into slabs of width exactly δ. In such a grid, if we fix a lattice point, all other lattice points are automatically fixed. For example, if $(0, 0)$ is a lattice point, then a slab along the x axis consists of all points that have their x coordinate between $j\delta$ and $(j+1)\delta$ for some integer j. In a degraded δ-grid, the slabs do not start and end at integer multiples of δ. The slabs have width at least δ and the slabs that contains points have width exactly δ. The empty slabs can have arbitrary width. In other words, whereas a δ grid can be fixed by choosing a lattice point independent of the point set it stores, the degraded grid is defined by the point set it stores. We first give a formal definition of a degraded one-dimensional δ-grid.

Definition 4.1 *[11] Let P be a set of n real numbers and let δ be a positive real number. Let $a_1, a_2, \ldots, a_l$ be a sequence of real numbers such that*

1. *for all $i \leq j < l$, $a_{j+1} \geq a_j + \delta$,*
2. *for all $p \in P$, $a_1 \leq p < a_l$,*
3. *for all $1 \leq j < l$, if there is a point $p \in P$ such that $a_j \leq p < a_{j+1}$, then $a_{j+1} = a_j + \delta$.*

The collection of intervals $[a_j, a_{j+1})$, $1 \leq j < l$, is called a one-dimensional degraded δ-grid for P.

4.1.1 Construction of one-dimensional δ-grid in parallel

We briefly discuss the construction of a one-dimensional δ-grid, given a real number δ. The details are omitted in this version. We first sort the elements in P according to increasing x coordinates and store them in an array X. Now, we associate a processor with each point in the sorted array X. The processor associated with point p_i checks whether $|x_{i+1} - x_i| \geq \delta$. If this check succeeds, the processor marks the point p_{i+1} with 1. The point p_{i+1} is the left boundary of a δ-slab. If the check fails, the processor marks the point p_{i+1} with 0. The left-most element in the array X, i.e., x_1 is marked with 1. This step clearly can be done by $O(n)$ processors in $O(1)$ time.

After this, for every point x_i, we find the nearest 1 to its left. This can be done by the algorithm for finding the *all nearest larger value* [5]. This computation takes $O(n/\log n)$ processors and $O(\log n)$ time in the CREW PRAM.

Now, every processor knows the nearest marked element to its left. Suppose, for the point x_m, the nearest marked point to its left is the point x_n. The processor associated with x_m computes the quantity

$$c = x_n + \delta \times \lfloor |x_m - x_n| / \delta \rfloor$$

Then, $c \leq x_m \leq c + \delta$. It is easy to see that c is the left boundary of the δ-slab to which x_m belongs. We can compute the quantity c without using the non-algebraic *floor function* in the following way. We know that the difference of x coordinates between two consecutive elements between x_n and x_m is less than δ. In other words, $0 \leq |x_m - x_n| \leq (m-n)\delta$ or, $0 \leq (|x_m - x_n|/\delta) \leq (m-n)$. So, we can find the integer $\lfloor |x_m - x_n|/\delta \rfloor$ by doing a binary search in the range 0 to $(m-n)$. This step again takes $O(n)$ processors and $O(\log n)$ time.

After this step, every element knows the left boundary of its δ-slab. The elements within a δ-slab can be counted easily since we know the boundaries of the δ-slabs. The difference of indices of two consecutive boundaries is the number of points inside that δ-slab. This step also can be done within $O(\log n)$ time using $O(n)$ processors.

4.1.2 Construction of two-dimensional δ-grid in parallel

A two-dimensional degraded δ-grid can be defined recursively in the following way. We construct a δ-grid for each one-dimensional δ-slab found in Section 4.1.1. We omit a formal definition in this version. The two dimensional δ-grid is constructed in the following way. We sort the points inside every one-dimensional δ-slab according to decreasing y coordinate. This takes overall $O(n)$ processors and $O(\log n)$ time. Now we apply the algorithm for constructing one-dimensional δ-grid within each slab. Again, the processor and time requirements are $O(n)$ and $O(\log n)$ respectively. After this computaion is over, we have the two-dimensional δ-grid as well as, we know the number of points in each occupied cell of the grid, since we know the grid boundaries. We construct the data structure $\mathcal{R}$ for point location in the following way. The boundaries of the delta slabs perpendicular to the x axis are kept in a balanced binary tree $\mathcal{T}$ in sorted order. This tree can be built in $O(\log n)$ time, using $O(n)$ processors. Within each such slab, the boundaries of the δ-slabs perpendicular to the y axis can be kept in balanced binary trees. Any point location query can be performed by first searching in the tree $\mathcal{T}$ for locating the δ-slab perpendicular to the x axis and then searching in the balanced tree for this δ-slab. So, for locating a point within a slab, we need to do two binary searches. This takes $O(\log n)$ time by a single processor.

Lemma 4.2 *Given a real number δ, a two-dimensional degraded δ-grid can be constructed in $O(\log n)$ time, using $O(n)$ processors in the CREW PRAM. Moreover, this grid can be stored in a data structure $\mathcal{R}$ of size $O(n)$, such that a single processor can answer point location queries in $O(\log n)$ time.*

We also need the following Lemma for our algorithms.

Lemma 4.3 *Let $p_i \in P$ and let B be the box for the degraded δ-box for P that contains p_i. Let c be an integer. All the points of P that are within distance $c\delta$ from p_i are contained in B and in the $(2c+1)^2$ boxes (including B) that surround B.*

Lemma 4.3 establishes the validity of Step $2(a)$ in Algorithm 1.

4.2 Parallel construction of a degraded δ-grid with $O(k)$ points per cell

We have to still discuss how to compute a real number δ such that we get a $(k, 4k; \delta)$ covering. In this section, we discuss a method for parallel construction of such a covering. We first use some notations from [11] and prove the existence of such a δ.

