Linear Programming on the Reconfigurable Mesh and the CREW PRAM

Paulina Węgrowicz
B.Sc., McGill University, 1987

School of Computer Science
McGill University

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

February 3, 1991

Abstract

This thesis presents a new parallel algorithm for solving the linear programming problem in R^d for the reconfigurable mesh architecture and for the CREW PRAM model. The algorithm is based on the sequential technique discovered independently by Megiddo [Meg83, Meg84] and by Dyer [Dye84, Dye86], which gives a linear time algorithm, in n, the number of constraints, to solve the linear programming problem in d variables, when d is fixed. The parallel algorithm runs in $O(\log^3 n)$ time in R^2, $O(n^{1/3}\log^3 n)$ time in R^3 and in $O(n^{1/2})$ time in R^d on the reconfigurable mesh of size n. A simplified version of the same algorithm runs in $O(\log^d n)$ time on the CREW PRAM. The $o(n^{1/2})$ running times achieved by the parallel linear programming algorithm in R^2 and R^3 are due to a novel selection algorithm, which is also presented in this thesis. The selection algorithm runs in $O(\log^3 n)$ time on the reconfigurable mesh. As is the case with the sequential technique, it will be shown that the parallel technique can be applied towards solving other problems such as linear separability, circular separability, digital disk and the Euclidean one-center problem, and can be extended to solve quadratic programming problems, in particular finding the smallest circle separating two sets of points.

Résumé

Cette thèse présente un nouvel algorithme permettant de resoudre le problème de programmation linéaire en R^d sur une architecture à maille reconfigurable et pour le modèle CREW PRAM. L'algorithme est basé sur la technique séquentielle découverte par Megiddo [Meg83, Meg84] et Dyer [Dye84, Dye86]. Cette technique permet de résoudre le problème de programmation linéaire à d variables, où d est fixe, et ce en un temps linéaire par rapport au nombre n de contraintes. L'algorithme parallèle s'exécute en un temps $O(\log^4 n)$ en R^2, $O(n^{1/3}\log^3 n)$ en R^3 et $O(n^{1/2})$ en R^d sur une maille reconfigurable de taille n. Une version simplifiée du même algorithme s'exécute en temps $O(\log^d n)$ sur une CREW PRAM. Les temps d'exécution de compléxité $o(n^{1/2})$ de l'algorithme de programmation linéaire parallele sont réalisés grâce à un nouvel algorithme de sélection présenté dans cette these. Cet algorithme s'exécute en temps $O(\log^3 n)$ sur une maille reconfigurable. Tout comme pour la technique séquentielle, il sera montré que la technique parallèle peut etre appliquée à d'autres problèmes, tels que la séparabilité linéaire, le cercle englobant minimal, le disque numérique, la programmation quadratique et le problème euclidien à un centre.

Acknowledgements

I would like to thank my advisor Dr Hossam ElGindy for introducing me to the field of parallel algorithms and providing inspiration and encouragement throughout my research and during the writing of this thesis. I would also like to thank my husband Michael for his patience, understanding and assistance during the last year.

Contents

List of Figures

Chapter 1

Introduction

The linear programming problem is that of minimizing (or maximizing) a linear function subject to a finite number of linear constraints, where the linear constraints are either linear equations or linear inequalities [Chv83] Many problems in production management, economics, network analysis and computational geometry can be formulated as linear programming problems The classical sequential approach for solving a linear programming problem is the Simplex method [Dan63], which unfortunately has been shown to have an exponential worst case running time [Kle72], but nonetheless provides an excellent approach for most practical cases Recently, a new sequential linear programming algorithm was developed with running time linear in the number of constraints when the dimension is fixed [Meg83, Dye84, Meg84, Dye86]

The desire to further reduce the running time for solving problems, such as the linear programming problem, beyond what can be achieved by sequential algorithms running on single processor architectures, has generated great interest in parallel algorithms which can exploit the advantages of multi-processor architectures The difficulty in developing parallel algorithms is to efficiently perform computation in parallel while successfully avoiding the problem of resource contention Two general models of parallel computation are usually considered when designing parallel algorithms, one in which the processors share a common memory and one in which the memory is distributed among the processors [Pre79]

The concurrent read exclusive write parallel random access machine (CREW PRAM) is an example of the shared memory model All processors can simultaneously access the memory as long as no two attempt to write to the same memory location simultaneously

This provides for an essentially unconstrained exchange of data between processors. In contrast, the reconfigurable mesh is an example of the distributed memory model. The processors are interconnected in a network, with each processor having a relatively small (constant size) local memory. Not all processors can communicate directly thus constraining the flow of data and increasing the running time of many algorithms. It should be noted that the reconfigurable mesh is a practical example that is well suited to current technology, whereas the CREW PRAM is an idealized example which remains technologically impossible to implement [Pre79].

This thesis proposes a new parallel algorithm which will allow an n constraint linear programming problem in 2, 3 and d dimensions to be solved in $O(\log^4 n)$, $O(n^{1/2}\log^4 n)$ and $O(n^{1/2})$ time respectively on a reconfigurable mesh of size n, with the constant of proportionality growing exponentially with dimension. The algorithm makes use of a novel selection algorithm, also presented in this thesis, which realizes an $O(\log^4 n)$ running time. It is assumed that data put on the reconfigurable bus is broadcast in unit time. A simplified version of the parallel linear programming algorithm will also be shown to run in $O(\log^d n)$ time on the CREW PRAM.

Recently, a number of algorithms have been proposed for solving the linear programming problem in parallel, all for a concurrent read concurrent write PRAM (CRCW PRAM) [Den90, Vai90, Alo90]. In addition, there has been a lot of work on parallel solutions to problems related to linear programming such as the convex hull problem, that can lead to a parallel solution to the linear programming problem [Agg88, Dad87, Ata86, Mil88c, Dob88]. If the linear programming problem is transformed into the dual space, a parallel solution may be found using parallel algorithms for computing convex hull [Dob80, Ede87]. Given a linear programming problem with n constraints in d dimensions, each constraint corresponds to a d-dimensional point in the dual space. It suffices to compute the convex hull of the resulting points and test each convex hull point for optimality. Unfortunately, there are as yet no parallel solutions for the convex hull problem in dimensions higher than three, whereas the parallel linear programming algorithm proposed in this thesis works in d dimensions. The selection problem has received considerably more attention and a number of parallel solutions are available for the mesh and the CREW PRAM [Tho77, Sto83, Pra87, Pla89, Col88]. The

parallel selection algorithm, proposed in this thesis, is believed to be the first such algorithm to achieve polylogarithmic running time on any mesh architecture

The thesis is organized into six chapters following this introductory chapter. Chapter 2 begins with a discussion of mesh architectures before presenting the reconfigurable mesh. The algorithms for computing the maximum and the parallel prefix operations are described next, and the chapter ends with a review of the CREW PRAM model Chapter 3 presents the parallel selection algorithm on the reconfigurable mesh Chapter 4 gives a review of the sequential linear programming algorithms in 2 and d dimensions including a discussion of the multi-dimensional search technique Chapter 5 presents the parallel linear programming algorithms on the reconfigurable mesh in 2 and d dimensions with a special case when $d = 3$, and a simplified version on the CREW PRAM Chapter 6 discusses applications of the parallel algorithm in solving other problems including linear separability, circular separability, digital disk, Euclidean one-center problem and quadratic programming. Chapter 7 concludes this thesis

Chapter 2

The reconfigurable mesh architecture and the CREW PRAM model

This chapter gives an introduction to the basic characteristics of mesh architectures and presents a detailed description of the reconfigurable mesh architecture. Algorithms for two abstract data movement operations, maximum and parallel prefix are described, in order to provide an introduction to designing algorithms for this architecture. Lastly, a brief presentation of the CREW PRAM model is given.

