

Fast and optimal parallel multidimensional search in PRAMs with applications to linear-programming and related problems ^{*}

Martin E. Dyer [†]

Sandeep Sen [‡]

Abstract

We describe a deterministic parallel algorithm for linear programming in fixed dimension d that takes $\text{poly}(\log \log n)$ time in the common CRCW PRAM model and does optimal $O(n)$ work. In the EREW model, the algorithm runs in $O(\log n \cdot \log \log^{d-1} n)$ time. Our algorithm is based on multidimensional search and effective use of approximation algorithms to speed up the basic search in the CRCW model. Our method also yields very fast $\text{poly}(\log \log n)$ algorithms for smallest enclosing sphere and approximate ham-sandwich cuts and an $O(\log n)$ time work-optimal algorithm for exact ham-sandwich cuts of separable point sets. For these problems, in particular for fixed-dimensional linear programming, $o(\log n)$ time efficient deterministic PRAM algorithms were not known until very recently.

1 Introduction

We consider the standard linear programming problem: given a set \mathcal{H} of n half-spaces in \mathbb{R}^d and a vector y , find a point $x \in \bigcap \mathcal{H}$ that minimizes $y \cdot x$. We restrict ourselves to the case where n is much larger than d , and we focus attention on parallel algorithms that achieve good performance with respect to n . Since the general problem is known to be P-complete [11], the restricted version assumes more relevance. For the most part we will regard d as a constant. However, since the running time of our algorithm grows rapidly with d , we will attempt to examine the exact nature of this dependence. Megiddo [28] described a linear time algorithm for linear-programming in fixed dimension d (hereafter referred to as LP d), using an elegant multidimensional search technique. The same search technique, also known as prune-and-search, yielded optimal algorithms for other optimization problems (see Megiddo [27] and Dyer [12, 13]). Following Megiddo's linear time algorithm, there have been significant improvements in the constant factor (which was doubly exponential in d) due to Dyer [13] and Clarkson [4]. Further progress was made, using random sampling, by Clarkson [5]. This algorithm was later derandomized by Chazelle and Matoušek [8]. With more careful analysis and some additional ideas, Matoušek et al. [26] improved it further that achieves an expected running time of $O(d^2 \cdot n + e^{O(\sqrt{d \log d})})$, a "sub-exponential" dependence on d . Independently, Kalai [23] discovered an algorithm with matching performance and these are presently the fastest known algorithms.

In the context of parallel algorithms for LP d , there have been a number of results in the direction of finding a fast analogue of Megiddo's linear-time sequential method [1, 2, 10, 30]. A straightforward parallelization of Megiddo's method yields an $O(\log^d n)$ time n -processor EREW PRAM algorithm. (We write $\log^k n$ for

^{*}Preliminary versions of the main results of this paper appeared in [14] and [31]

[†]School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK, dyer@scs.leeds.ac.uk

[‡]Department of Computer Science and Engineering Indian Institute of Technology New Delhi 110016, India, ssen@cse.iitd.ernet.in

$(\log n)^k$, and similarly $\log \log^k n$ for $(\log \log n)^k$.) However, this is far from the best that can be achieved. Alon and Megiddo [2] obtained an optimal parallel algorithm that runs in constant time (i.e. time depending only on d) on an n -processor randomized CRCW PRAM model. They used a variant of Clarkson's algorithm [5], which has smaller constants (in terms of d) than Megiddo's. More recently, Ajtai and Megiddo [1] developed a deterministic $O(\log \log^d n)$ algorithm in an n processor model which allows $O(\log \log n)$ time selection. Since there is an $\Omega(\log n / \log \log n)$ bound for exact selection [3], their model is considerably more powerful than the CRCW model, and partly non-uniform. It may also be noted that the work bound in the algorithm of Ajtai and Megiddo is super-linear, since they use a linear number of processors. Thus an $o(\log n)$ time optimal deterministic algorithm for LP d had hitherto proved elusive. Very recently, Goodrich [18] (see also Goodrich and Ramos [?]), independently obtained a $\text{poly}(\log \log n)$ time CRCW algorithm with optimal speed-up using fast parallel derandomization techniques. His approach is different from ours, being directly based on parallel derandomization techniques. His underlying randomized algorithm is similar to that of Dyer and Frieze [15]. This is derandomized using fast deterministic constructions of ε -approximations in \mathbb{R}^d . Here, we generalize the basic multidimensional search algorithm of Megiddo. Instead of pruning a constant fraction of the constraints in each round, we increase the rate of pruning as a function of *processor advantage*. The processor advantage is proportional to ratio of the number of processors to the number of constraints, which increases monotonically during the course of the search. This allows us to improve the $\Omega(\log n)$ phases seemingly required by Megiddo's approach, and leads to a fast EREW algorithm. However, it does not directly yield a $\text{poly}(\log \log n)$ CRCW algorithm for the following reason. All known versions of multidimensional search algorithms rely on exact selection procedures, either deterministic or randomized. We make use of the observation that exact selection may be substituted by *approximate splitting* (defined formally in section 3) without significantly affecting the asymptotic convergence rate of the procedure. This is crucial, in order to surmount the $\Omega(\log n / \log \log n)$ barrier for exact parallel selection: an impossibility in Ajtai and Megiddo's approach. We apply this observation, using fast deterministic splitting algorithms in the CRCW model, to obtain a $\text{poly}(\log \log n)$ time algorithm. Moreover, we are able to achieve linear work simultaneously. Although all presently known $\text{poly}(\log \log n)$ approximate splitting algorithms are based on derandomization, they are relatively simpler and more efficient than constructing ε -nets in \mathbb{R}^d as in Goodrich's algorithm [18]. However, it may be noted that the fastest algorithm known for approximate splitting follows from applying Goodrich's techniques. This could be used in our algorithm to save an $O(\log \log n)$ factor. However we have chosen to ignore this in our later description. Rather, we feel that other $\text{poly}(\log \log n)$ time approximate splitting algorithms, like that of Hagerup and Raman [22], which are relatively simpler (although slower), are more in keeping with the basic simplicity of our approach. Our method also implies a very simple $\text{poly}(\log \log n)$ randomized CRCW algorithm, although this is asymptotically slower than Alon and Megiddo's algorithm. Our approach has immediate applications to other optimization problems where Megiddo's prune-and-search technique is effective. We are not aware of any previous $\text{poly}(\log \log n)$ time deterministic algorithms for problems like *minimum enclosing sphere* and *ham-sandwich cuts*. The latter especially has numerous applications to divide-and-conquer algorithms, including those for convex hulls and range-searching. The paper is organized as follows. In section 2, we describe our basic approach, generalizing Megiddo's multidimensional search technique. This directly yields a fast EREW algorithm. In section 3, we modify the algorithm of section 2, substituting some of the exact procedures by their faster approximate counterparts. In section 4, we analyze the algorithms for the EREW and CRCW PRAM and bound the total work by $O(n)$. In section 5, we present further applications of our search algorithm to problems like *Euclidean 1-center* and *ham-sandwich cuts* for separable planar point-sets. Because of the close similarity of these algorithms to that for LP d , we omit detailed analysis. In section 6, we make some observations regarding the relationship between *hyperplane cuttings* and our approach. We show the existence of a direct geometric construction for cuttings that does not require derandomization. However, the size of this cutting is probably too large for general

application.