Definition 4.4 *Suppose, P is a set of n points on the plane.*

- *Assume δ is a real number and $\mathcal{R}$ is a degraded δ-grid for P. We number the boxes arbitrarily from $1, 2, \ldots, r$, where the number of cells in $\mathcal{R}$ is r. We define n_i to be the number of points of P, that are contained in the ith box in $\mathcal{R}$. Then we denote $M(\mathcal{R}) = max_{1 \leq i \leq r} n_i$.*
- *Let P' be a subset of P of size $4k$ with minimal L_∞ diameter among all $(4k)$-point subsets. Then, δ^* denotes the L_∞-diameter of P'.*

Lemma 4.5 *Using the above notation, the following holds:*

1. *For any $\delta \geq \delta^*$ and any degraded δ-grid $\mathcal{R}$ for P, we have $M(\mathcal{R}) \geq k$.*
2. *For any $\delta \leq \delta^*$ and any degraded δ-grid $\mathcal{R}$ of P, we have $M(\mathcal{R}) \leq 4k$.*

Proof: For proving condition 1, let $\delta \geq \delta^*$ and let $\mathcal{R}$ be a degraded δ-grid for P. The set P' is contained in an axes-parallel square of side-length δ^*. This square overlaps at most 4 boxes of $\mathcal{R}$. Since the size of P' is $4k$, there must be at least one box in $\mathcal{R}$ which contains at least k points of P. This shows that $M(\mathcal{R}) \geq k$.

For proving condition 2, we assume that $\delta \leq \delta^*$ and $\mathcal{R}$ is a degraded δ-grid for P. Assume that $M(\mathcal{R}) > 4k$. Then there is a box in $\mathcal{R}$ which contains more than $4k$ points of P. Since this box has side lengths δ, there are $4k$ points in P which has L_∞ diameter less than δ, this contradicts our definition of δ^*. □

The parallel algorithm searches for a real number δ together with a degraded δ-grid $\mathcal{R}$ of P such that $k \leq M(\mathcal{R}) \leq 4k$. This grid is the $(k, 4k; \delta)$-covering we want for our algorithms. Lemma 4.5 implies that there is at least one such δ for which such a covering exists, namely $\delta = \delta^*$. Notice that, such a δ is contained in all the L_∞ distances between pairs of points in P. Our search for such a δ is based on the sequential algorithm of Johnson and Mizoguchi [24]. A parallel algorithm based on the algorithm in [24] has been already used by Lenhof and Smid [26]. Their algorithm takes $O(\log^2 n \log\log n)$ expected time and $O(n \log n \log\log n)$ work in the *randomized CRCW PRAM* model of parallel computation. We present a much simpler algorithm for computing a $(k, 4k; \delta)$ covering for the set P in the weaker CREW PRAM.

We recall the notion of weighted medians from [24]. Let, $x_1, x_2, \ldots, x_n$ be a sequence of real numbers such that every element x_i has a weight w_i, and w_i is a positive real number. Let $W = \sum_{j=1}^{n} w_i$. Element x_i is called a weighted median if

$$\sum_{j: x_j < x_i} w_j < W/2 \quad \textit{and} \sum_{j: x_j \leq x_i} w_j \geq W/2 \tag{3}$$

Lemma 4.6 *[24] The weighted median of a set of n weighted real numbers can be computed in $O(n)$ time.*

In the following discussion, we will use a simpler algorithm for computing weighted medians which requires $O(n \log n)$ work. As mentioned earlier, we search for the appropriate δ in the set of all L_∞ distances between pairs of points. We do not enumerate all candidate differences since that will require $\Omega(n^2)$ work. We maintain the candidate differences in an implicit way. The algorithm maintains two sorted arrays X and Y which store the x and y coordinates of the points in sorted order. For the jth element in the array X, i.e., $X[j]$, we store an interval $[a_{xj} : b_{xj}]$, where, a_{xj} and b_{xj} are integers, such that $j < a_{xj} \leq b_{xj} + 1 \leq n + 1$. Similarly, with the jth entry in the array Y, i.e., $Y[j]$, we store an interval $[a_{yj} : b_{yj}]$, where a_{yj} and b_{yj} are integers and $j < a_{yj} \leq b_{yj} + 1 \leq n + 1$. Let, j and $j^{'}$ $(j < j^{'})$ be such that $X[j] = p_r$ and $X[j^{'}] = p_s$. Then $|x_s - x_r|$ is a candidate difference iff $a_{xj} \leq j^{'} \leq b_{xj}$.

Hence, the total number of candidate differences is equal to

$$\sum_{m=x,y} \sum_{j=1}^{n-1} (b_{mj} - a_{mj} + 1) \tag{4}$$

The intervals are initialized as follows $[a_{mj} : b_{mj}] = [j + 1, n]$, where $m = x, y$. So, initially this sum is

$$\sum_{m=x,y} \sum_{j=1}^{n-1} (n - j) = 2\binom{n}{2} \tag{5}$$

As in [24], the parallel algorithm performs a sequence of iterations and in each iteration, the summation in Equation (4) is decreased by a constant factor. The algorithm maintains the following invariant.

Invariant: *At each step of the iteration, the value δ^* is contained in the set of candidate differences.*

We describe the main steps of the algorithm in Algorithm 2.

Lemma 4.7 *The algorithm correctly maintains the invariant at every iteration.*

Proof: After the initialization, the total number of candidate differences is equal to $2\binom{n}{2}$ as indicated in Equation (5). Obviously, δ^* is included in this set. So, the invariant holds initially. Consider an iteration and assume that Step $4(b)$ occurs. i.e., $M(R) < k$. Then, from Lemma 4.5, we know that $\delta < \delta^*$. The algorithm removes differences from the set

1. Sort the points according to x and y coordinates and store them in the arrays X and Y. For every entry $X[j]$ (resp. $Y[j]$) $(1 \leq j \leq n)$, initialize the intervals associated with $X[j]$ and $Y[j]$ in the following way. $[a_{mj} : b_{mj}] = [j+1, n]$, where, $m = x, y$. This initialization can be done in $O(1)$ time using $O(n)$ processors.

2. For each $i \leq j < n$, if $a_{xj} \leq b_{xj}$, take the pair $X[\lfloor (a_{xj}+b_{xj})/2 \rfloor]$ and $X[j]$, and take the positive difference of their x coordinate. Give this difference weight $(b_{xj} - a_{xj} + 1)$. Do a similar computation for the elements in the array Y. This computation gives at most $2(n-1)$ weighted differences. This computation can be done in $O(1)$ time using $O(n)$ processors.

3. Let $s_1, s_2, \ldots, s_r$ be the sequences of numbers obtained from Step 2. Note that, $r = O(n)$. We sort the s_i's according to increasing order and find a weighted median δ by using a prefix sum algorithm and Equation (3). This can be done in $O(\log n)$ time, using $O(n)$ processors [6, 23].