2.1 Basic characteristics of mesh architectures

A mesh [Tho77], a mesh with row and column buses [Pra87] and the reconfigurable mesh [Mil88a] are all interconnection networks with processing elements (PE's) arranged on a two-dimensional grid. Their PE's have similar computational capabilities and the ability to exchange data with their neighbours through local communication links. The N processors of a mesh of size N are placed at the intersections of a two-dimensional square grid of size $N^{1/2} \times N^{1/2}$. Each processor is connected to its four neighbours (if they exist) through unit time communication links. The *communication diameter* [Mil88c], defined as the maximum of the minimum distance (number of communication links) between any two processors in the network, is of $O(N^{1/2})$. This can be seen by computing the distance between processors in opposite corners of the mesh, which is $2(N^{1/2} - 1)$. Thus, for any problem for which a processor in one corner of the mesh requires data from a processor in the opposite corner, a lower bound on the running time is $\Omega(N^{1/2})$.

The above observation can be used to show that many problems require $\Omega(N^{1/2})$ time to be solved on the mesh. In particular consider finding the maximum of a set $S = \{a_i\}$, $i = 1, \ldots, n$, $(n \leq N)$ Let the n values be distributed in no particular order, one value per processor, and suppose that the answer (the maximum of the n values) is to arrive at the top left-most processor of the mesh. If the distribution of the data were such that the maximum value $a_{max} = \max\{a_i,\ i = 1, \ldots, n\}$ was originally at the bottom right-most processor, then it must have traveled for $\Omega(N^{1/2})$ time in order to arrive at its destination processor. Since this argument applies for any choice of destination processor, in the worst case finding the maximum will take $\Omega(N^{1/2})$ time.

The above reasoning cannot be applied when additional communication channels, in the form of buses, are added to the mesh. They allow a piece of data to be sent over a long distance much faster than using local neighbour-to-neighbour connections. Such channels may be in the form of row and column buses [Pra87], that is all processors in each row, and similarly in each column, are connected to a bus. One piece of data can be put on the bus and read by any of the processors connected to that bus. If it is assumed that this takes unit time, then a value can be exchanged between any two processors in at most two time units, by using one row and one column broadcast The time required to find the maximum is reduced to $\Theta(n^{1/6})$ as shown in [Pra87]

Even though the communication diameter, when row and column buses are present, may no longer be the limiting factor for problems such as finding the maximum, another difficulty arises. Many algorithms designed for parallel models of computation are based on the divide-and-conquer technique [Aho83] where the original problem is subdivided into a number of subproblems of smaller size, each of which is solved recursively and their solutions are combined to give solution of the original problem. Examples of this technique applied in parallel computation include the algorithm to find the maximum by Miller et al [Mil88a] or the algorithm to compute the convex hull of an ordered set of points on the reconfigurable mesh due to ElGindy [ElG90] For such algorithms it is desirable that the available resources can be subdivided so that recursion can be applied consistently However in the case of row and column buses, there is always the same number of them available $(2N^{1/2})$ to be shared independently of the number of subproblems created.

2.2 The reconfigurable mesh architecture

The reconfigurable mesh architecture [Mil88a] combines the advantages of the mesh with the power and flexibility of a dynamically reconfigurable bus structure. A reconfigurable mesh of size N consists of a two-dimensional array of N processors arranged on a $N^{1/2} \times N^{1/2}$ rectilinear grid, overlaid by a reconfigurable broadcast bus of the same shape (Figure 2.1). Each processor has a fixed number of registers with $O(\log N)$ bits each, on which it can perform standard arithmetic and logic operations, where it is assumed that each such operation takes $O(1)$time. Each processor $P_{i,j}$ has stored in its registers its row and column indexes i and j, with $i, j \in \left[0, \ldots, N^{1/2} - 1\right]$, where for simplicity it is assumed that $N = 4^k$, for some positive integer k. Each processor is connected by local links to its neighbours: $P_{i,j}$ is connected to $P_{i\pm1,j\pm1}$, if they exist, with $i, j \in \left[0, \ldots, N^{1/2} - 1\right]$, and can send and receive data through these links.

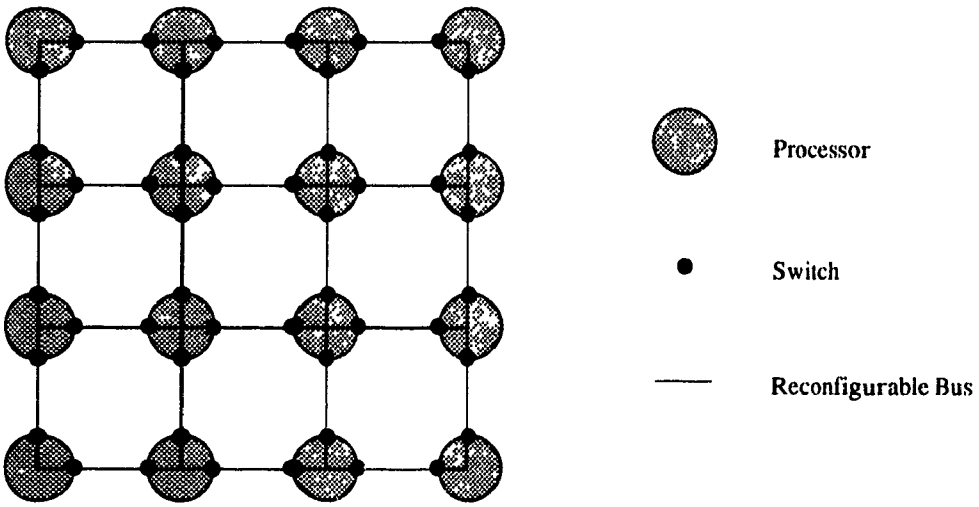

Figure 2.1: The reconfigurable mesh architecture

In addition to being indexed by row and column numbers, processors can also be indexed by a chosen ordering scheme which represents a one-to-one mapping from $\{0, 1, \ldots, N^{1/2} - 1\} \times \{0, 1, \ldots, N^{1/2} - 1\}$ onto $\{0, 1, \ldots, N - 1\}$ [Tho77, Mil88c]. Some common ordering schemes are illustrated in Figure 2.2. The *row-major* ordering is obtained by numbering processors in each row left to right beginning with row 0 and ending with row $N^{1/2} - 1$. This is equivalent to the mapping $k = j + iN^{1/2}$, where i is the row number and j