2 A Parallel Multidimensional search method

Without loss of generality, we assume that LP d has feasible region

$$\{x \in \mathbb{R}^d : x_d \leq \sum_{j=1}^{d-1} \alpha_{ij} x_j + b_i, \quad (i = 1, 2, \dots, n)\},$$

and the objective is to minimize x_d . Megiddo [28] noted that LP d can be viewed as determining the *critical* linear constraints which define the optimal point x^* . If we assume non-degeneracy (no $d + 1$ hyperplanes intersect in a point of the feasible region), then exactly d constraints define the optimal point and the remaining constraints may be eliminated without affecting the optimum value. In its full generality, the LP d problem also requires us to report if the optimum is unbounded or if the feasible region is empty. In the description that follows, we do not discuss these conditions explicitly, noting that Megiddo's approach can handle these cases effectively. Suppose we have a set of n hyperplanes,

$$H_i = \{x \in \mathbb{R}^d : a_i \cdot x = b_i\}, \quad (i = 1, 2, \dots, n).$$

The *sign* of H_i with respect to $y \in \mathbb{R}^d$ is defined as $\{-, +, =\}$ depending on

$$a_i \cdot y - b_i \begin{matrix} < \\ = \\ > \end{matrix} 0$$

Geometrically, we wish to determine which side of the hyperplane contains y (including the possibility that y may lie on the hyperplane). We would like to determine the sign of y with respect to all the hyperplanes H_i , ($i = 1, 2, \dots, n$). To do this, we extend our definition to the sign of an *arbitrary* hyperplane in a similar fashion. Megiddo showed that, by determining the signs of a constant number $A(d)$ of carefully chosen hyperplanes, we can determine the sign of a constant fraction $B(d)$ of the H_i 's. Here $A(d)$ and $B(d)$ are functions of d only. By repeatedly applying this strategy, one can determine all the signs relatively quickly (as compared to evaluating them individually). Megiddo used an elegant inductive argument to show the existence of $A(d)$ and $B(d)$, starting from the observation that for $d = 1$, $A(1) = 1$ and $B(1) = 1/2$ (by determining the sign of the median half-line). If y is x^* , and the hyperplanes are the constraints of LP d , then determining the sign of all hyperplanes is clearly equivalent to solving LP d . Hereafter, we will be interested in determining the sign of a hyperplane (not necessarily a constraint) with respect to x^* . However, x^* is itself unknown at the outset. Megiddo showed that the sign of an arbitrary hyperplane can be determined by solving at most three linear programming problems (recursively) in one lower dimension. Using a nontrivial recursive strategy (involving recursion in both n and d), Megiddo constructed an algorithm for LP d with running time linear in n but doubly exponential in d . For convenience, we assume that no constraint hyperplane is parallel to x_d . (This can be achieved by standard perturbation techniques, if necessary.) From here, we focus on the following *hyperplane location* problem: Given a set N of hyperplanes of the form

$$H_i = \{x \in \mathbb{R}^d : x_d = \sum_{j=1}^{d-1} a_{ij} x_j + b_i\} \quad (i = 1, 2, \dots, n),$$

we wish to determine the sign at x^* of all H_i . This is often referred to as the *multidimensional search* problem. We will say that H_i is *located* if we know the sign of H_i with respect to the linear programming optimum x^* .

As already noted, a straightforward parallelization of Megiddo's algorithm yields an $O(\log^d n)$ time algorithm with n processors. It is inefficient because, in the later stages of the algorithm when relatively few constraints remain, most of the processors are idle. We now describe a parallelization of multidimensional search which uses processors more efficiently in the later stages, when the number of processors greatly exceeds the number of constraints. The algorithm proceeds in stages, where during each stage, we determine the sign of some hyperplanes. At the beginning of the stage i , we denote the set of hyperplanes by N_i , their number by n_i and define processor advantage r_i to be $\max\{2, (p/n_i)^{1/d}\}$ where p is the number of processors. For example, $n_0 = n$ and $r_0 = 2$ if $p \leq n$. Again, for simplicity of presentation, we assume that r_i is an integer (or else take the floor function). Also, in the remaining part of this section we will write N , n and r instead of N_i , n_i and r_i since the description is limited to iteration i , for some i .

Par_Mul_Search(N, p, d)

1. The set N of hyperplanes is partitioned into r equal sized sets based on their $a_{\ell 1}$ values ($\ell = 1, 2, \dots, n$). This is done by selecting the $k(n/r)$ th ranked elements (denoted by σ_k) from $\{a_{11}, a_{21} \dots a_{n1}\}$ for $k = 1, 2, \dots, r$. For convenience, write $\sigma_0 = -\infty$. Denote this partition of N by \mathcal{P} , where the k th class of \mathcal{P} consists of hyperplanes H_ℓ which satisfy $\sigma_{k-1} < a_{\ell 1} \leq \sigma_k$ ($k = 1, 2, \dots, r$).
2. For $d = 1$, solve the problem directly and exit. (See below.) Otherwise, partition N into r -sized subsets (groups) by picking (for each group) one constraint from each class of \mathcal{P} . Consider two hyperplanes H_i and H_j in a group and say $a_{i1} \leq a_{j1}$. Since they come from different classes of \mathcal{P} , there are values σ_k, σ_l such that $\sigma_k \in [a_{i1}, a_{j1}]$ and $\sigma_l \notin [a_{i1}, a_{j1}]$.
3. Using the transformation

$$x_1' = x_1 + \sigma_k \cdot x_2 \text{ and } x_2' = x_1 + \sigma_l \cdot x_2$$

we obtain two hyperplanes H_{ij} and H_{ji} where H_{ij} (respectively H_{ji}) is obtained by eliminating x_1' (respectively x_2') between H_i and H_j . Intuitively these are planes that pass through the intersection of H_i and H_j and are parallel to the x_1' and x_2' coordinate axes respectively. There are two such hyperplanes for each pair in the group. From Megiddo's observation (see also Dyer [13]), it follows that if we can locate both H_{ij} and H_{ji} (which are hyperplanes in \mathbb{R}^{d-1}) then we can locate at least one of H_i and H_j . Figure 1 illustrates the situation in 2 dimensions.

4. We recursively apply the search algorithm in one lower dimension to all such hyperplanes H_{ij}, H_{ji} . Note that all H_{ij} lie in a subspace corresponding to the absence of x_1' but possible presence of x_2' . Note there are only r different variables of the type x_1' , one for each of the r different σ_k values. Thus there are only r distinct subspaces for which the algorithm must be called recursively. The algorithm is called in parallel for all the distinct subspaces, allocating processors to the subproblems proportional to their sizes. At the bottom of this recursion on d , we will generate *special* hyperplanes with respect to which we must locate x^* . Location for these special hyperplanes involves recursively solving three $(d-1)$ -dimensional linear programming problems. Each such problem is a restriction of the original LPd to a particular hyperplane. (See Megiddo [28] for further details of these lower-dimensional problems).
5. Repeat the previous step R_d times, where R_d is a function of d that we will determine during the analysis. By repeating this step, we eliminate a fraction of the remaining hyperplanes.

6. The located hyperplanes are now eliminated from consideration, the new processor advantage is determined, and the above procedure is repeated.

The total number of hyperplanes generated in step 2 is $(n/r)r(r-1) < nr$ since every pair of hyperplanes in a group generates two lower-dimensional hyperplanes. Step 4 actually involves a double recursion, one of which we have unfolded (i.e. how to generate the special hyperplanes inductively in \mathbb{R}^d) for exposition of the basic algorithm. This is similar to Megiddo's exposition [28] of the sequential algorithm, where a certain proportion $B(d)$ of hyperplanes is located using $A(d)$ queries, these functions being defined inductively. However, instead of the number of queries, we bound the number of *rounds* of parallel recursive calls. The number R_d will actually be defined according to the following recursive scheme. For the same set of σ_k 's and r , we do c rounds of location (where c is some suitable constant) for the H_{ij} 's, where a certain proportion of H_{ij} 's are located recursively. This is done in order to boost the proportion of hyperplanes located in d dimensions, for inductive application. This idea has been used before in [4, 13]. The difference here is that we cannot wait for the results of location in one group before we start on the next, as is done in the sequential algorithm. The number of groups r may be large, and so we must operate on them all in parallel. For $d = 1$, the constraints are half-lines and the linear programs can be solved directly. The problem is equivalent to extremal selection (selection of minimum or maximum), which can be solved in $O(\log n)$ time on an EREW PRAM and $O(\log \log n)$ time on a CRCW PRAM [?]. As constraints are eliminated in each repetition of Step 4, data compaction must be done, to ensure efficient processor utilisation in the next. The processor requirement is determined by the total size of all problems that have to be solved simultaneously on all levels of the recursion on d . We must show that this remains bounded by the total number of processors. We will show this below and, more significantly, we will show that the recursive procedure in d dimensions takes only $O(\log \log n)$ rounds to eliminate all constraints. This will lead to an overall time bound of $O(\log n \log \log^{d-1} n)$ for LP d on the EREW PRAM.

3 Multidimensional search using approximate computations

To circumvent the $\Omega(\log n / \log \log n)$ lower bound for exact selection in the CRCW PRAM, we will modify step 1 of the previous section. We propose to use approximate splitting in the partitioning step and approximate compaction to do load balancing. As noted previously, extremal selection can be done in $O(\log \log n)$ time in CRCW PRAM. Formally, the problem of *approximate splitting* is defined as :

Given a set of N of n elements and an integer r , $1 \leq r \leq n$ and an expansion factor $\theta > 1$, choose a set R of r elements (out of N) such that the maximum number of elements of N in any interval induced by (the sorted order of) R is less than $\theta \cdot n/r$.