4. Construct a δ-covering $\mathcal{R}$ of P by using the algorithm in Section 4.1 and compute $M(\mathcal{R})$. This construction takes $O(n)$ processors and $O(\log n)$ time. There are three possible cases (which can be done in $O(\log n)$ time, using $O(n)$ processors):

   (a) $k \leq M(\mathcal{R}) \leq 4k$, then output δ and $\mathcal{R}$ and stop.

   (b) If $M(\mathcal{R}) < k$ then, for each pair $X[\lfloor (a_{xj}+b_{xj})/2 \rfloor]$ and $X[j]$ (resp. $Y[\lfloor (a_{yj}+b_{yj})/2 \rfloor]$ and $Y[j]$) selected in Step 2 such that the difference of their x (resp. y) coordinate is at most δ, set $a_{xj} := \lfloor (a_{xj}+b_{xj})/2 \rfloor + 1$ (resp. $a_{yj} := \lfloor (a_{yj}+b_{yj})/2 \rfloor + 1$). Goto Step 2.

   (c) If $M(\mathcal{R}) > 4k$, then for each pair $X[\lfloor (a_{xj}+b_{xj})/2 \rfloor]$ and $X[j]$ (resp. $Y[\lfloor (a_{yj}+b_{yj})/2 \rfloor]$ and $Y[j]$) selected in Step 2 such that the difference of their x (resp. y) coordinate is at least δ, set $b_{xj} := \lfloor (a_{xj}+b_{xj})/2 \rfloor - 1$ (resp. $a_{yj} := \lfloor (a_{yj}+b_{yj})/2 \rfloor - 1$). Goto Step 2.

Algorithm 2: Main steps in the algorithm for choosing $(k, 4k; \delta)$-covering of candidate differences which are at most δ. Hence, the invariant holds after such an iteration. If Step $4(c)$ occurs, then from Lemma 4.5, $\delta > \delta^*$ and the algorithm removes differences which are at least equal to δ. The invariant holds in this case also. $\square$

Lemma 4.8 *The algorithm makes at most $O(\log n)$ iterations.*

Proof: Let W and W' be the number of candidate differences immediately before and after an iteration. Also, the current weighted median found by the algorithm is δ. In the following discussion, the subscript m takes two values, namely, x and y. Suppose, a_{mj} and b_{mj} (resp. a'_{mj} and b'_{mj}) denote the intervals immediately before (resp. after) the iteration. We consider Step $4(b)$. If $a'_{mj} \neq a_{mj}$, then $a'_{mj} = \lfloor (a_{mj} + b_{mj})/2 \rfloor + 1$ and $b'_{mj} = b_{mj}$. Since a_{mj} and b_{mj} are integers, we have $a'_{mj} \geq (a_{mj} + b_{mj} + 1)/2$. So,

$$\begin{aligned} W &= W' + \sum_{m=x,y} \sum_{j:a'_{mj} \neq a_{mj}} (a'_{mj} - a_{mj}) \\ &\geq W' + \frac{1}{2} \sum_{m=x,y} \sum_{j} (b_{mj} - a_{mj} + 1) \\ &\geq W' + \frac{1}{2}\frac{W}{2} \end{aligned}$$

The last inequality follows from the fact that δ is a weighted median. Hence,

$$W' \leq \frac{3}{4} W. \tag{6}$$

If Step $4(c)$ applies, the inequality can be proved in a similar way. Since from Lemma 4.7 we know that the invariant is always maintained correctly, the algorithm terminates. From Equation (6), it is clear that at each iteration, at least $(1/4)$th of the candidate differences are discarded. Hence, the lemma follows. $\square$

Lemma 4.9 *A $(k, 4k; \delta)$ covering for the set P can be constructed in $O(\log^2 n)$ time , using $O(n)$ processors in the CREW PRAM.*

Proof: It is clear from the description of Algorithm 2 that each step of the algorithm takes $O(\log n)$ time and $O(n)$ processors. From Lemma 4.8, we know that the algorithm terminates within $O(\log n)$ steps. Hence, the lemma follows. $\square$

5 Parallel algorithm for computing minimum L_∞ perimeter k point subset

In this section, we discuss an algorithm for finding a k-point subset with minimum L_∞ perimeter from a set of $O(k)$ points. This will imply a solution for the problem $S(P, k)$ as discussed in Theorem 3.3. We first prove the following

Lemma 5.1 *For μ the L_∞ perimeter measure, Assumptions 3.1 and 3.2 hold with $c = 4$.*

Proof: If there is no closed $\mu_{opt}(P)$-box that contains the optimal solution P_{opt}, then the L_∞ perimeter of P_{opt} must be greater than $\mu_{opt}(P)$. This shows that Assumption 3.1 holds. For proving Assumption 3.2, let $\delta < \mu_{opt}(P)/4$. If there is a closed δ-box that contains at least k points, then there are k points that have L_∞ perimeter at most $4\delta < \mu_{opt}(P)$. This is clearly a contradiction. $\square$

We need the following property for our algorithm $\mathcal{A}$.

Property 5.1 *The four sides of the minimum L_∞ perimeter k-point subset pass through four points from the set.*

Proof: Otherwise, we can reduce the perimeter by moving the sides. $\square$

In [4], an algorithm is presented for finding such a k-point subset. But their algorithm does not seem to be easily parallelizable. We solve this problem by using the parallel range searching method presented in Section 2.1.

We first sort the points in the $O(k)$ subset according to increasing x coordinate and store them in an array X. This requires $O(\log k)$ time and $O(k)$ processors [6]. We also construct a range tree T as described in Section 2. In this range tree, one processor can perform orthogonal range counting query in $O(\log k)$ time. We also preprocess this range tree, such that given an orthogonal query range, a single processor can find the maximum x coordinate point in that range in $O(\log k)$ time.