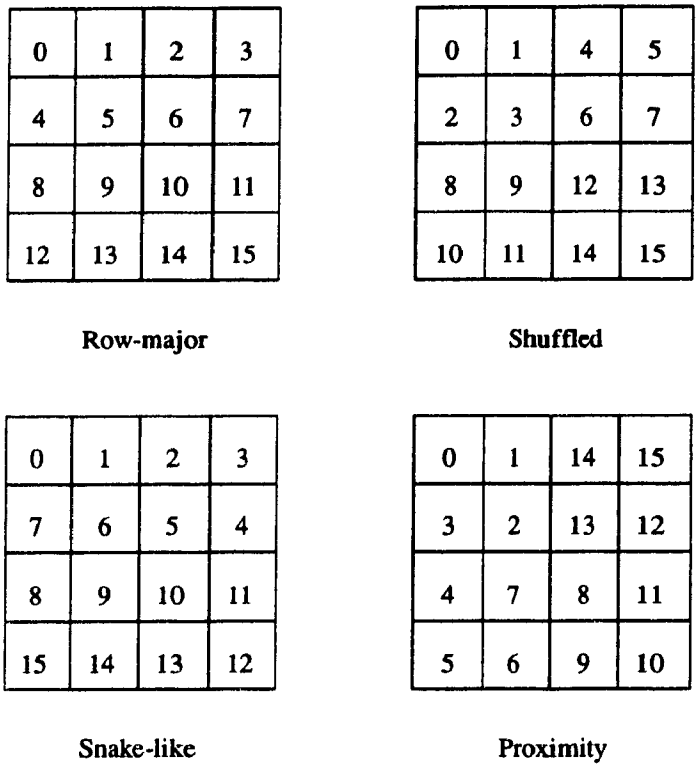

Figure 2.2: Indexing schemes for the processors of a mesh [Mil88c].

is the column number of a given processor. The *shuffled row-major* ordering is obtained by shuffling the binary representation of the row-major index, that is "abcdefgh" becomes "aebfcgdh". This ordering has the property that the first $N/4$ processors form the first quadrant, the second $N/4$ processors form the second quadrant and so on, with this property holding recursively in each quadrant. The *snake-like* ordering is a variation of the row major ordering obtained by reversing the ordering in the odd rows. This gives the property that processors with consecutive indices are adjacent on the mesh, as in Figure 2.2. The *proximity* ordering combines the properties of the shuffled row-major and the snake-like orderings. The proximity index of a processor can be computed in $O(\log n)$ time by that processor based on its row and column indices.

The snake-like ordering will be used throughout this thesis. Each processor can easily compute its snake-like ordering index from its row and column indices and vice versa, but it is convenient to have both stored and available. Therefore, each processor of the mesh will contain a register initialized to represent its index in the the snake-like ordering.

Each processor is also connected to the broadcast bus through four locally controllable

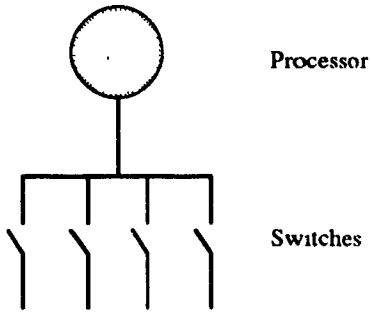

Figure 2.3: Connection of a processor to the reconfigurable bus through switches

switches (three for boundary and two for corner processors), as shown in Figure 2.3. Each processor can dynamically set its switches. Any of the switches may be on or off, realizing four-, three-, two-, one-way or no connections to the reconfigurable bus. It is not possible, however, to simultaneously realize two connections between two pairs of switches. By controlling the switches, the bus can be subdivided into independent connected components called subbuses. All processors connected to a subbus, or the whole bus, can simultaneously read a data value from it, but only one processor can write to a subbus at a time [Mil88a]. This is more restrictive than the model in [Mil88c] but is consistent with [Mil88a] where it is also shown that this model of the reconfigurable mesh with exclusive write can simulate, without loss in time, a bus system where multiple identical values may be broadcast simultaneously on the bus or subbus. This is accomplished through a bus-splitting technique, which will be described in the next section.

The processors of the reconfigurable mesh operate synchronously in single instruction multiple data (SIMD) mode. That is, at each time unit all processors perform the same instruction, but each takes as operands the particular data stored in its registers. Each PE can perform a number of different primitive operations in unit time:

- carry-out arithmetic and logic operations on the contents of its registers,
- send or receive data from its neighbours through local communication links,
- set any of its four switches,
- send or receive data from the bus.

It is assumed, as in [Mil88a], that under the unit time delay model the data put on the reconfigurable bus reaches all processors in constant time.

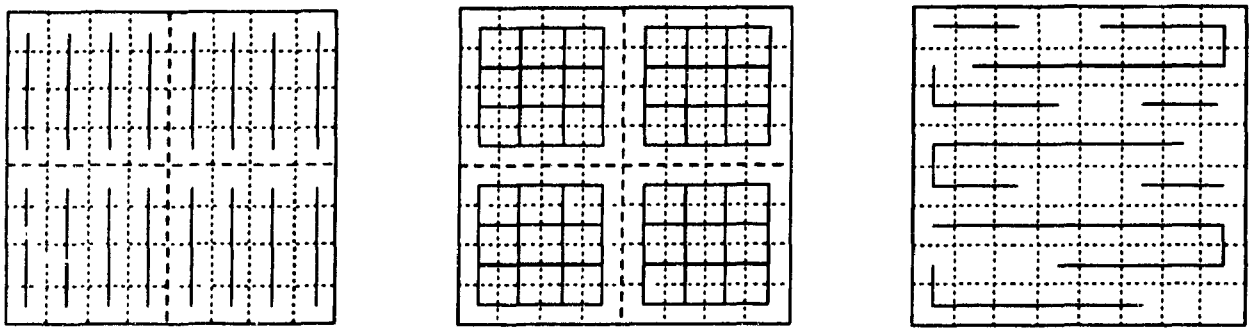

Figure 2.4. A variety of bus configurations.

The distinguishing characteristic of the reconfigurable mesh is the ability to dynamically obtain substructures consisting of groups of processors connected to an independent subbus. Each such substructure can function independently and has the same characteristics as the reconfigurable mesh (except possibly for its size and shape). For example, all the switches can be connected so that one global bus exists with all processors connected to it. Then any processor can broadcast a value to all others in one step By connecting all column switches and disconnecting all row switches another configuration may be obtained with $O(N^{1/2})$ column buses Such buses can be used similarly to static column buses, but have the advantage in that they can be subdivided (also recursively) to give for example $N^{1/2}$, $N^{1/4} \times N^{1/4}$ size meshes with column buses This, of course, cannot be done with the mesh with row and column buses architecture as only a fixed $N^{1/2}$ buses exist there. Other dynamic configurations can be obtained, a few of which are illustrated in Figure 2.4.

By subdividing the bus, a large number of subbuses of some intermediate length or diameter can be created. For example, in creating $N^{1/2}$ groups of $N^{1/4} \times N^{1/4}$ processors with column buses, as in Figure 2.4, $N^{3/4} = N^{1/2} \times N^{1/4}$ column buses were created, each of length $N^{1/4}$ Since each bus can broadcast one piece of data in unit time, as many as $N^{3/4}$ values can move simultaneously over a distance of $N^{1/4}$ each. In general, as much data can be moved as there exist distinct subbuses, but the more subbuses that exist, the shorter they are.