Intuitively, $\theta = 1$ gives even splitting but we will relax that significantly to speed-up our algorithm. For approximate splitting, on CRCW model, we use the following result implied by the work of Hagerup and Raman[22] and Goldberg and Zwick[20] (see appendix for details).

Lemma 1 *For all given integers $n \geq 4$, and $C \leq r \leq n^{1/4}$, approximate splitting can be done with expansion factor \sqrt{r} in $O(\log \log^2 n)$ time and $O(r \cdot n)$ CRCW processors where C is a sufficiently large constant.*

For smaller r , we partition the a_{i2} 's into two sets by an approximate median which can be found quickly using the following result of Goldberg and Zwick.

Lemma 2 *For a given set X of n elements and any fixed $\epsilon > 0$, an element $x \in X$ whose rank r^l satisfies $\frac{n}{2} \leq r^l \leq \frac{n}{2}(1 + \epsilon)$ can be found in $O(\log \log n)$ steps using $n / \log \log n$ CRCW processors.*

The problem of *approximate compaction* was defined by Hagerup [21] as follows.

Given an n -element set, of which a are active, place the active elements in an array of size $(1 + \lambda)a$, where λ is the padding factor.

As one might expect, the running time of fast parallel algorithms for this problem increases as λ decreases. Hagerup described an $O(\log \log^3 n)$ time work-optimal CRCW algorithm for the above problem for $\lambda = 1/\text{poly}(\log \log n)$. The following improved result was recently obtained by Goldberg and Zwick[20].

Lemma 3 *The approximate compaction problem can be solved on a CRCW PRAM in time $t \geq \log \log n$, using n/t processors, with a padding factor of $1/\text{poly}(\log \log n)$.*

Suppose we substitute exact compaction by approximate compaction. Then, over the $O(d)$ levels of recursion, there will only be a slowdown in running time by a factor of $(1 + \lambda)^{O(d)}$, i.e. a constant. We will also use the following result on *semisorting* to collect the elements of subproblem (i.e. a class or a group) together. The semisorting problem may be defined as follows.

Let \mathcal{A} be an array of n elements such that each element has an integer label $\ell \in \{1, 2, \dots, k\}$. Sort the elements of \mathcal{A} into disjoint subarrays \mathcal{A}_ℓ ($\ell = 1, 2, \dots, k$) such that, if there are n_ℓ elements with label ℓ , the subarray \mathcal{A}_ℓ must be of size $O(n_\ell)$.

The following is known about semisorting from [22]:

Lemma 4 *Semisorting problems of array size n and range size k can be solved in $O(\log \log n)$ CRCW time using kn processors.*

So, if C is the constant for the splitting algorithm above, the modified algorithm is as follows. If $(p/n_i)^{1/d} \geq \beta$, we choose $r_i = (p/n_i)^{1/d}$, otherwise we choose $r_i = 2$, where $\beta \geq C$ is a constant that we will determine below, and use the method of Lemma 1 to partition the a_{i1} 's, $1 \leq i \leq n$ into classes. Once the splitters are determined, the partitioning can be done by brute force, comparing a_{i1} against all splitters simultaneously in parallel. (There are sufficient processors for this to be done.) We put the hyperplanes in each class in near-contiguous locations by an application of semisorting and approximate compaction. Then we give each hyperplane in a class a group number corresponding to its position in the derived subarray. We now form groups by semisorting on group number, followed by approximate compaction of the group subarray. Thus each group will pick at most one hyperplane from each class, each hyperplane will belong to exactly one group, and each group is held in an array little longer than its size. There are sufficient processors throughout since each uncompact array is only a constant factor larger than the group size, by Lemma 4. The per-processor overhead caused by arrays not being exactly compacted is a small constant (in fact $o(1)$), and hence we ignore it in the analysis. Note, however, that some groups may have less than r hyperplanes. The number of groups is determined by the size of the largest partition since we take one hyperplane from each partition. Because of the approximate splitting, there could be n/\sqrt{r} groups, rather than the n/r which would result from exact splitting. However the size of each group is no more than r because the number of classes is only r . We will denote the set of groups by \mathcal{G} , and their number by $|\mathcal{G}|$. As indicated in the previous section, we pair hyperplanes from a group and solve the lower dimensional problem recursively a constant number of times, c (to be determined below). Then we start a new iteration by throwing out constraints and compacting the remaining using Lemma 3. At the bottom level (when the hyperplanes are points), we choose r splitters, so that the size of the largest partition is at most n/\sqrt{r} . Location with respect to the r splitters can be done simultaneously in $O(\log \log n)$ time using extremal selection. We have chosen r so that we will have enough processors at the bottom level.

4 Analysis

First we will prove a slightly weaker result, that our algorithm takes $O(\log \log^{d+1} n)$ in a CRCW PRAM using $p = n$ processors, and subsequently reduce the number of processors. The analysis is generalized to handle approximate splitting. Therefore, in particular, the bounds that we prove are applicable to the basic algorithm of section 2. For convenience of presentation, we will refer to hyperplanes in \mathbb{R}^d as d -hyperplanes. Our first objective is to bound the number of rounds of hyperplane locations required to eliminate a constant fraction of d -hyperplanes. Let π_d denote the fraction of d -hyperplanes that are located in R_d rounds for a fixed r_i , say r . Since the maximum size of a partition at the bottom level is n/\sqrt{r} , $\pi_1 \geq (\sqrt{r} - 1)/\sqrt{r}$. The following will be used to write a recurrence for π_d . The groups refer to the groupings of \mathcal{G} as defined in the previous section.

Lemma 5 *The number of $(d - 1)$ -hyperplanes that have to be located in the recursive call is bounded by $n \cdot (r - 1)$.*

Proof: As pointed out in the previous section, although $|\mathcal{G}|$ could be n/\sqrt{r} because of uneven splitting, the number of $(d - 1)$ -hyperplanes can be bounded by $\sum_{g \in \mathcal{G}} n_g(n_g - 1)$ where n_g is the number of d -hyperplanes in group $g \in \mathcal{G}$. Since $n_g \leq r$, and $\sum n_g = n$, the number of $(d - 1)$ -hyperplanes is bounded by $n/r \cdot r(r - 1)$. \square

Observation 1 *Suppose, in a certain group of d -hyperplanes, at most $m(m - 1)/2$ of the $(d - 1)$ -hyperplanes have not been located. Then at most m d -hyperplanes are not located.*

Proof: Recall that the $(d - 1)$ -hyperplanes are obtained by taking all possible pairs in a group and generating two $(d - 1)$ -hyperplanes from each pair. Suppose there are $m' (> m)$ unlocated d -hyperplanes; consider the $m'(m' - 1)/2 > m(m - 1)/2$ pairs formed by these. At least one $(d - 1)$ -hyperplane from each pair has not been located, since locating both $(d - 1)$ -hyperplanes in a pair would imply that one of the d -hyperplanes is located. This contradicts the antecedent of the observation that at most $m(m - 1)/2$ pairs have not been located. \square

In order to write a recurrence for π_d , we let \bar{n}_g denote the number of unlocated d -hyperplanes in group g and let u_g denote the number of unlocated $(d - 1)$ -hyperplanes (among the $n_g(n_g - 1)$ generated in g). From the above notations,

$$\pi_d = \frac{1}{n} \left(n - \sum_g \bar{n}_g \right)$$

From Lemma 5 and the recursive application (i.e., a fraction π_{d-1} of the $d - 1$ -hyperplanes are located in R_{d-1} rounds),

$$\sum_g u_g \leq (1 - \pi_{d-1})^c \cdot n(r - 1) \tag{1}$$

From Observation 1,

$$\begin{aligned} u_g &\geq \frac{1}{2} \bar{n}_g \cdot (\bar{n}_g - 1) \\ \text{implying} \quad 2u_g + 1 &\geq \bar{n}_g^2 - \bar{n}_g + 1 \\ \text{implying, in turn} \quad 2u_g + 1 &\geq n_g \end{aligned}$$