If the left side of a rectangle passes through a point p_k, we say that p_k is the left support for the rectangle. Consider a pair of points (p_i, p_j) such that $x_i < x_j$ and $y_i > y_j$. Such a pair (p_i, p_j) fixes the bottom-left corner of a rectangle. We search for a k-point subset enclosed in a rectangle with the bottom-left corner fixed in this way. The left (resp. bottom) support of such a rectangle is the point p_i (resp. p_j). The top support is a point p_k such that $x_k > x_i$ and $y_k > y_i$. We associate $O(k)$ processors with every pair (p_i, p_j). One processor is associated with a point p_k which satisfies the condition above. Now, three supports of the rectangle are fixed and we have to fix the right support such that the rectangle contains k points. This is done in the following way. The processor associated with p_k does a binary search for finding the right support. Notice that, during this binary search, we should consider a point p_m as a candidate for the right support such that, $x_m > x_i$, $x_m > max(x_k, x_j)$ and $y_k > y_m > y_j$. For this purpose, it will be convenient if we have all the points in the semi-unbounded rectangle defined by p_k, p_i and p_j sorted according to increasing x order. But sorting these points for every such semi-unbounded rectangle requires too much work. Instead we do the folloing. We do a binary search in the array X. Suppose, the current candidate for the right support is the point p_n. There are two possibilities. In the first case, $y_k > y_n > y_j$. We do a range search in the range tree T. The query range is a rectangle R such that its three sides are fixed as explained before and the right side is the point p_n. If the number of points inside this rectangle is k, we store this rectangle as a possible candidate rectangle. Otherwise, if the number of points is less than k (resp. greater than k), we shift the right support towards higher (resp. lower) x coordinate points. In the other case, y_n may not be inside the interval $[y_k, y_j]$. We do a range search again with the right side at the position x_n. If there are exactly k points inside this rectangle, we can find the right side of this rectangle by doing a range searching in the tree T and finding the maximum x coordinate point inside this rectangle.

There are $O(k^2)$ pairs like (p_i, p_j). For each such pair, we allocate $O(k)$ processors. So, the overall processor requirement is $O(k^3)$. The binary search takes $O(\log k)$ time and at every position during the binary search, we do a range search which takes $O(\log k)$ time. So, the overall time is $O(\log^2 k)$. After this, there will be at most $O(k^2)$ candidate rectangles left and among them the minimum perimeter rectangle can be found easily in $O(\log k)$ time, using $O(k^2)$ processors. This concludes the discussion of our algorithm. The main steps of the algorithm are given in Algorithm 3.

1. Sort the points according to increasing x coordinates and store them in an array X. This can be done in $O(\log k)$ time using $O(k)$ processors [6].

2. Construct a range tree T for answering orthogonal range counting queries. Also, preprocess this range tree for finding the maximum x coordinate point within a query range. This step takes $O(\log k)$ time and $O(k)$ processors, by the method in Section 2.1.

3. For every pair of points (p_i, p_j) such that $x_i < x_j$ and $y_i > y_j$, we allocate $O(k)$ processors. This group of processors find the minimum L_∞ perimeter k point subset with p_i (resp. p_j) as the left (resp. bottom) support. This step takes $O(\log^2 k)$ time and $O(k^3)$ processors, as discussed above.

4. Among the possible $O(k^3)$ rectangles found in Step 3, find the minimum perimeter one. This can be done in $O(\log k)$ time, using $O(k^3)$ processors.

Algorithm 3: The main steps in the algorithm for finding a minimum L_∞ perimeter k point subset.

Lemma 5.2 *The overall time and work requirements of Algorithm 3 is $O(\log^2 k)$ and $O(k^3 \log^2 k)$.*

Algorithm 3 is applied simultaneously to all the (n/k) subsets of size $O(k)$ each. We state the result in the following

Theorem 5.3 *The minimum L_∞ perimeter k point subset can be computed in $O(\log^2 n)$ time using $O(n \log^2 n + nk^2 \log^2 k)$ work.*

Proof: We get the stated processor and work bounds by plugging in the values from Lemma 5.2 into Equations 1 and 2 in Theorem 3.3. The total time is $O(\log^2 n + \log^2 k)$,i.e., $O(\log^2 n)$, since $n \geq k$. $\square$

6 Parallel algorithm for computing a k point subset with minimum L_∞ diameter

In this section, we describe an algorithm for finding a k point subset with minimum L_∞ diameter. We apply this algorithm simultaneously in parallel on small sets of $O(k)$ points each. This will imply a solution for problem $S(P, k)$ by Theorem 3.3.First, we state the following

Lemma 6.1 *For μ the L_∞ diameter measure, Assumptions 3.1 and 3.2 hold with $c = 1$.*

Proof: Suppose, there is no closed $\mu_{opt}(P)$-box that contains the optimal solution. Then there must be two points in P_{opt} that have L_∞ distance greater than μ_{opt}, a contradiction. This shows that Assumption 3.1 holds. For proving Assumption 3.2, let $\delta < \mu_{opt}(P)$. Assume that there is a closed δ-box that contains k points. Then there is a subset of k points that have L_∞ diameter less than μ_{opt}. This is a contradiction. $\square$

Our algorithm is based on the following

Lemma 6.2 *The minimum L_∞ diameter of a k point subset is determined by the difference, either $|x_i - x_j|$ or $|y_i - y_j|$ for two points p_i, p_j in the set P. Moreover, two opposite sides of the minimum L_∞ diameter set must be supported by these two points.*

Proof: The first part of the Lemma is obvious. If two opposite sides are not supported by the pair of points determining the L_∞ diameter, the L_∞ diameter of the set can be decreased. $\square$

We also need the following result.

Lemma 6.3 *Given a point set P and a fixed size square S, in $O(\log n)$ time and using $O(n)$ processors, we can find an axes-parallel placement of S such that S contains maximum number of points from the set P. This computation can be done in the CREW PRAM.*

Proof: We transform this problem in the following way. We replace each point in the set P by a square of size S. The square S_i corresponding to the point $p_i \in P$ is placed such that its bottom-left corner coincides with the point p_i and its sides are parallel to the axes. Consider the arrangement of this set of squares $S_1, S_2, \ldots, S_n$. We define a graph $\mathcal{H}$ such that the set of vertices for this graph is the set of squares and two vertices are connected by an edge, if the two corresponding squares intersect. Consider the maximum clique in the graph $\mathcal{H}$. It is easy to see that all the squares in this clique has a common point p inside them. We place the square S with its top-right corner at this point. This placement of S will contain all the points in P which gave rise to the clique $\mathcal{H}$. For an algorithm, we can transform the point set into squares of size S in $O(1)$ time, using $O(n)$ processors. Then we apply an algorithm by Chandran *et al* [7] for finding a maximum clique and a deepest point in this arrangement of squares. This algorithm takes $O(\log n)$ time and $O(n)$ processors in CREW PRAM. $\square$