The above observation was essential to developing the selection algorithm of section 3.2 and the parallel linear programming algorithm described in chapter 5. It also leads to an understanding of the limitations of the reconfigurable mesh architecture in solving problems

which require extensive data movement, such as sorting Suppose there are n values distributed one per processor on the reconfigurable mesh of size n, which must be arranged in ascending order in the processors. Imagine cutting the mesh across on a diagonal, which gives a cut the length of the diameter of the mesh. In the worst case the $n/2$ smallest values will be located in processors below the cut, but in order to solve the problem they must be moved to the processors above the cut (analogously for descending order) If the paths crossing the cut through which the $n/2$ values must pass in order to reach their destination processors are counted, it can be seen that there are $2(n^{1/2} - 1)$ mesh links plus the same number of bus links crossing the cut for a total of $O(n^{1/2})$ This is because no matter how the bus is subdivided, the number of crossings at any given cut remains constant Since as many as $O(n)$ values may have to cross and since one value may travel at a time through any link, it must take $\Omega(n^{1/2})$ time for the $O(n)$ values to cross. It will therefore take $\Omega(n^{1/2})$ time, in the worst case, to sort n values on a reconfigurable mesh of size n This time is in fact optimal since the Odd-Even Merge Sort for the mesh architecture [Tho77] can be used to sort on the reconfigurable mesh in $\Theta(n^{1/2})$ time Thus, for some problems, especially those which require extensive data movement, the same asymptotic running time is required on the reconfigurable mesh as on a mesh with no buses It will be shown that better worst-case running times may be achieved for "easier" problems on the reconfigurable mesh

2.3 Data movement operations on the reconfigurable mesh

Abstract data movement operations are commonly used by many parallel algorithms and will be used extensively in the parallel linear programming algorithm Two such operations, the max and the parallel prefix, are presented in detail as they appear in [Mil88a, Mil88b], where it is shown that they can be computed in $O(\log \log n)$ and $O(\log n)$ time respectively, for any set of data with at most n elements on a reconfigurable mesh of size n The algorithms for these operations are adapted from similar, less complex algorithms for the PRAM model The random access read and write operations, also used in the parallel linear programming algorithm, will not be presented here, but are described in [Mil88a]

2.3.1 The max operation

In [Mil88a], an algorithm for the max operation is presented, which computes the maximum of a set of n or fewer values on a reconfigurable mesh of size n. The algorithm is based on a technique called *bus-splitting*, which can be used to compute the maximum of $n^{1/2}$ or fewer values, using a reconfigurable mesh of size n in $O(1)$time. This section will describe the algorithm for the max operation including the bus-splitting technique and review the analysis of its running time

The procedure for computing the maximum of $n^{1/2}$ or fewer values on a reconfigurable mesh of size n is as follows Let the $n^{1/2}$ or fewer data values be distributed in the ith row of the reconfigurable mesh, with the value in processor $P_{i,j}$ called x_j. The values are distributed one per processor If there are fewer than $n^{1/2}$ values, some $P_{i,j}$ will store $-\infty$ The first step is to obtain all pairs of data values, with the pair x_i and x_j stored in $P_{i,j}$. To do this, first form column buses and broadcast x_j to all processors in column j, for all columns in parallel. Then form row buses and broadcast x_i from $P_{i,i}$ to all processors in the ith row, for all rows in parallel Now, all processors can simultaneously perform the comparison $x_i \geq x_j$ and store its result (0 if true and 1 otherwise). The column index (or indices if multiple maxima) of the column in which all comparisons resulted in a 0 corresponds to the maximum value of the data Since only one processor can broadcast to the bus at a time, the bus is divided into column buses and then each column bus is split into segments by having any processor containing a 1 disconnect its switch to the processor below Then each processor containing a 1 will broadcast 1 on its subbus and the processor in the top row will read the top-most 1 in its column, if it exists. Next a row bus is formed in the top row, and this bus is subsequently split into segments by having each processor containing a 0 disconnect the switch to its right. These processors now broadcast their index and $P_{0,0}$ can read the index k of the left-most occurrence of the maximum Now a global bus can be formed, and the maximum value can be broadcast from $P_{0,k}$ to all processors

Since all of the above steps take constant time and since any subbus has only one value broadcast on it at a time, as required by the model, the above procedure can find the maximum of $n^{1/2}$ values distributed in one row of the reconfigurable mesh in $O(1)$ time.

Incorporating the above procedure into an algorithm for the max operation leads to the

following $O(\log\log n)$ time algorithm on the reconfigurable mesh [Mil88a]. Assume there are n or fewer values distributed one per processor on a reconfigurable mesh of size n. Since maximum is an associative operation, the n values can be grouped into smaller sets, the max operation can be applied to these sets and then the maximum of the group maxima can be computed to give the final result. Applying this idea recursively gives the following algorithm.

- Divide the mesh together with the bus into $n^{1/2}$ groups of $n^{1/4} \times n^{1/4}$ size. Each group contains at most $n^{1/2}$ values.
- Compute the max of each group recursively.
- Arrange the resulting $n^{1/2}$ numbers in the top row of the mesh using subbuses and apply the algorithm to compute the max of (at most) $n^{1/2}$ values with n processors.

The running time $t(n)$ of this algorithm can be expressed by the following recurrence relation

$$\begin{cases} t(n) = t(n^{1/2}) + O(1) \\ t(1) = O(1) \end{cases}$$

which, when expanded, gives $t(n) = O(\log\log n)$.

The algorithm takes advantage of the ability to subdivide the bus, which results in a vast improvement in running time over what is possible on other types of meshes. In fact, it is even better than the algorithm for CREW PRAM and attains the lower bound of $\Omega(\log\log n)$, when the number of processors is equal to the size of the data set for any multi-processor computer with binary comparisons as the primitive operation [Val75].

The analysis makes it clear, that efficiency is lost when the number of available processors is strictly greater than the size of the set on which the maximum is being computed. Of course it is always desirable to have these equal, but this may not be possible when computing the maximum is part of an algorithm which at each stage eliminates some arbitrarily distributed subset of the data. In this case, both the size of the remaining problem and the distribution among the processors of the remaining data are initially unknown.

2.3.2 The parallel prefix operation

Another important and useful operation which will be employed in the linear programming algorithm is parallel prefix. It can be used to sum values, broadcast data or count and number active processors. Miller et al [Mil88a] describe in detail how to compute this operation on the reconfigurable mesh with processors in row-major ordering in optimal $O(\log n)$ time. Their algorithm will now be presented with a slight modification in order that it comply with the snake-like indexing scheme adopted in this thesis.

Assume the n values of a set $S = \{a_i\}$ are distributed one per processor on a reconfigurable mesh of size n, with processor P_i containing a_i $(0 \leq i \leq n-1)$ and a unit time binary associative operation $\cdot$. At the end, each processor P_i will contain $a_0 \cdot a_1 \cdot \ldots \cdot a_i$. The idea of the algorithm is to compute a partial parallel prefix in each row of the mesh, then compute the row-wise prefix solutions from the partial values available in the last processor of each row, in the snake-like ordering, and lastly updating the row entries with the row-wise prefix of the previous row. The algorithm is as follows:

- For $i = 1$ to $\log_2 n^{1/2}$
  - For all rows in parallel, form disjoint row subbuses of length 2^i, thus grouping consecutive processors with each group starting at $l \cdot 2^i$, $\left(l = 0, \ldots, \frac{\log_2 n^{1/2}}{2^i} - 1\right)$. Let the processors in each group of size $k = 2^i$ be $P_0, \ldots, P_{k-1}$. Let $P_{k/2}$ (in each group) broadcast its value on its subbus and all processors P_j, $k/2 < j \leq k$ perform $a_j \leftarrow a_j \cdot a_{k/2}$.

Now, each processor will store the prefix restricted to its row, in particular, the last processor in each row will hold the "total" prefix of its row.