Therefore $\sum_g \bar{n}_g \leq \sum_g (2u_g + 1)$. Substituting in equation 1, we can write the recurrence for π_d as follows:

$$\pi_d \geq \frac{1}{n} \left(n - 2(1 - \pi_{d-1})^c n(r - 1) - n/\sqrt{r} \right), \tag{2}$$

where the last term on the right corresponds to $\sum_g 1 = |\mathcal{G}| \leq n/\sqrt{r}$. Simplifying, we obtain

$$\pi_d \geq 1 - 2(1 - \pi_{d-1})^c(r - 1) - 1/\sqrt{r}$$

It is straightforward to verify by induction that, for $d \geq 2$, $\pi_d \geq \frac{\sqrt{r}-2}{\sqrt{r}}$, provided $r \geq 8$ and $c \geq 12$. If $r = 2$, we choose an approximate median with accuracy $1/6$ (i.e. $\epsilon = 1/6$ in Lemma 2). Then equation 2 becomes

$$\pi_d \geq \frac{1}{n}(n - 2(1 - \pi_{d-1})^c \cdot n - 2n/3),$$

since $|\mathcal{G}| \leq n/2 + n/6 = 2n/3$. For $c = 12$, and $\pi_1 \geq 1/3$, $\pi_d \geq 1/4$ by induction. Thus we take $c = 12$ and $\beta = \max\{8, C\}$ in our algorithm description where C is the constant from Lemma 1. The total number of rounds R_d for location in dimension d satisfies $R_d = c^{d-1}$, since $R_d = c \cdot R_{d-1}$ and $R_1 = 1$. Hence we can write the recurrence for n_i , the number of surviving constraints at the beginning of iteration i as

$$n_{i+1} \leq n_i(1 - \pi_d) \leq \begin{cases} \frac{3}{4}n_i & (r_i = 2) \\ \frac{2}{\sqrt{r_i}}n_i & (r_i \geq \beta) \end{cases}.$$

After $O(d)$ iterations with $r_i = 2$, we will have $(n/n_i)^{1/d} \geq \beta$. Note that then $n_i/n \leq \frac{1}{8}$. Thereafter

$$n_{i+1} \leq 2n_i^{1+1/2d} / n^{1/2d}$$

Letting $\xi_i = n_i/n$, we can rewrite the previous inequality as

$$\xi_i \leq 2\xi_i^{1+1/2d}$$

where $\xi_1 = \frac{1}{8}$. Therefore $\xi_i = O(1/n)$ (i.e. $n_i = O(1)$) when $i \geq c_1 \log \log n$, for some constant $c_1 = O(d)$. Each iteration involves splitting, compaction and a constant number of recursive call to lower dimensional problems. Let $T_d(n)$ be the running time for a d -dimensional problem of size n . Then we can write the recurrence

$$T_d(n) \leq c_1 \log \log n [c^{d-1}(T_{alloc} + T_{split} + 3T_{d-1}(n))] \quad (3)$$

where T_{alloc} and T_{split} denote the times for data compaction and approximate splitting respectively. From Lemma 3, $T_{alloc} = O(\log \log n)$ and from Lemma 1, $T_{split} = O(\log \log^2 n)$ respectively. Using $T_1(n) = O(\log \log n)$, $T_d(n)$ is $O(\log \log^{d+1} n)$ for $d \geq 2$. Since $c_1 = O(d)$, the implied constant is $2^{O(d^2)}$, which is of the same form as that in the improvement of Megiddo's algorithm due to Dyer [13] and Clarkson [4]. The processor requirement is bounded by the number of $(d-1)$ -dimensional linear programming problems which must be solved simultaneously at the bottom level. Each such problem has n_i constraints per problem. This number is the total number of hyperplanes that must be located simultaneously. This is bounded by $n_i(r_i - 1)^{d-1}$ from Lemma 5, since the number of distinct linear programming instances grows by an $(r_i - 1)$ factor at each level of the recursion on d . But this is at most n , since either $r_i = 2$ or $r_i = (n/n_i)^{1/d} \geq 8$. We can now state our intermediate result as

Lemma 6 *Linear programming in fixed dimension $d \geq 2$ can be solved in $O(\log \log^{d+1} n)$ CRCW PRAM steps, using n processors.*

Remark: Using a faster approximate splitting algorithm due to Goodrich [18], with $T_{split} = O(\log \log n)$, the running time would decrease to $O(\log \log^d n)$. The previous analysis also holds for the basic algorithm of section 2 and therefore we can analyze the performance of the algorithm on the EREW PRAM simply by substituting bounds for exact selection and compaction. Since then $T_{alloc} = O(\log n)$ and $T_{split} = O(\log n)$, we obtain similarly to the above the following result.

Lemma 7 *Linear programming in fixed dimension $d \geq 2$ can be solved in $O(\log n \log \log^{d-1} n)$ steps using n processors in a EREW PRAM.*

To reduce the number of operations to $O(n)$, we use a slow-down technique, applied inductively. We outline our strategy in the case of CRCW PRAM. We have observed that the one-dimensional problem can be solved in $O(\log \log n)$ steps using $n / \log \log n$ processors. Assume that the $(d - 1)$ -dimensional problem can be solved in $O(\log \log^d n)$ steps optimally. The standard slow-down method implies that, for any time $t > \log \log^d n$, these problems can be solved with optimal work. We start with $p = n / \log \log^{d+1} n$ processors and reduce the number of constraints to $n / \log \log^{d+1} n$ by running the algorithm with $r_i = 2$ for $O(\log \log \log n)$ iterations. Let $t_d = \log \log^{d+1} n$. In each iteration, we apply the work-optimal $(d - 1)$ -dimensional method. Now iteration i (starting with $i = 0$) takes $\max\{\gamma^i t_d, O(\log \log^d n)\}$ steps. Here $\gamma \leq \frac{3}{4}$ is the fraction of constraints eliminated at each iteration. In each iteration, we use the result of Lemma 3 to do load balancing. This takes $O(\max\{\gamma^i t_d, \log \log n\})$ steps. The total time taken to reduce the number of constraints to $n / \log \log^{d+1} n$ by the above method can therefore be bounded by $O(\log \log^{d+1} n + \log \log^d n \cdot \log \log \log n)$. At this stage, we switch to the n -processor algorithm described previously. Therefore we can state our final result.

Theorem 1 *Linear programming in fixed dimension $d \geq 2$ can be solved in $2^{O(d^2)} \log \log^{d+1} n$ CRCW PRAM steps using $n / \log \log^{d+1} n$ processors.*

Following an identical strategy, but using $t_d = \log n \log \log^{d-1} n$ and $O(\log n)$ for the load balancing time, we obtain an analogous result for the EREW PRAM. However, the running time increases by an $O(\log^* n)$ factor, since the best work-optimal EREW selection algorithm known runs in $O(\log n \log^* n)$ time (Cole [9]).

Theorem 2 *Linear programming in fixed dimension $d \geq 2$ can be solved in $2^{O(d^2)} \log n \log^* n \log \log^{d-1} n$ EREW PRAM steps, using $n / (\log n \log^* n \log \log^{d-1} n)$ processors.*

5 Other applications

Although the search algorithm was described in the context of linear programming, the method extends to problems where prune-and-search methodology has produced linear-time sequential algorithms. We outline two of these applications here, namely, finding *the smallest enclosing circle* (or, more generally, *the Euclidean 1-center problem*) and finding *ham-sandwich cuts* of separable point-sets on the plane.