From Lemma 6.2, it is clear that the L_∞ diameter for the minimum k point set is present in the set of all coordinate differences. In our algorithm, we search for the optimal such difference along each axis. We first sort the points along each coordinate axis x and y and store them in two arrays X and Y. Now, the difference in the coordinates along each axis can be represented as a triangular matrix. There are two $O(k) \times O(k)$ triangular matrices in our case. We represent them as M_x and M_y. The first row of M_x has the difference of x coordinate between the first point and all the other points. The matrix M_y is also defined in a similar way. Note that, both rows and columns are in sorted order in these matrices. For example, the first two elements in the jth column of matrix M_x actually denote the differnces $|x_1 - x_j|$ and $|x_2 - x_j|$. Obviously, the first quantity is greater than the second, since the numbers $x_1, x_2, \ldots, x_{O(k)}$ are in sorted order. We do not actually construct the matrices M_x and M_y, since that will require $\Omega(k^2)$ work. But we can generate any element in $O(1)$ time from the sorted arrays X and Y. So, we can effectively assume as if the complete matrices are available. We have to repeatedly select elements from these matrices for doing a binary search to find an appropriate L_∞ diameter. An $O(k)$ time sequential algorithm has been presented in [15] for selection from matrices with sorted rows and columns. The best known parallel algorithm for selecting a specified element from sorted matrices is by Sarnath and He [30]. Their algorithm runs in $O(\log k \log\log k \log^* k)$ time using $O(k/\log k \log^* k)$ processors in the EREW PRAM. The work done by the algorithm is $O(k \log\log k)$. We use their algorithm for our binary search. At every step of our binary search, we select an L_∞ distance from the matrices M_x and M_y. We compute a placement of a square with side lengths equal to this L_∞ distance such that this square contains maximum number of points from the set. This computation can be done by the algorithm in Lemma 6.3, in $O(\log k)$ time and $O(k)$ processors. We stop the binary search when such a square contains k points. Clearly, the time for every step of the binary search is dominated by the complexity of the selection algorithm in [30]. The overall time requirement for the complete binary search is $O(\log^2 k \log\log k \log^* k)$. The total work done by the selection algorithm is $O(k \log k \log\log k)$ and that by the algorihm in Lemma 6.3 is $O(k \log^2 k)$. So, the overall work is $O(k \log^2 k)$. The main steps of the algorithm are given in Algorithm 4.

1. Sort the points according to increasing x and y coordinates and store them in two arrays X and Y.
2. Do a binary search in the matrices M_x and M_y as explained above. At each step of the binary search, find an optimal placement of the square with the current L_∞ distance as its side length. This is done by the algorithm in Lemma 6.3.
3. Stop when an L_∞ distance is found such that the corresponding square contains k points inside it.

Algorithm 4: The main steps in the algorithm for finding a k point subset with minimum L_∞ diameter

Lemma 6.4 *The minimum L_∞ diameter k point subset within a set of $O(k)$ points can be computed in $O(\log^2 k \log\log k \log^* k)$ time and $O(k \log^2 k)$ work in the CREW PRAM.*

As mentioned earlier, this algorithm is applied in parallel to all the $O(k)$ subsets in the grid. So, the overall complexity of the algorithm is stated in the following

Theorem 6.5 *The minimum L_∞ diameter k point subset within a set of n points can be found in $O(\log^2 n + \log^2 k \log\log k \log^* k)$ time and $O(n \log^2 n)$ work.*

Proof: We get the stated time complexity by plugging in the values from Lemma 6.4 into Equations 1 and 2 in Theorem 3.3. The total work is $O(n\log^2 n + n \log^2 k)$, i.e., $O(n \log^2 n)$, since $n \geq k$. $\square$

7 Parallel algorithms for bipartition problems under rectangular and square measures

We consider the bipartition problem under the rectangular measure and the solution for the square measure follows from that. We recall the definitions for the bipartitioning problem under rectangular measure from [21]. Given a point set S, the bounding box of S, denoted by $\square(S)$, is the smallest axes-parallel rectangle that contains S. A rectangular measure is a two-argument function of the height and width of $\square(S)$ and monotone nondecreasing in both the arguments. The interesting rectangular measures are the area, perimeter and diagonal length of the bounding box.

We briefly review the sequential algorithm for the rectangular bipartition problem. Suppose, the extreme points of the set S are denoted by *top*, *bottom*, *left* and *right*. We represent these points by t, b, l and r. The points t and b (resp. l and r) are the points with maximum and minimum y (resp. x) coordinates. Suppose, μ_1 and μ_2 are the specified measures for the rectangular bipartition problem. In other words, if S_1, S_2 are the two partitions, then $\mu(S_1) \leq \mu_1$ and $\mu(S_2) \leq \mu_2$. We assume without loss of generality that $\mu_1 \leq \mu_2$. If $\mu(S) \leq \mu_2$, the bipartition problem is trivial,i.e., $S_1 = \emptyset$ and $S_2 = S$. Otherwise, the four extreme points of S must be divided between S_1 and S_2. This can be done basically in two ways. One of the subsets either gets two or three extreme points. The algorithm in [21] essentially tries out all these assignments of the extreme points. After that they try to build the two partitions based on these assignments. They show that for each such assignment of the extreme points, they can solve the problem in $O(n \log n)$ time. The algorithm in [21] sweeps the set of points in S from minimum to maximum x coordinate. The assignments of the points to the two sets S_1 and S_2 are done during this plane sweep. This plane sweep algorithm seems to be strictly sequential and it is not clear how to parallelize it directly. We solve these problems by a different method. This is based on the parallel range searching method presented in Section 2. It is straightforward to implement our algorithms as alternative sequential algorithms to solve these problems.

As in [21], we solve this problem by assigning the extreme points to the two partitions and then building the partitions based on these assignments. We consider two subcases, when one partition gets three and two extreme points respectively. In the following discussion, we assume that we have two range trees T_1 (resp. T_2) such that we can perform range query for optimum label for x (resp. y) coordinate on T_1 (resp. T_2). In other words, given a query range, we can compute the maximum x (resp. y) coordinate point inside this range in $O(\log n)$ time by a single processor by searching in the tree T_1 (resp. T_2). The construction of these two range trees can be done in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM by the method of Section 2. In the following, the x (resp. y) coordinate of the point p_i are denoted by x_i (resp. y_i). The x (resp. y) coordinate of an extreme point e.g., *top* are denoted by $x(top)$ (resp. $y(top)$).