- Perform the above procedure on the values in the last processor in each row. Because of the snake-like ordering, the last processor of a row is not adjacent to the last processor of the following row. Using row buses, copy, for all odd columns, the value from the last processor to the first processor in that row. Now all row prefixes are located in the last column of the mesh and by performing the same procedure as in the first step, but only on this one column (not all rows), the row-wise prefixes are computed.

- It remains to update all entries in each row with the row-wise prefix of the previous row. Again, using row buses, the right-most processor in each row will broadcast the row-wise prefix of the previous row, obtained from the processor above it. All processors in each row will then update their value, with the one being broadcast, to give the final prefix.

The first and second steps of this procedure take $O(\log n)$ time each and the last step takes $O(1)$ time for a total of $O(\log n)$ running time for the parallel prefix operation on the reconfigurable mesh.

2.4 The CREW PRAM model

Unlike the reconfigurable mesh interconnection network, the CREW PRAM (concurrent read, exclusive write parallel random access machine) is an idealized model of computation. It consist of a large number of processors connected to a common memory. Any number of processors can read or write to any of the memory locations in unit time but no two processors may write to the same location simultaneously. Because processors can effectively communicate through storing and accessing information in memory locations and since many such exchanges can occur simultaneously, the problem of limited availability of paths between processors common to most interconnection networks does not arise in the CREW PRAM model.

The MAX and parallel prefix data movement operations are of interest as they will be used in the linear programming algorithm. Given n data values, they can be computed in $O(\log n)$ time using n processors [Val75, Lad80]. It is surprising to see that a faster, $O(\log \log n)$ algorithm exists for computing the maximum on the reconfigurable mesh, due to the bus splitting technique which allows the max of $n^{1/2}$ values to be computed in constant time. This is similar to the arguments given in [Coo82], namely, that processors can communicate information not only by writing values, but also by not writing. Furthermore, the splitting of the bus allows large groups of processors to communicate in this fashion. In this respect, the capabilities of the reconfigurable mesh are similar to those of the comparison model of Valiant [Val75], except of course for the limited number of interconnections between processors.

Chapter 3

The selection algorithm

Many existing parallel algorithms are based on the divide-and-conquer approach where a problem is subdivided into smaller size subproblems, each of which is then solved separately. In order to subdivide the problem, it is often required to subdivide the data set $S = \{a_i\}$ in such a way that one subproblem will have all the data values less than a_k and the other one values greater than a_k, where a_k is itself a chosen data value, the kth smallest one in the set S. Although the linear programming algorithm is not based on the divide-and-conquer paradigm, but rather on a prune-and-search approach, it relies on a subdivision of data similar to the one described above, which requires finding a_k for a chosen value of k.

In this chapter an algorithm is presented, with $O(\log^3 n)$ running time, for solving the selection problem on the reconfigurable mesh architecture. It is also shown that a splitter can be obtained on the reconfigurable mesh after only one iteration of the selection algorithm, that is in $O(\log^2 n)$ time. Finally, selection on the CREW PRAM is discussed.

3.1 The selection problem

Finding the kth smallest element of an ordered set S of n elements, where the order is not known, is called the *selection problem*. When $k = \lfloor \frac{n}{2} \rfloor$, the kth element is called the median. One way of finding the kth smallest element is to compute the ranks of all elements and pick the kth element. This can be accomplished in $\Theta(n^{1/2})$ time on a mesh with no broadcasting buses with a sorting algorithm such as the Odd-Even Merge Sort described in [Tho77]. When only one processor is available, sorting requires $\Theta(n \log n)$ time, but the selection problem can be solved in $\Theta(n)$ time with an algorithm by Blum et al [Blu72]. An algorithm, based

on [Blu72], exists for the selection problem which runs in $\Theta\left((n \log n)^{1/3}\right)$ time on a mesh with a single broadcast bus [Sto83]. This is reduced even further on a mesh with multiple broadcasting, where an algorithm is available which runs in $O\left(n^{1.6}(\log n)^{2.3}\right)$ [Pra87]. For the reconfigurable mesh, an algorithm can be implemented, based on the ideas in [Blu?], which runs in $O(n^{\epsilon})$ time, where ϵ is a chosen constant, such that $0 < \epsilon \leq \frac{1}{2}$. The running time of this algorithm is bounded below by the number of recursive calls made throughout its execution. Since two consecutive recursive calls are required at each level of recursion, this algorithm would not result in a polylogarithmic running time.

Another sequential method due to Munro and Paterson [Mun80] will be shown in the next section to lead to a parallel algorithm with running time $O(\log^3 n)$. Their method, designed to select from a file stored on a read-only tape with limited amount of internal storage Q available for computation, runs in $O\left(n\left(\frac{\log n}{\log Q}+1\right)\right)$ time. Two values a_u and a_v are chosen from $S = \{a_i\}$ to form a *filter* -- an interval which is known to contain the kth element. Initially a_u and a_v are the minimum and maximum elements of S respectively. On each pass, elements within this interval are used to form a *sample* from which new values for a "narrower" filter, containing fewer elements, will be chosen. A sample is constructed recursively from a *population*—the remaining active values (those within the filters). For a fixed s, an s-sample at level i is a sorted set of s elements chosen from a population of $s2^i$ elements. At level 0 (the bottom) it is just the whole population ($s2^0$ elements) in sorted order. An s-sample at level $i+1$ is formed by taking two samples at level i, each from half of the population $s2^{i+1}$. The level i samples are thinned by removing every second element of each sample with the remaining elements merged to form the $i+1$ level sample. At any iteration of the algorithm, a sample at level r is taken with the relationship $n' \leq s2^r$ so that all remaining n' elements are in the population. From this sample a new filter is chosen with a_u being the $\lceil \frac{k}{2^r} \rceil - r$ smallest element in the sample and a_v being the $\lfloor \frac{k}{2^r} \rfloor$ smallest one. It is shown in [Mun80] that at most $(2r+1)2^r$ candidates remain between the new filters.

Frederickson [Fre83] adapted this sequential algorithm to give parallel algorithms for the ring, the mesh and the complete binary tree, with a modified sampling technique in order to reduce the number of messages passed between processors on lower levels of recursion. These run in $O(n)$, $O(n^{1/2})$and $O(\log^3 n)$ times respectively.

3.2 The selection algorithm on the reconfigurable mesh

It will now be shown that the original, sequential technique by Munro and Paterson [Mun80] can lead to a selection algorithm for the reconfigurable mesh architecture with $O(\log^3 n)$ running time. Given a set S of n or fewer elements, the objective is to find the kth smallest one. The elements are distributed in no particular order, one per processor of a reconfigurable mesh of size n. Call the processors holding an element "active" and the empty ones "inactive". As elements are eliminated as possible candidates for the kth element, more processors become inactive. By connecting the bus switches to previous and next processors in the snake-like ordering, the inactive processors can serve as constant time communication bridges between consecutive but not adjacent active processors. This eliminates the need to compress the data after each iteration. It also allows the reconfigurable mesh to be viewed as a linear array of processor where any active processor can communicate with the previous and the next active processors in constant time, as if the active elements were always compressed.

The algorithm will proceed, executing the following steps, until only a constant number of candidates remain and the problem can be solved directly.

Procedure Select(n, k, S)

Step 1 Number the active processors by performing parallel prefix operation as addition with active processors holding a 1 and inactive ones a 0. Form communication bridges between non-adjacent active processors.