5.1 Smallest enclosing circle

Given a set $N = \{(a_i, b_i), 1 \leq i \leq n\}$ of points in the plane, we wish to determine a circle \mathcal{C} that encloses all the points and is smallest among all such circles. This has a natural analogue in higher dimension, where we determine the smallest enclosing sphere in the appropriate dimension. Megiddo [27] described a linear time algorithm for the smallest enclosing circle problem. Dyer [13] extended this to solve the more general *weighted Euclidean one-center* problem in any fixed dimension, using additional tools from convex optimization. The idea of Megiddo's solution is very similar to the linear programming algorithm. The minimum enclosing circle \mathcal{C} is defined by at most three points, so the solution remains unchanged if we eliminate the remaining points. The algorithm proceeds in iterative phases, where in each phase a constant fraction of the input points are eliminated with linear work. The crucial issue is to recognize which points can be eliminated using carefully designed queries. In the following, we follow Megiddo's [27] description with appropriate modifications for the parallel algorithm. We will first solve a restricted version of the problem, where the center of \mathcal{C} is constrained to lie

on a given straight line. Without loss of generality, assume this line is the x -axis. Denote the center of \mathcal{C} by x_c . Pair up the points arbitrarily and denote the perpendicular bisector of pair $(a_i, b_i), (a_j, b_j)$ by \perp_{ij} . It can be readily seen that there exists a critical value x_{ij} (the intersection of \perp_{ij} with the x -axis), such that depending on $x_c \stackrel{<}{\underset{>}{\approx}} x_{ij}$ one of the points in the pair can be eliminated, the one that is closer to x_c . So if we can determine answers to queries of the form “Is $x \stackrel{<}{\underset{>}{\approx}} x_c$? for arbitrary x , we can use these to eliminate some of the points. For example, by evaluating the query at x_m , the median value of x_{ij} s, we can eliminate one point from every pair for half the number of pairs, i.e. a quarter of the points. Determining the sign of an arbitrary x ($x \stackrel{<}{\underset{>}{\approx}} x_c$?) is easily done by finding the furthest point (in Euclidean distance) from x . Denote the square of this distance by $g(x)$. Let $I = \{i : (x - a_i)^2 + b_i^2 = g(x)\}$. If $x < a_i$ for every i then, $x < x_c$. If $x > a_i$ for every i then $x > x_c$, else $x = x_c$. All this can be done in linear time. So, applying the above procedure recursively to the remaining points (at most $3/4n$), x_c can be determined in $O(n)$ steps. More formally, we are solving the following optimization problem (unrestricted case).

$$\min_{x,y} f(x, y) = \max_i \{(x - a_i)^2 + (y - b_i)^2\}.$$

Note that $f(x, y)$ is a convex function of its arguments. In the preceding paragraph, we described a method to solve the constrained problem when $y = 0$. Given an arbitrary y , the sign of y is defined to be $\{+, =, -\}$ depending on $y \stackrel{<}{\underset{>}{\approx}} y^*$ where (x^*, y^*) is the unconstrained optimum of $g(x, y)$. Because of the convexity, the sign of y can be determined from the sign at (x_c, y) , where x_c is the constrained optimum. The sign at (x_c, y) can be determined by an application of linear programming in plane or alternatively by a direct method of finding a line separator (see Megiddo [27]). The overall algorithm is very similar to the linear programming algorithm. Here we consider pairs of points instead of pairs of constraints. Each pair of point $(a_{2i-1}, b_{2i-1}), (a_{2i}, b_{2i})$ defines a perpendicular bisector \perp_i such that, if we could determine which half-plane of \perp_i contains (x^*, y^*) , we could eliminate one point from the pair. This is similar to determining the sign of \perp_i . We pair perpendicular bisectors, \perp_i and \perp_j say, and compute their intersection p_{ij} . Consider the two lines X_{ij} and Y_{ij} , passing through p_{ij} , parallel to the x and y axes respectively. If the slopes of \perp_i and \perp_j do not have the same signs, then by determining the signs of X_{ij} and Y_{ij} , we can eliminate one of the four points (defining \perp_i and \perp_j). Figure 2 illustrates this situation. Determining the sign of X_{ij} or Y_{ij} is essentially a lower dimensional problem (the constrained version) that we solved before. So, we have reduced the searching problem to one lower dimension. The strategy is identical to that of computing signs of hyperplanes in section 2. Here the hyperplanes are defined by pairs of points (the perpendicular bisectors). The groupings are done on the basis of slopes and the algorithm is applied recursively to the r different subspaces generated. Here r is processor advantage, which will be defined similarly to the linear programming algorithm. The analysis is carried out in a fashion almost identical to section 4, and we omit further details. Note that computing the sign of a bisector eliminates at least one of the two defining points. Hence we can summarize as

Theorem 3 *The minimum enclosing circle of n points on the plane can be determined in $O(\log \log^3 n)$ CRCW time, using $n / \log \log^3 n$ processors.*

The above algorithm extends to any fixed dimension d . To determine the sign of a hyperplane in dimension $d > 2$, we will use linear programming in dimension d after determining the center of the constrained smallest sphere (center lies on the hyperplane). This is to determine if the center of the constrained sphere is contained within the the convex hull of the points I determining the optimum. If it is contained, then we have the global unconstrained optimum. Otherwise the optimum lies in the direction perpendicular to the hyperplane bounding

the d extreme points of I . This additional complication increases the running time by an $O(\log \log n)$ factor compared with the linear programming algorithm.

Corollary 3.1 *The minimum enclosing sphere of n points in E^d for $d > 2$ can be determined in $O(\log \log^{d+2} n)$ time using $n / \log \log^{d+2} n$ CRCW processors.*

5.2 Ham sandwich cuts and partitioning planar point-set

A line l is called the *bisector* of a point set S if each open half-plane defined by the line contains at most half the points of S . It is known that, given two point sets P and Q , it is possible to bisect them simultaneously using a line l . This line is called a *ham-sandwich* cut of P and Q . In the special case where P and Q are linearly separable, Megiddo [29] described a linear-time algorithm based on prune-and-search. Later, Lo and Steiger [24] described a linear-time algorithm for the general case. Megiddo's algorithm is useful in situations where we need to partition a point set into four quadrants using two lines. We can choose one line to be the median with respect to one of the axes. This partitions the points into two sets S_1 and S_2 . Then, using Megiddo's algorithm, we can take the other line to be a ham-sandwich cut of S_1 and S_2 . This has numerous applications to divide-and-conquer algorithms. Below we show that our search strategy yields a fast parallel algorithm for this special case of ham-sandwich cuts. However, it must be noted that ham-sandwich cuts can be used to compute an exact median of a set of points on the x -axis. (Simply take this set as P and an arbitrary two-point set in the upper half-plane as Q .) Thus the lower bound on exact median-finding prevents us from attaining $\text{poly}(\log \log n)$ performance. Therefore, we will first design a $O(\log n)$ time algorithm for this problem. Then we will show how to obtain *approximate ham-sandwich* cuts in $\text{poly}(\log \log n)$ time. An approximate ham-sandwich cut will mean that each of P and Q will be partitioned *approximately* equally. We describe Megiddo's method briefly. (Our version actually follows the description in [16].) The problem is solved in the dual space where a point $p = (a, b)$ is mapped to the line $D(p) : y = 2ax - b$, and a non-vertical line $l : y = \lambda_1 x + \lambda_2$ is mapped to the point $D(l) : (\lambda_1/2, -\lambda_2)$. Thus $D^2 = D$, and it is known moreover that D preserves incidence and the below-above relationship. (See [16] for details). Thus the P and Q are mapped to sets of lines $G = D(P)$ and $H = D(Q)$. Assume, with loss of generality, that the lines of G and H have non-positive and non-negative slopes respectively (by choosing the x -axis as a separating line in the primal plane). In the arrangement $\mathcal{A}(S)$ formed by a set of lines S , a *level* k , ($1 \leq k \leq |S|$) is the set of points of $\mathcal{A}(S)$ that lie below *exactly* k lines of S . We will write $\mathcal{L}_i(S)$ for the i th level of S . In the arrangements $\mathcal{A}(G)$ and $\mathcal{A}(H)$, the median levels correspond to lines that bisect P and Q in the primal plane. Because of the slope constraints we have imposed, the levels in G and H are monotonically non increasing and nondecreasing respectively. This implies that $\mathcal{L}_i(G)$ and $\mathcal{L}_j(H)$ have a non-empty intersection for all i, j . We wish to determine a point p^* in the intersection of the median levels. Then $D(p^*)$ is a ham-sandwich cut for P and Q . We actually solve a more general problem. We determine a point s in the intersection of the i th level of G and the j th level of H . Let $m = |G|$ and $n = |H|$. The algorithm proceeds in phases, where in the k th phase, we are looking for a point common to the levels g_k of $\mathcal{A}(G^k)$ and h_k of $\mathcal{A}(H^k)$. Here G^k and H^k are the subsets of G and H active during the k -th phase. Initially, $k = 0, g_0 = i, h_0 = j, G^0 = G$ and $H^0 = H$. In the k th phase, we eliminate lines from G^k and H^k that cannot contain the common point using *line tests*. A test consists of determining which side of a given line t contains p^* . (The test may also yield a point in the intersection, in which case the algorithm terminates.) Then new sets G^{k+1}, H^{k+1} are calculated after elimination of some lines. New values of g_{k+1} and h_{k+1} are chosen, and we proceed to iteration $k + 1$. A test with respect to a line t is similar to computing the sign in our general search strategy. It can be done in linear time sequentially and involves computing a g_k th and h_k th intersection point of G^k and H^k respectively on the line t . There are various cases, and we refer the reader to [16, pp 339–341] for details. We use the following optimal selection algorithm from Chaudhuri, Hagerup and Raman [6].