7.1 Partition S_1 gets three extreme points

We assume without loss of generality that the extreme points *top*, $left$ and *bottom* go to partition S_1 and the extreme point $right$ goes to partition S_2. Consider the bounding box BR for the whole set S. The sides of BR must pass through the four extreme points. We place S_1 with its three sides passing through the three extreme points and the fourth side is placed as far away as possible without violating the constraint $\mu(S_1) \leq \mu_1$. Note that the rectangle BR is now partitioned into two rectangles and the subset S_2 should cover all the points in the set $S \setminus S_1$. We know that one side of S_2 should pass through the extreme point *right*. The other three supports of the sides of S_2 are found in the following way. The top (resp. bottom) side should pass through the maximum (resp. minimum) y coordinate points and the left side through the minimum x coordinate point in $S \setminus S_1$. These three points can be found in $O(\log n)$ time by a processor by range searching in the range trees T_1 and T_2. We do this range searching with the rectangle $BR \setminus S_1$. After that, it can be decided in $O(1)$ time by a single processor whether S_2 can be placed without violating the constraint that $\mu(S_2) \leq \mu_2$.

The other cases when S_1 gets any other three supports can be solved in a similar way.

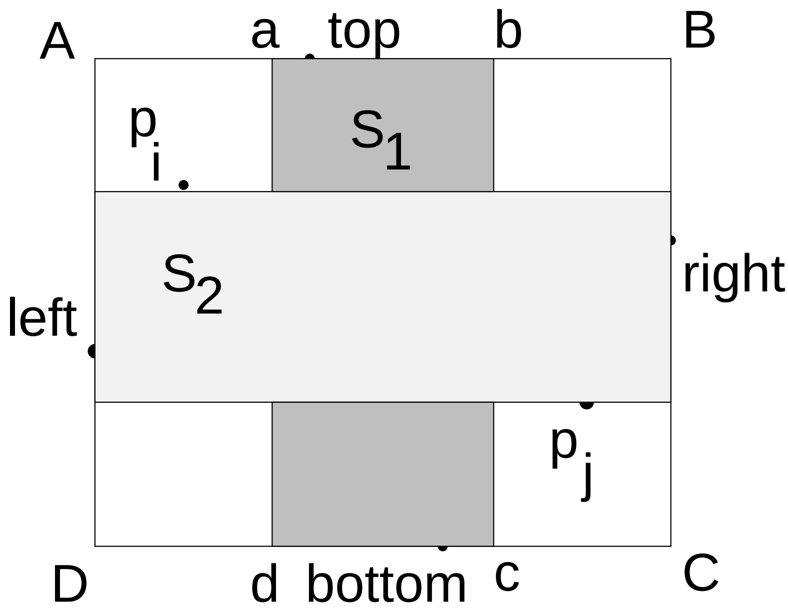

Figure 2: Illustration for Section 3.2.

7.2 Partition S_1 gets two opposite extreme points

We consider the case when the extreme points *top* and *bottom* are in S_1. We refer to Figure 2. Since the top and bottom sides of S_1 pass through the extreme points *top* and *bottom*, we have to fix the other two sides of S_1. We sort the points in the set S according to increasing x coordinates and store them in an array X. This can be done in $O(\log n)$ time using $O(n)$ processors [6]. Now, we assign one processor to each element of the array X. The processor P_k associated with point p_k checks whether $x_k < x(top)$ and $x_k < x(bottom)$ and if so, it finds another point p_l such that $x_l > x_k$ by a binary search in the array X. This point p_l has the property that if the left (resp. right) side of the rectangle S_1 passes through p_k (resp. p_l), $\mu(S_1) \leq \mu_1$ and p_l is such a point with maximum x coordinate. Moreover, both the points *top* and *bottom* lie in the interval $[x_k, x_l]$. It is possible to find such a furthest point p_l through a binary search because all our measures (area, perimeter or diagonal length) are monotone increasing. Consider Figure 2. The partition S_1 is indicated by the rectangle *abcd*. The left side of S_1 passes through the point p_k. The partition S_2 must cover all the points in S not covered by S_1. Since the left and right sides of S_2 must pass through the extreme points $left$ and $right$, we have to fix the other two sides. We find the maximum y coordinate points in the rectangles $AadD$ and $bBCc$. The top side of S_2 must pass through higher of these two points. In Figure 2, this point is indicated by p_i. This point can be found by two range maxima queries in the tree T_2 in $O(\log n)$ time by the processor P_k. Similarly, the bottom side of S_2 should pass through the least y coordinate point in the two rectangles $AadD$ and $bBCc$. If the resulting rectangle S_2 has a measure less than μ_2, this is a valid bipartition. This can be checked by processor P_k in $O(1)$ time. This computation is done in parallel for all the elements of the array X. It is easy to see that the overall computation takes $O(\log n)$ time and $O(n)$ processors. If there is a valid bipartition, it can be detected after this computation is over. The case when partition S_2 gets the two extreme points *top* and *bottom* is similar and the description is omitted.

7.3 Partition S_1 gets two adjacent extreme points

The bounding rectangle for the set S is denoted by BR. We assume without loss of generality that S_1 gets the extreme points $left$ and *bottom*. The other two extreme points *top* and $right$ must go to the set S_2. The inclusion of the two extreme points $left$ and *bottom* essentially fixes the

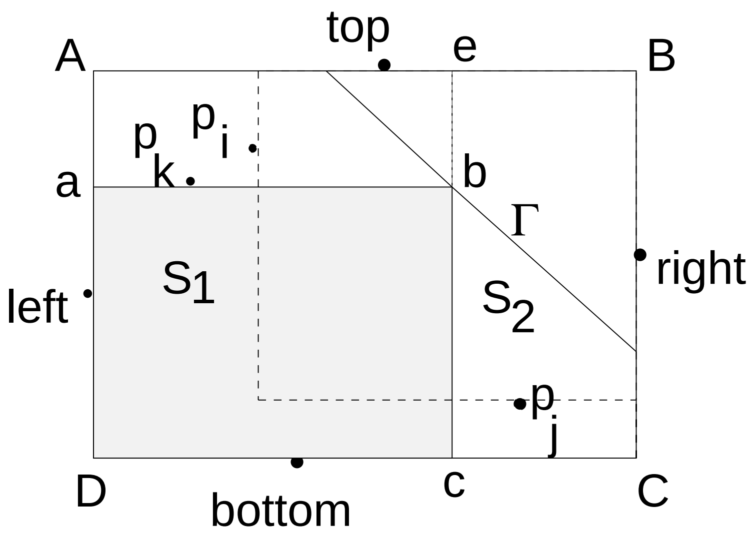

Figure 3: Illustration for Section 3.3.