Step 2 Compute the sample by calling the procedure Sample($4 \log n$, n, S).

Step 3 Choose the new filter values and broadcast them to all processors. Perform parallel prefix again with values less than a_u holding a 1 to compute l, the number of elements smaller than a_u which will no longer be active. Mark as inactive all elements outside the new filters. Perform parallel prefix again to compute the number of remaining active elements n' comprising S'.

Step 4 If only a constant number of candidates remain, sort them and pick the kth one, otherwise call Select(n', $k - l$, S').

Let A, the data in the active processors, be indexed $a_1, \ldots, a_n$, as computed by the parallel prefix in step 1. Let s be the number of elements in the sample and t the number of elements in the population. Then step 2 is performed by the following procedure.

Procedure Sample(s, t, A)

Step 1 If $s = t$ sort the active elements with all other processors acting as bus bridges and exit.

Step 2 Divide the data set A into two groups $A_1 = \{a_1, \ldots a_{t/2}\}$ and $A_2 = \{a_{t/2+1}, \ldots, a_t\}$. Compute the sample of each group recursively, for both groups in parallel, by calling Sample(s, $t/2$, A_1) and Sample(s, $t/2$, A_2).

Step 3 Using only every second element in each recursively computed sample, merge the two samples by sorting, while all other processors act as bridges.

At the bottom of the recursion in procedure Sample (step 1), it is required to sort $s = 4 \log n$ values. Since these values can be viewed as being held in a linear array of s processors (thanks to the bus bridges), they can be sorted in $O(s) = O(\log n)$ time as in [Akl85, Knu73]. The same sorting algorithm can be used to merge the two thinned samples into one also in $O(s)$ time. Note that the sample size s was chosen as to sort groups of data that are as small as possible, yet be able to show that the size of the problem is diminishing after each iteration. The total time to compute the sample at level r, $r = \log n/s$, can be expressed as

$$\begin{cases} t(r) = t(r-1) + O(s) \\ t(0) = O(s) \end{cases}$$

which gives $t(r) = O(s^2) = O(\log^2 n)$.

Going back to the algorithm Select, step 1 requires $O(\log n)$ time. For step 3, consider the jth largest element in a sample at level i. Let L_{ij} and M_{ij} respectively be the least number and most number of elements, from the corresponding population, which can be greater than the jth largest element in the sample. Lemma 2 in [Mun80] states that

$$L_{ij} = j2^i - 1 \quad \text{and} \quad M_{ij} = (i + j - 1)2^i$$

In choosing the new filter, it must be ensured that the kth element is one of the filter values

or lies between them, that is

$$k-1 \geq M_{ru} = (r+u-1)2^r$$

and

$$k-1 \leq L_{rv} = v2^r - 1$$

The choice for the new filter will therefore be $u = \lceil \frac{k}{2^r} \rceil - r$ and $v = \lceil \frac{k}{2^r} \rceil$. Broadcasting these two values over the whole mesh and comparing with the data in the active processors, will allow the elimination of all values lying outside the new filters as candidates for the kth element, thus completing step 3 in constant time.

The remaining number of elements is at most

$$\begin{aligned} M_{rv} - L_{ru} - 1 &= (2r-1)2^r \\ &= \left(2\log\frac{n}{s} - 1\right)\frac{n}{s} \\ &= \left[2\left(\log n - \log(4\log n)\right) - 1\right]\frac{n}{4\log n} \\ &= \frac{n}{2} + \frac{n\log\log n}{2\log n} - \frac{5n}{4\log n} \\ &= n\left(\frac{1}{2} + \frac{\log\log n}{2\log n} - \frac{5}{4\log n}\right) \end{aligned}$$

and since $\frac{1}{4} \geq \frac{2\log\log n - 5}{4\log n}$, or equivalently $\log n \geq 2\log\log n - 5$ for all n, at most $\frac{3}{4}n$ elements remain.

The running time of algorithm Select is

$$\begin{cases} t(n) = t(\frac{3}{4}n) + O(\log^2 n) \\ t(4\log n) = O(\log n) \end{cases}$$

which gives $t(n) = (O\log^3 n)$.

Note that no stack is required. To keep track of the recursion only two registers per processor are needed One register stores the current level of recursion, which is the same for all processors and one which stores the level at which the processor may become active again

3.3 Finding a splitter on the reconfigurable mesh

In many instances it is not required that the data set S be divided in an exact manner. It suffices to find an element p of S, not of exact rank, but rather one for which it is known that at least a constant proportion α, $0 < \alpha < 1/2$, of the elements of S are greater than p and at least αn are smaller than p. Such an element will be called an *α-splitter*. It will be shown that a splitter can be found among the elements of the r level s-sample for any chosen $\alpha < \frac{3}{8}$ and hence can be obtained in $O(\log^2 n)$ time.

The objective is to find an integer j $(1 \leq j \leq |s|)$ such that

$$L_{rj} > \alpha n \qquad \text{and} \qquad M_{rj} < (1-\alpha)n$$

for any given α, $\left(0 < \alpha < \frac{3}{8}\right)$. Taking a sample at level r and recalling that $r = \log \frac{n}{s}$, gives

$$L_{rj} = j2^r - 1 = j\frac{n}{s} - 1 \qquad \text{and} \qquad M_{rj} = (r + j - 1)2^r = \left(\log \frac{n}{s} + j - 1\right)\frac{n}{s}.$$

Combining the two gives

$$j > (\alpha n + 1)\frac{s}{n} \qquad \text{and} \qquad j < (1-\alpha)s + 1 - \log \frac{n}{s}.$$

In order to guarantee that an integer value for j can be found, it is required that

$$(\alpha n + 1)\frac{s}{n} < j - 1 \qquad \text{and} \qquad j + 1 < (1-\alpha)s + 1 - \log \frac{n}{s}$$

or

$$\begin{aligned}
(\alpha n + 1)\frac{s}{n} &< (1-\alpha)s - \log \frac{n}{s} \\
(\alpha n + 1)s + n &< (1-\alpha)ns - n \log \frac{n}{s} \\
(\alpha n + 1)4\log n + n &< (1-\alpha)n\,4\log n - n \log \frac{n}{4\log n} \\
4\alpha n \log n + 4\log n + n &< 4n\log n - 4\alpha n \log n - n(\log n - \log 4 - \log\log n) \\
8\alpha n \log n &< 3n\log n + n\log\log n - 4\log n + n
\end{aligned}$$

So for

$$\alpha < \frac{3n\log n + n\log\log n - 4\log n + n}{8n\log n}$$

which is at least $\frac{3}{8}$, it can be guaranteed that the jth element of a sample, with j taken to be $\lceil (\alpha n + 1)\frac{4\log n}{n} + 1 \rceil$, is an α-splitter of S.

3.4 Selection on the CREW PRAM

A recent result due to Plaxton [Pla89] gives a lower bound on the selection problem of $\Omega((n/p)\log\log p + \log p)$, where n is the number of elements to select from and p the number of processors in the network The result applies to a number of common network models and the CREW PRAM. Plaxton also presents an algorithm which runs in $O((n/p)\log\log p + (T_1 + T_2\log p)\log(n/p))$ time, where T_1 is the time to sort $n = p$ values with p processors and T_2 is the time to perform broadcasting and summing. If the time T_3 required for a given network of p processors to perform selection on $n = p$ values is less than $T_1/\log p$, the running time is reduced to $O((T_2 + T_3)\log p\log(n/p))$. An algorithm due to Cole [Col88], designed for an exclusive-read exclusive-write PRAM model, which runs in $O(n/p + \log p\log^* p)$ time, provides optimal efficiency [Akl85] for $n = \Omega(p\log p\log^* p)$. Since the CREW PRAM is a stronger model, the same result applies. However, for the algorithms presented in this thesis, the number of processors is taken to be equal to n, therefore, the fastest way of solving the selection problem on the CREW PRAM is in $O(\log n)$ time through sorting [Col86].