Lemma 8 For all integers $n \geq 4$, selection problems of size n can be solved in $O(\log n / \log \log n)$ CRCW time using $n \log \log n / \log n$ processors.

Using this as a subroutine, we can compute the *sign* of a line t in $O(\log n / \log \log n)$ time. Consider the set of lines $L = H \cup G$. Suppose $l_1, l_2 \in L$ have slopes of the opposite sign and $p_{1,2}$ is their common intersection point. Let t_x and t_y be the horizontal and vertical lines through $p_{1,2}$. By determining the signs of t_x and t_y , we can determine at least one of the lines l_1 or l_2 which cannot contain any point s in the intersection of the levels. (Otherwise, we can actually determine such a point s , and the algorithm terminates.) We can therefore apply our search strategy to this situation by partitioning the lines of L using their slopes. Here we can use exact selection using Lemma 8 since we have more time at our disposal. The remaining algorithm is very similar to our previous search algorithms, and the analysis follows along similar lines. Since each testing costs us $O(\log n / \log \log n)$ time and the algorithm has $O(\log \log n)$ stages, the total time is $O(\log n)$ using n processors. We may also use exact compaction. Finally, using a technique similar to Theorem 1, we can reduce the number of processors to $O(n / \log n)$. Thus

Theorem 4 A ham-sandwich cut of two linearly separable sets P and Q can be computed in $O(\log n)$ CRCW steps, using $n / \log n$ processors, where $n = |P \cup Q|$.

An *approximate* ham-sandwich cut of P and Q , with *relative accuracy* λ , ($0 < \lambda < 1$) will be defined as follows. The cut is a line l which simultaneously partitions P and Q , so that the partition of P (respectively Q) does not contain more than $|P|(1 + \lambda)/2$ (respectively $|Q|(1 + \lambda)/2$) points. In the context of the dual setting this implies that we have to determine a point s that lies in the intersection of $\mathcal{L}_i(G)$ and $\mathcal{L}_j(H)$ where $(1 - \lambda)|G|/2 \leq i \leq (1 + \lambda)|G|/2$ and $(1 - \lambda)|H|/2 \leq j \leq (1 + \lambda)|H|/2$. Ghose and Goodrich [19], describe a simple algorithm for this problem using random sampling followed by verification. We follow the lines of our previous algorithms, with the modification that testing with respect to a line is done using approximate selection. We use the following algorithm for parallel approximate selection from [6].

Lemma 9 For all integers $n \geq 4$ and $t \geq \log \log^4 n$, approximate selection with relative accuracy $2^{-t / \log \log^4 n}$ can be achieved in $O(t)$ time, using an optimally in $O(n)$ operations. Furthermore, for $q \geq 1$, a relative accuracy of 2^{-q} can be achieved in $O(q + \log \log^4 n)$ time, using $O(qn)$ operations.

We also use Lemma 3 to update the values of g_k, h_k at each iteration. Because of the use of approximate algorithms, testing with respect to a line t returns an answer satisfying the following property. Given g_k, h_k and a line t , the test returns a half-plane (bounded by t) which contains an intersection point of level $(1 \pm \epsilon)g_k$ of G^k with level $(1 \pm \epsilon)h_k$ of H^k . (Here ϵ is the relative accuracy of our selection algorithm.) Moreover, the updated values of g_k and h_k are accurate within a multiplicative factor $(1 + 1/\text{poly}(\log n))$. Since there are $O(\log \log n)$ iterative phases of the algorithm, the overall accuracy can be bounded by $(1 \pm \epsilon)^{O(\log \log n)}$. From Lemma 9, $\epsilon \leq 1/\text{poly}(\log n)$ is achievable in $\text{poly}(\log \log n)$ steps. Hence an overall relative accuracy of $1/\text{poly}(\log n)$ is achievable using this approach. Therefore we have the following result

Theorem 5 An approximate ham-sandwich cut of linearly separable sets P and Q , with relative accuracy $\lambda \leq \log^{-a} n$ for some fixed constant a , can be computed in $O(\log \log^6 n)$ steps, using $O(n)$ operations, where $n = |P \cup Q|$.

6 Some observations on our method

In this section we offer some insights into the multidimensional search technique relative to other methods, in particular random-sampling based strategies. The randomized algorithm of Clarkson [5] (later derandomized

by Chazelle and Matoušek [8]) exploits efficient geometric partitioning methods based on ϵ -net constructions (often referred to as *cuttings* in the context of this problem). A $1/r$ -cutting \mathcal{C} , for a set H of hyperplanes in \mathbb{R}^d , is a collection of d -dimensional simplices (with disjoint interiors) which covers \mathbb{R}^d . Furthermore, no simplex intersects more than $|H|/r$ hyperplanes. The number of simplices in \mathcal{C} is called the *size* of the cutting. The derandomization methods are actually techniques for constructing cuttings whose existence is guaranteed by the probabilistic method. It was shown by Matoušek [25] and Chazelle [7] that $1/r$ -cuttings of size $O(r^d)$ can be constructed in time $O(n \cdot r^{d-1})$, using a derandomization technique based on the method of conditional probabilities (known as *Raghavan-Spencer* technique). It is fairly clear that, given such a cutting, the multidimensional search method becomes easier to implement. We will show that the converse is also true— that the multidimensional search method implies a cutting. This was noted somewhat less explicitly in some previous papers (see [?, ?]). While the size of the cutting is much worse than one obtains from the derandomization methods, it does not involve derandomization, and is a direct geometric constructon. The intuition is as follows. The multidimensional search in d dimensions eliminates a fixed fraction $B(d)$ of hyperplanes by location with respect to a set of hyperplanes R of size $A(d)$. Let us denote the arrangement in \mathbb{R}^d induced by the hyperplanes in R by $\mathcal{A}(R)$, and the region in $\mathcal{A}(R)$ that contains the optimum by Δ^* . Clearly the number of hyperplanes intersecting Δ^* cannot exceed $|H|(1 - B(d))$, since any plane that has been located cannot intersect Δ^* . However, it is not obvious how many hyperplanes intersect other regions $\Delta \in \mathcal{A}(R)$. If the same bound applies, then we have a $(1 - B(d))$ -cutting. This is indeed so, since Megiddo’s algorithm selects, at the bottom-most level of recursion, a fixed set of hyperplanes and does the location with respect to these. Irrespective of where the query point lies, the previous bound holds. The size of this cutting is less obvious. We reduce the number of hyperplanes intersecting Δ^* by iterating a fixed number of times c on the lower dimensional problem. This is an adaptive improvement that is local to Δ^* and it is not clear how it affects other regions. However, modifying equation 2 from section 4 as follows, will give us a bound on the size of a $1/r$ -cutting.

$$\pi_d \geq \frac{1}{n}(n - 2(1 - \pi_{d-1}) \cdot n(r - 1) - n/r). \quad (4)$$

We have set the exponent $c = 1$ for the reasons mentioned above, and replaced n/\sqrt{r} with n/r since we may do exact splitting in the present context. In equation 4, it can be verified that $\pi_d \geq 1 - 2/r$ if $\pi_{d-1} \geq 1 - \frac{1}{2r^2}$. Applying this inductively, we find that $\pi_1 \geq 1 - \frac{1}{2^{d-1}r^{2^{d-1}}}$ implies $\pi_d \geq 1 - 2/r$. This implies a $2/r$ cutting in \mathbb{R}^d . For any m , a $1/m$ -cutting of \mathbb{R}^1 has size $(m - 1)$ (by choosing $(m - 1)$ equally spaced values). Thus the overall cutting contains $2^{d-1}r^{2^{d-1}}$ hyperplanes. This is exponentially larger than the cuttings of size $O(r^d)$ obtained from ϵ -net constructions. However for small constants r and d , this may be a reasonable alternative to the derandomized schemes for constructing $1/r$ -cuttings. We may summarize the discussion of this section as follows.