bottom-left corner for the partition S_1. For finding a valid partition, we have to find the top-right corner of S_1 without violating the constraint that $\mu(S_1) \leq \mu_1$. The locus of the top-right corner is a different curve Γ depending on the specific measure considered. For the subset Z, if $\mu(Z)$ is the area of $\square(Z)$, then Γ is a hyperbola. If $\mu(Z)$ is the perimeter of $\square(Z)$, Γ is a straight line with slope -1. If $\mu(Z)$ is the diagonal length of $\square(Z)$, then Γ is a quarter-circle of radius μ_1 centered at the bottom-left corner of the bounding box BR for the set S. For any such curve Γ, we assume that the intersection of Γ with an axis-parallel straight line can be computed in $O(1)$ time. In the following discussion, we consider the case when the measure μ is the perimeter. The other cases are similar and the description is omitted. For the perimeter case, the equation of Γ (which is a straight line) can be found in the following way. We first restrict the top side of the partition S_1 at the point $left$ and extend the right side as far as possible without violating the constraint $\mu(S_1) \leq \mu_1$. The resulting position of the top-right corner of S_1 is one of the points on Γ. Similarly, if we restrict the right side of S_1 at the point $bottom$ and extend the top side as far as possible without violating the constraint $\mu(S_1) \leq \mu_1$, the top-right corner of the resulting partition S_1 is another point on Γ. The equation of Γ can be found from these two points, since Γ is a straight line.

Since the two extreme points $left$ and $bottom$ are assigned to the partition S_1 (denoted by the shaded area in Figure 3), the bottom-left corner of S_1 is fixed at D. As explained earlier, the top-right corner of S_1 can slide on the curve Γ without violating the constraint $\mu(S_1) \leq \mu_1$. Again, we sort the points in the set S according to increasing x coordinate and store them in the array X. For every point p_k in the array X, we assign a processor P_k. In $O(1)$ time, P_k checks whether the point p_k is in the same side of Γ as the corner D. In that case, it checks whether $y_k > y(left)$. If this check succeeds, p_k is a candidate for the top support of the rectangle S_1. Suppose the intersection of Γ and the horizontal line segment through p_k is the point b. Moreover, the projection of the point b on the bottom side of the bounding rectangle BR is the point c. All the points in the set S outside the rectangle $abcD$ (i.e., outside S_1) should be assigned to the partition S_2. We only consider the case when the rectangle $aAeb$ is non-empty. Here, e is the intersection of the line AB with a vertical line through b. The point with the least x coordinate in this rectangle should be the left support for S_2 (the partition S_2 is shown by dashed lines in Figure 3). We represent this point by p_i. Similarly, the point with the least y coordinate in the rectangle $eBCc$ should be the bottom support for the rectangle S_2. These two points can be found by range searching in the trees T_1 and T_2 in $O(\log n)$ time by processor P_k. Then in $O(1)$ time it can be decided whether the partition S_2 can cover all the points outside S_1 without violating the constraint $\mu(S_2) \leq \mu_2$. This computation is done in parallel for all the points in the array X and after the computation is over, a valid bipartition (if it exists) can be found. The complete computation takes $O(\log n)$ time and $O(n)$ processors.

The algorithm needs concurrent read capability because the computation is done simultaneously for all the elements of the array X. The algorithm for the square measure is similar and is omitted from this version. We summarize the results in the following

Theorem 7.1 *The bipartition problems for the rectangular and square measures can be solved in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM.*

8 Diameter bipartitioning under the L_∞ and L_1 metric

In this section, we briefly discuss the bipartitioning algorithm for the diameter measures. This algorithm is similar to the algorithms in Section 3. A disk of radius r in the L_∞ metric is an axes-parallel square of side length $2r$. In the L_1 metric, a disk is a similar square with side length $\sqrt{2}r$ and rotated by 45^o. We discuss the algorithm only for the case of the L_∞ metric. The algorithm works exactly in the same way for the L_1 metric, if we rotate the plane by 45^o and scale everything by a factor of $\sqrt{2}$.

We denote the point set by S and the axes-parallel minimum bounding box of S by BR. Given two diameters d_1 and d_2, the problem is to find a bipartition such that $diam(S_1) \leq d_1$ and $diam(S_2) \leq d_2$ and $S_1 \cup S_2 = S$. We assume that $d_1 \geq d_2$. We first find the pair of points which determine the L_∞ diameter of the set S. This can be done in $O(\log n)$ time using $O(n)$ processors. We assume without loss of generality that these two points are the two extreme points in the x direction and denoted by p_i and p_j, where $x_i < x_j$. We sort the set of points according to increasing x coordinates and store them in an array X. If $|x_j - x_i| \leq d_1$, we have a bipartition with $S_1 = S$ and $S_2 = \emptyset$. Otherwise, we assign the two extreme points to the two partitions S_1 and S_2 and try to grow the partitions. This computation is done in parallel in the following way. First assume that the point p_i (resp. p_j) has been assigned to partition S_1 (resp. S_2). In the array X, we assign one processor to every point. The processor P_k associated with the point p_k checks the x distance from the extreme point x_j. If $|x_j - x_k| \leq d_2$, P_k does the following computation. It partitions the rectangle B into two rectangles R_1 and R_2 by drawing vertical line through p_k. Suppose the part R_2 contains p_j. In R_2, P_k finds the two extreme points p_m and p_n in the y direction by range searching in $O(\log n)$ time. If $|y_m - y_n| \leq d_2$, this is a valid partition with the diameter of S_2 as d_2. In this case, P_k checks the other rectangle R_1 whether it is a valid partition with diameter d_1. This can be again done in $O(\log n)$ time by finding extreme points through range searching.

This computation is done in parallel for all the points in the array X.