Chapter 4

The sequential linear programming algorithms

The Simplex method [Dan63] has long been known as an excellent, practical algorithm for solving linear programming problems. Typically, a problem with n constraints and d variables can be solved with the number of iterations proportional to n and increasing very slowly (logarithmically) with d [Chv83]. However, in the worst case, as was demonstrated by the Klee-Minty examples [Kle72], the algorithm visits each vertex of the feasible region, the number of which grows exponentially with dimension and so the number of iterations is also exponential in d. Even in 2 dimensions, the worst case running time of the Simplex method is $O(n^2)$ since at most n iterations may be required and each iteration takes $O(n)$ time as each constraint must be inspected The time for solving a 2-dimensional linear programming problem can be reduced to $O(n \log n)$, by finding the intersection of the n half-planes defined by the n constraints with an algorithm due to Shamos [Sha78]

A novel and ingenious technique discovered independently by Megiddo [Meg83] and by Dyer [Dye84] demonstrated that linear programming can be solved in time proportional to the number of constraints n, in two and three dimensions (worst case time analysis) This technique was extended by Megiddo [Meg84] to an arbitrary number of dimensions and shown to give an algorithm linear in n when the dimension is fixed Megiddo's approach [Meg84] presented a novel multi-dimensional search technique applicable to the linear programming problem, which was improved by Clarkson [Cla86] and both improved and further generalized by Dyer [Dye86], resulting in reduction of the constant of proportionality from doubly exponential to singly exponential in d.

This chapter will present a review of Megiddo's and Dyer's algorithms for solving the linear programming problem in 2 and d dimensions. While not everything in their papers directly relates to linear programming, the concentration will be on the material relevant to the problem being addressed in this thesis. However, a number of the applications from the above papers will be discussed in Chapter 6.

4.1 Linear programming in two dimensions

This section follows the development of the linear programming algorithm in two dimensions as presented in [Meg83, Dye84]. A two-dimensional (two-variable) linear programming problem with n constraints can be stated as

$$\begin{array}{ll} \text{minimize} & ax_1 + bx_2 \\ \text{subject to} & a_i x_1 + b_i x_2 + c_i \leq 0, \quad i = 1, \ldots, n. \end{array}$$

The function $ax_1 + bx_2$ is called the *objective function* and the polygonal region formed by the intersection of all the constraints is called the *feasible region* [Chv83]. The solution, if it exists, to a linear programming problem, is a vertex of the feasible region which minimizes the value of the objective function. The objective of the algorithm is to remove constraints which are guaranteed not to contain a vertex of the feasible region minimizing the objective function, ie an optimal solution, as well as any redundant constraints, until only a small number of constraints remain and the problem can be solved directly. This approach has been called *prune-and-search* [Lee84], since at each stage a part of the problem is eliminated and the search continues within the remaining part.

Applying a linear transformation $y = ax_1 + bx_2$ and $x = x_1$ the linear programming problem can be stated in an equivalent form with the objective function equal to the y coordinate

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & \alpha_i x + \beta_i y + c_i \leq 0, \quad i = 1, \ldots, n, \end{array}$$

where $\alpha_i = a_i - \frac{a}{b} b_i$ and $\beta_i = \frac{b_i}{b}$ Finding an optimal solution is now reduced to finding a minimum value of a piecewise linear convex function of x. This function is implicitly defined by the set of linear constraints Depending on β_i being negative, positive or zero, the constraint set is partitioned into I_1, I_2 and I_3 respectively. From I_3, which contains constraints defined by a line parallel to the y-coordinate, $u_1 = \max\left\{-\frac{c_i}{\alpha_i}, i \in I_3\right\}$ and $u_2 =$

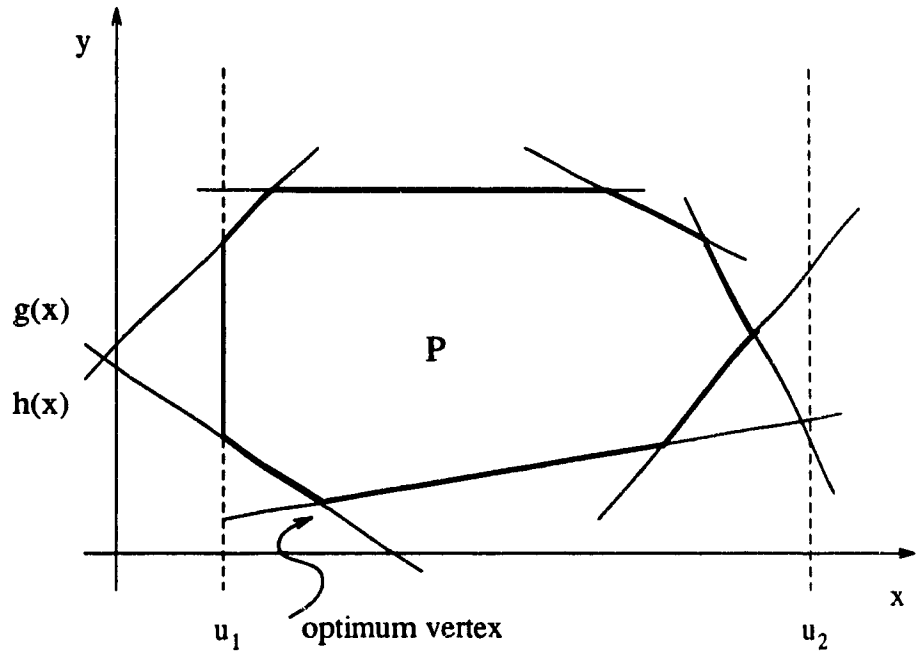

Figure 4.1: The feasible region P defined by a set of linear constraints

$\min\left\{-\frac{c_i}{\alpha_i}, i \in I_3\right\}$ are defined. The intersection of the two inequalities $x \geq u_1$ and $x \leq u_2$ delimits a region in which the optimal solution must lie. Obviously, if $u_1 \not\leq u_2$ then the problem is infeasible.