Observation 2 *Megiddo’s algorithm implies a simple linear time method (without derandomization) for constructing $1/r$ -cuttings of size $(2^{d-1}r^{2^{d-1}})^d$ in \mathbb{R}^d , where d is a fixed constant.*

7 Concluding Remarks

This paper makes two main contributions to parallel fixed-dimensional linear programming. First, we show that an alternative implementation of Megiddo’s [28] technique enables efficient processor utilisation. Second, we circumvent the bottleneck of median-finding, found in earlier adaptations of Megiddo’s approach, by substituting approximate selection algorithms. (Note that even randomized selection has the same bottleneck.) However, the underlying approximate algorithms have large associated constants, and the algorithms of Goldberg and Zwick [20] use expanders. We can eliminate the need for expanders by settling for a somewhat slower algorithm

(by a factor of $O(\log \log^3 n)$), using the method of Hagerup and Raman [22]. On the other hand, the running time of Theorem 5 can be improved somewhat by using the selection algorithm of [20]. Finally, we remark that reducing the dependence on d of the running time of our algorithm for LP_d is a challenging open problem.

Acknowledgment The second author wishes to thank Rajeev Raman for very helpful discussions regarding formulating the present version of Lemma 1 and Pankaj Aggarwal for bringing [18] to his attention. The authors also acknowledge several helpful comments of the anonymous reviewers that helped improve the presentation. In particular one of the reviewers pointed out that a simple modification in the approximate splitting algorithm (described in the appendix) can improve the running time by a factor of $O(\log \log n)$.

References

- [1] M. Ajtai and N. Megiddo. A deterministic $\text{poly}(\log \log n)$ time n processor algorithm for linear programming in fixed dimension, *SIAM J. on Computing*, Vol. 25, 1996, 1171–1195.
- [2] N. Alon and N. Megiddo. Parallel linear programming in fixed dimensions almost surely in constant time, *Journal of the ACM*, 1994, 422–434.
- [3] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAMS. *Proc. of the 19th Annual Symposium on Theory of Computing*, 1987, 83–93.
- [4] K. Clarkson. Linear Programming in $O(n3^{d^2})$ time. *Information Processing Letters*, 22, 1986, 21–24.
- [5] K.L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM*, Vol 42, 1995, 488–499.
- [6] T. Chan, J. Snoeyink and C. Yap. Primal dividing and dual pruning: output-sensitive construction of 4-d polytopes and 3-d Voronoi diagrams. *Discrete and Computational Geometry*, 18, 1997, 433 – 454.
- [7] S. Chaudhuri, T. Hagerup and R. Raman. Approximate and exact deterministic parallel selection. In *Proc. 18th Math. Foundations of Comp. Sci.*, LNCS 711, Springer-Verlag, 1993, pp. 352-361.
- [8] B. Chazelle Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9(2), 1993, pp. 145–158.
- [9] B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. *Proc. of the 4th ACM Symposium on Discrete Algorithms*, 1993, pp. 281–290.
- [10] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters* 26, 1988, 295–299.
- [11] X. Deng. An optimal parallel algorithm for linear programming in the plane. *Information Processing Letters* 35, 1990, 213–217.
- [12] D. Dobkin, R. Lipton and S. Reiss. Linear programming is log-space hard for P, *Information Processing Letters* 8, 1979, 96–97.
- [13] M. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM Journal on Computing* 13, 1984, 31–45.
- [14] M. Dyer. On a multidimensional search technique and its application to the Euclidean one-centre problem, *SIAM Journal on Computing* 15, 1986, 725–738.
- [15] M. Dyer. A parallel algorithm for linear programming in fixed dimension. *Proc. of the 11th Symposium on Computational Geometry*, 1995, pp. 345–349.
- [16] M. Dyer and A. Frieze. A randomized algorithm for fixed-dimensional linear programming. *Math. Programming* 44, 1989, 203–212.
- [17] H. Edelsbrunner. Algorithms in combinatorial geometry. EATCS Monographs in Theoretical Computer Science, Elsevier, 1986.
- [18] M. Goodrich. Geometric partitioning made easier even in parallel *Proc. of the 9th ACM Symposium on Computational Geometry*, 1993, 73–82.
- [19] M. Goodrich. Fixed-dimensional parallel linear programming via relative ϵ -approximations, *Proc. of the Seventh ACM-SIAM SODA*, 1996.
- [20] M. Goodrich and E. Ramos. Bounded-Independence Derandomization of Geometric Partitioning with Applications to Parallel Fixed-Dimensional Linear Programming, *Discrete and Computational Geometry*, Vol 18, 1997, 397 – 420.
- [21] M. Ghose and M.T. Goodrich. In-place techniques for parallel convex-hull algorithms. *Proc. 3rd ACM Sympos. Parallel Algo. Architect.*, 1991, pp. 192-203,

- [22] T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. *Proc. Israel Symp. on Theory and Computing Systems (ISTCS'95)*, 1995, pp. 220-228.
- [23] T. Hagerup. Fast deterministic processor allocation. *Proc. of the 4th ACM Symposium on Discrete Algorithms* 1993, pp. 1–10.
- [24] T. Hagerup and R. Raman. Fast deterministic approximate and exact parallel sorting. *Proc. of the 5th Annual Symposium on Parallel Algorithms and Architectures*, 1993, pp. 1–10.
- [25] Joseph Ja' Ja'. *Introduction to Parallel Algorithms*, Addison Wesley. 1992.
- [26] G. Kalai. A subexponential randomized simplex algorithm, *Proc. of the 24th ACM Symposium on Theory of Computing*, 1992, pp. 475–482.
- [27] C.Y. Lo and W. Steiger. An optimal-time algorithm for ham-sandwich cuts in plane. *Proc. 2nd Canadian Conf. on Comput. Geom.*, 1990, pp. 5–9.
- [28] J. Matoušek. Computing the center of planar point sets. *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 6, 1991, 221 – 230.
- [29] J. Matoušek. Cutting hyperplane arrangements. *Discrete and Computational Geometry* 13, 1991, 385–406.
- [30] J. Matoušek, M. Sharir and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, Vol. 16, 1996, 498–516.
- [31] N. Megiddo. Linear time algorithm for linear programming in R^3 and related problems. *SIAM Journal on Computing* 12, 1983, 759–776.
- [32] N. Megiddo. Linear programming in linear time when dimension is fixed *Journal of the ACM* 31, 1984, 114–127.
- [33] N. Megiddo. Partitioning with two lines in the plane. *Journal of Algorithms* 6, 1985, 430–433.
- [34] S. Sen. Finding an approximate median with high probability in constant parallel time, *Information Processing Letters* 34, 1990, 77-80.
- [35] S. Sen. Parallel multidimensional search using approximate algorithms: with applications to linear programming and related problems. *Proc. of the ACM Symp. on Parallel Algorithms and Architectures Padua, Italy*, 1996.

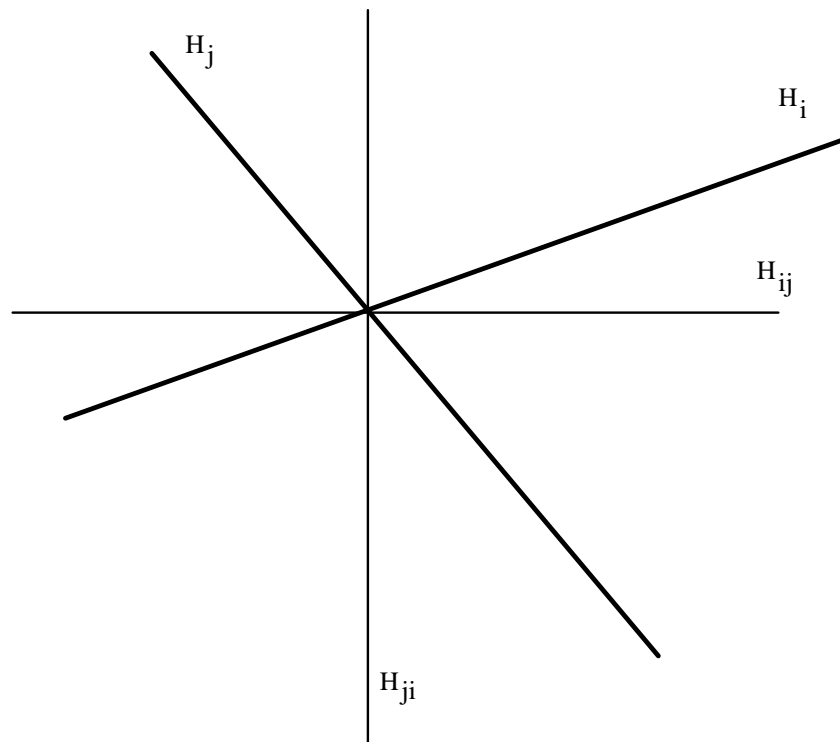


Figure 1: Each quadrant is completely contained in a half-plane determined by H_i or H_j . So by determining the signs of both H_{ij} and H_{ji} , we can determine the sign of H_i or H_j . For example, if the optimum lies in the NW quadrant, H_i 's sign is determined. (H_{ij} and H_{ji} are not necessarily perpendicular.)