So, in $O(\log n)$ time, using $O(n)$ processor, it can be checked whether a bipartition with this assignment of the extreme points exists. The computation is repeated with assigning p_i (resp. p_j) to partition S_2 (resp. S_1). After this computation is over, a valid bipartition with respect to L_∞ diameter can be found if it exists. The details are omitted. We summarize in the following

Theorem 8.1 *The diameter bipartitioning problem with respect to L_∞ and L_1 metric can be solved in $O(\log n)$ time using $O(n)$ processors in the CREW PRAM.*

9 Conclusion

We have presented efficient parallel algorithms for several clustering and partitioning problems. In case of geometric clustering, we have considered the problems of choosing a k-point subset of a point set which can be covered by a minimum perimeter rectangle and a minimum area square. In case of geometric partitioning, we have considered the problems of bipartitioning a point set in the plane under the L_1 and L_∞ metrics under several different measures. We have also presented optimal parallel algorithms for range searching.

There are several open problems related to the problems considered in this paper. For clustering problems, we have succeeded in parallelizing only two of the problems considered in [11, 14]. It is not known whether the other clustering problems can be solved efficiently in parallel. For partitioning problems, it is an interesting open problem to design efficient parallel algorithms in the L_2 metric. It is not clear how to parallelize the methods presented in [21]. Another interesting open problem is to construct a range tree in parallel in dimensions ≥ 3.

Acknowledgements

The author would like to thank Prof. Rolf Klein and Prof. Thomas Ottmann for their support during the period in which most of this work was done, and Michiel Smid for many interesting discussions on clustering problems.

References

[1] H. C. Andrews. *Introduction to Mathematical Techniques in Pattern Recognition.* Wiley-Interscience, New York, 1972.

[2] D. Avis, "Diameter partitioning", *Discrete Comput. Geom.*, **1** (1986), pp. 265-276.

[3] T. Asano, B. Bhattacharya, M. Keil and F. Yao, "Clustering algorithms based on minimum and maximum spanning trees", *Proc. 4th ACM Symp. on Comp. Geom.* (1988), pp. 252-257.

[4] A. Aggarwal, H. Imai, N. Katoh and S. Suri. "Finding k points with minimum diameter and related problems", *J. Algorithms*, **12** (1991), pp. 38-56.

[5] O. Berkman, D. Breslauer, Z. Galil, B. Scheiber and U. Vishkin. "Highly parallelizable problems", *Proc. of 21st Annual ACM Symp. on Theory of Computing*, 1989, pp. 309-319.

[6] R. Cole. "Parallel merge sort", *SIAM J. Comput.*, **17**, (1988), pp. 770-785.

[7] S. Chandran, S. K. Kim and D. M. Mount. "Parallel computational geometry of rectangles", *Algorithmica*, **7**, (1992), pp. 25-49.

[8] S. Chandran, *Merging in Parallel Computational Geometry.* PhD thesis, Department of Computer Science, University of Maryland, College Park, MD, 1989.

[9] A. Datta, "Efficient parallel algorithms for geometric k-clustering problems",*Proc. 11th Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science, Springer-Verlag*, **Vol. 775**, (1994), pp. 475-486.

[10] A. Datta, "Efficient Parallel Algorithms For Geometric Partitioning Problems Through Parallel Range Searching", *Proc. 23rd International Conference on Parallel Processing*, Illinois, USA, (1994), **Vol. III**, pp. 202-209.

[11] A. Datta, H. P. Lenhof, C. Schwarz and M. Smid. "Static and dynamic algorithms for k-point clustering problems", *Proc. Workshop on Algorithms and Data Structures (WADS), Lecture Notes in Computer Science*, **Vol. 709**, (1993), pp. 265-276. A full version will appear in *Journal of Algorithms.*

[12] Z. Drezner, "On the rectangular p-center problem", *Naval Res. Logist.*, **34**, (1987), pp. 229-234.

[13] D. P. Dobkin, R. L. Drysdale and L. J. Guibas. "Finding smallest polygons", In: F. P. Preparata (ed.), Advances in Computing Research, Vol. **1**, Computational Geometry, J. A. I. Press, London, (1983), pp. 181-214.

[14] D. Eppstein and J. Erickson. "Iterated nearest neighbours and finding minimal polytopes", *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, (1993), pp. 64-73.

[15] G. N. Frederickson and D. B. Johnson. "The complexity of selection and ranking in $X+Y$ and matrices with sorted columns", *Journal of Computer and System Sciences*, **24**, (1982), pp. 197-208.

[16] M. T. Goodrich, "Intersecting line segments in parallel with an output- sensitive number of processors", *SIAM J. Comput.*, **20**, (1991), pp. 737-755.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, Freeman, San Fransisco, 1972.

[18] S. Guha, *An $O(n)$ time set bipartition algorithm under the L_∞ metric.* Manuscript, EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, 1989.

[19] S. Guha, *Personal communication*, 1993.

[20] J. A. Hartigan. *Clustering Algorithms*, John-Wiley, New York, 1975.

[21] J. Hershberger and S. Suri, "Finding tailored partitions", *J. Algorithms*, **12**, (1991), pp. 431-463.

[22] D. S. Johnson, "The NP-completeness column", *J. Algorithms*, **3**, (1982).

[23] J. JáJá. *An introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[24] D. B. Johnson and T. Mizoguchi. "Selecting the Kth element in $X+Y$ and $X_1 + X_2 + ... + X_m$", *SIAM J. Comput.*, **7**, (1978), pp. 147-157.

[25] M. D. Katz and D. J. Volper, "Geometric retrieval in parallel", *J. Parallel Distrib. Comput.*, **5**,(1988), pp. 92-102.

[26] H. P. Lenhof and M. Smid. "Sequential and parallel algorithms for the k-closest pairs problem", Max Planck Institut für Informatik, Technical Report, MPI-I-92-134. August 1992. A preliminary version appears in *Proc. 33rd Annual IEEE Symp. on Foundations of Computer Science*, pp. 380-386.

[27] C. Monma and S. Suri, "Partitioning points and graphs to minimize the maximum or the sum of diameters", *Proceedings, Sixth International Conference on the Theory and Applications of Graphs*, Wiley, New York, 1989.

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

[29] R. Sridhar, S. S. Iyengar and S. Rajanarayanan, "Range search in parallel using distributed data structures", *J. Parallel Distrib. Comput.*, **15**, (1992), pp. 70-74.

[30] R. Sarnath and X. He. "Efficient parallel algorithms for selection and searching on sorted matrices", *Proc. 6th International Parallel Processing Symposium*, (1992), pp. 108-111.