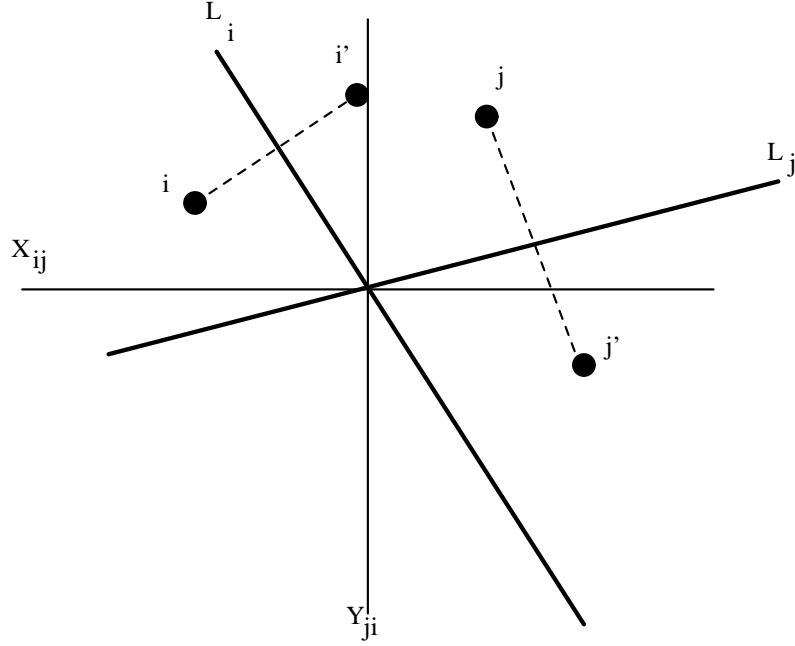


Figure 2: If the center of the optimum circle is located in the SW quadrant, clearly point i will be strictly closer than i' and hence can be eliminated.

8 Appendix

We give a brief outline of the proof of Lemma 1. The ideas are adapted mainly from Hagerup and Raman[22], where the reader can find more detailed proofs. Throughout, we will use $\nu(r)$ to denote a factor of the form $1 + \frac{1}{\text{polylog}(r)}$.

Proof of Lemma 1

Hagerup and Raman [22] describe a method for computing r approximate splitters in $O(\log \log^5 n)$ CRCW time, using rn processors. Below we describe how to speed up their method by substituting faster routines for *approximate prefix computation* due to Goldberg and Zwick [20]. The problem *approximate prefix-sum* is defined as follows:

A sequence $0 = b_0, b_1, \dots, b_n$ is said to be an ϵ -approximate prefix sum of a given nonnegative sequence a_1, \dots, a_n if we have

$$\sum_{j=1}^i a_j \leq b_i \leq (1 + \epsilon) \sum_{j=1}^i a_j$$

and $b_i - b_{i-1} \geq a_i$ ($1 \leq i \leq n$). Lemma 3 is actually derived from the following result on approximate prefix sums.

Lemma 10 ([20]) *For any fixed $\alpha > 0$, and fixed $\delta > 0$, an $1/(\log^\alpha n)$ -approximate prefix sum sequence can be computed in $O(1)$ CRCW time, using $n^{1+\delta}$ processors.*

Substituting the previous result in the proof of Lemma 11 of [22], we obtain

Lemma 11 *A set of n keys can be padded-sorted with padding factor $\nu(n)$ in $O(\log \log n)$ time, using a polynomial number of processors.*

To improve the processor bound in the previous lemma, we apply Lemma 10 recursively to a sample-sort algorithm. For this we compute a *group sample* as follows:

Let A be a set of n numbers, m be an integer and let $s = \frac{n}{m^2}$. Partition A into s groups A_1, \dots, A_s , each of size m^2 . Padded-sort the A_i 's into subarrays of size $\nu(n)m^2$ and let B_i be the multiset consisting of the elements with rank $\nu m, 2\nu m \dots \nu m^2$ in A_i . Then $B = \bigcup_i B_i$ is a group sample of A .

Remark: Computing a group sample of size n/m involves sorting sets of size $O(m^2)$. By choosing a group sample of appropriate size and using its members as splitters in a Quicksort-like algorithm (but for only a constant depth of recursion), we obtain the following result, along the same lines as Hagerup and Raman [22].

Lemma 12 *For any fixed $\epsilon > 0$, n keys can be padded-sorted in $O(\log \log n)$ CRCW time, with padding-factor $\nu(n)$, using $n^{1+\epsilon}$ processors.*

An (m, λ) -sample of a set A is a subset B such that, for each pair of elements $x, y \in B$,

$$|\text{rank}_A(x) - \text{rank}_A(y)| \leq \frac{m|A|}{|B|} |\text{rank}_B(x) - \text{rank}_B(y)| + \lambda|A|.$$

By choosing a group sample $B \subseteq A$, using the procedure described previously, we can show that:

Lemma 13 *B is a $(\nu(m), \nu(m)/m)$ -sample of A , computable in $O(\log \log n)$ CRCW time, using mn processors. Moreover $|B| = n/m$.*

The proof follows by summing $(|\text{rank}_{B_i}(x) - \text{rank}_{B_i}(y)| + 1)$ over all B_i . From the definition of (m, λ) sample, the following can easily be verified.

Lemma 14 ([22]) *Let B be a (r, λ) -sample of A and let C be a (r', λ') sample of B . Then C is a $(rr', r\lambda' + \lambda)$ sample of A .*

We now describe the algorithm for finding r approximate splitters of a set X of n numbers. We may assume we have nr processors initially. Thus, using Lemma 13, we compute a subset $B_1 \subset X$, which is a $(\nu(m), \nu(m)/m)$ -sample, where $m = r$. Note that $|B_1| = n/r$, so we now have a processor advantage of r^2 . Next we compute B_2 , which is a $(\nu(r^2), \nu(r^2)/r^2)$ -sample of B_1 . At stage i , we will compute a $(\nu(r^{2^i}), \nu(r^{2^i})/r^{2^i})$ -sample of B_i . By induction, we can show that $|B_i| = n/r^{2^i-1}$. Now, from Lemma 14, we can show that B_i is a $(\prod_i \nu(r^{2^i}), \prod_i \nu(r^{2^i})/r)$ -sample of X . We continue this process until $B_j \leq n^{3/4}$ for the first time, call this subset D . Clearly $j < \log \log n$ and from $r \leq n^{1/4}$ it follows that $n^{1/4} \leq |D| \leq n^{3/4}$. algorithm. Thus we can ensure that D is a $(\prod_i \nu(r^{2^i}), \prod_i \nu(r^{2^i})/r)$ -sample of X . Since $(1+x) \leq e^x$, it follows that $\prod_i (1 + \frac{1}{2^i}) = O(1)$. So D is a $(c', c'/r)$ sample of X , for some constant c' . Suppose $|D| = q$, then choose r equally spaced elements from D and call this set S . We claim that S is a set of r elements which approximately splits X with expansion factor \sqrt{r} . To see this, note that between two consecutive elements of S there are q/r elements of D . Since D is a $(c', c'/r)$ sample of X , it follows (from the definition of an (m, λ) -sample) that there are a maximum of $c'(n/q)(q/r) + c'(1/r)n$ elements of X . Substituting $c'' = c'^2$ the number of elements can be bounded by $O(\sqrt{c''}n/r)$. For $r \geq c''$, this is less than $\frac{n}{\sqrt{r}}$. This completes the proof of Lemma 1. Note that $C = c''$ in the statement of Lemma 1.

Remark The running time of the above algorithm for finding approximate splitters can be improved to $O(\log \log n)$ time by noting that only the first group sampling step takes $O(\log \log n)$ and $O(1)$ thereafter by exploiting processor advantage carefully. This was pointed out by one of the reviewers.