The transformed problem is illustrated in Figure 4.1, where $g(x) = \max_{i \in I_1}(\delta_i x + \gamma_i)$ and $h(x) = \min_{i \in I_2}(\delta_i x + \gamma_i)$ are convex piecewise linear functions which delimit the feasible region, with $\delta_i = -\frac{\alpha_i}{\beta_i}$ and $\gamma_i = -\frac{c_i}{\beta_i}$. A given value of x is feasible if $h(x) \geq g(x)$, $u_1 \leq x \leq u_2$, and the problem can be stated as

$$\begin{array}{ll} \text{minimize} & g(x) \\ \text{subject to} & g(x) \leq h(x) \\ & u_1 \leq x \leq u_2 \end{array}$$

The algorithm iterates by testing values of x, in a way similar to binary search, to determine if x gives the optimal solution and if not to which side of x the optimal solution may lie. At each iteration either the solution is found or at least a constant proportion of constraints are eliminated as candidates for containing the optimal solution, until the number of constraints is small and the problem can be solved directly

To begin, constraints in I_1 are paired together by taking the ith and $i+1$st constraints, with $i = 1, 3, 5, \ldots, |I_1|$ The same is done with constraints in I_2 If $\delta_i \neq \delta_{i+1}$, then the intersection of the two lines corresponding to the two constraints in each pair is computed

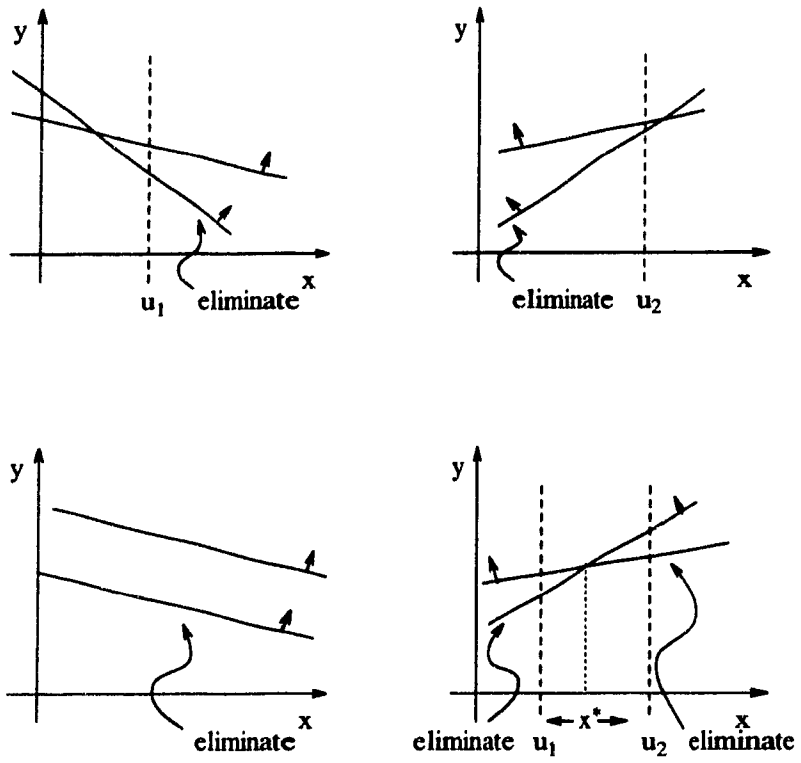

Figure 4.2: Eliminate one constraint in each of the above cases.

to give at most $\frac{n}{2}$ intersection points, which are candidates x for the optimal solution. One constraint can immediately be eliminated from all pairs when either the intersection of the constraints lies outside the interval $[u_1, u_2]$ or the constraints are defined by parallel lines, as illustrated in Figure 4.2 for pairs of constraints in I_1. For the remaining pairs of constraints, find the median x_m of the x-coordinates of their intersection points. This value can now be tested and one of the following conclusions can be drawn: the problem is infeasible, x_m is the optimal solution, or the interval $[u_1, u_2]$ can be reduced to $[u_1, x_m]$ or $[x_m, u_2]$. Such a conclusion can be reached based on the values of $g(x_m)$ and $h(x_m)$ and their slopes to the left and to the right of x_m, which can be evaluated in time proportional to $|I_1|$. For example, if $g(x_m) < h(x_m)$ and the slope of g to the left of x_m is non-positive and to the right is non-negative, then x_m is itself the minimum of g.

If the problem is found to be infeasible, or the optimal solution is found, then the algorithm terminates. Otherwise, at least half of the intersections, as defined by the pairs of constraints, will be outside the new interval $[u_1, x_m]$ or $[x_m, u_2]$ and one constraint per each such pair can be dropped (see Figure 4.2), for a total of at least a quarter of the constraints.

The original problem is in this way reduced in $O(n)$ time to a linear programming problem with at most $\frac{3}{4}n$ constraints. Thus, the overall running time $T(n)$ with $\alpha \quad \frac{1}{4}$ is linear in n since

$$T(n) = T\left(\tfrac{3}{4}n\right) + kn \quad \leq \quad \sum_{i=1}^{\log_{\frac{1}{(1-\alpha)}} n} k(1-\alpha)^{i-1}n$$
$$< \quad \frac{k}{\alpha}n = O(n). \qquad (11)$$

This worst case time analysis relies on the ability to find the median of n values in $O(n)$ time [Blu72].

4.2 Linear programming in d dimensions

The following algorithm was developed by Megiddo [Meg84] to solve linear programming problems in d dimensions (d variables) when d is fixed A d-dimensional linear programming problem can be stated as

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \geq b_i, \quad i = 1, \ldots, n. \end{aligned}$$

Similarly to 2 dimensions, the algorithm repeatedly removes a constant proportion of constraints until the problem can be solved directly by solving a set of d equalities, to obtain the intersection of the remaining tight constraints.

The problem is first transformed to a subspace orthogonal to the direction of the objective function.

$$\begin{aligned} \text{minimize} \quad & x_d \\ \text{subject to} \quad & x_d \geq \sum_{j=1}^{d-1} a_{ij} x_j + b_i, \qquad i \in I_1 \\ & x_d \leq \sum_{j=1}^{d-1} a_{ij} x_j + b_i, \qquad i \in I_2 \\ & \sum_{j=1}^{d-1} a_{ij} x_j + b_i \leq 0, \quad i \in I_3 \end{aligned}$$

with $|I_1| + |I_2| + |I_3| = n$.

Considering a pair of inequalities i, k in the same set, say I_1, two possibilities exist

- If $(a_{i1}, \ldots a_{i,d-1}) = (a_{k1}, \ldots, a_{k,d-1})$, then one of the constraints is redundant and can be dropped
- If $(a_{i1}, \ldots a_{i,d-1}) \neq (a_{k1}, \ldots, a_{k,d-1})$, then $\sum_{j=1}^{d-1} a_{ij}x_j + b_i = \sum_{j=1}^{d-1} a_{kj}x_j + b_k$ is an equation of a $(d-1)$-dimensional hyperplane which divides the space so that on one side of this hyperplane constraint i dominates constraint k, that is $\sum_{j=1}^{d-1} a_{ij}x_j + b_i < \sum_{j=1}^{d-1} a_{kj}x_j + b_k$, and constraint k dominates constraint i on the other side.

As shown by Megiddo, one can test such a $(d-1)$-dimensional hyperplane to determine on which side of it lies the optimal solution (if one exists) and then eliminate the constraint dominating on that side

4.2.1 Testing a hyperplane

The testing of a hyperplane $h = \sum_{j=1}^{d} a_j x_j + b$ can be accomplished recursively by solving at most three $(d-1)$-dimensional linear programming problems with at most n constraints each. Given the original linear programming problem and the hyperplane h consider the same problem with the equation of the hyperplane as an additional constraint. This gives a d-dimensional problem with $n+1$ constraints, but through an elimination of one variable will give a $(d-1)$-dimensional problem with n constraints. If this problem is unbounded, then the original problem is unbounded and the algorithm is finished. Otherwise, a solution is obtained for which it remains to be determined if it is the final solution and, if not, on which side of the hyperplane to continue. This can be determined by solving at most two additional linear programming problems. The details are presented in [Meg84].

4.2.2 The multi-dimensional search technique

Testing a hyperplane h, as described above, will lead to elimination of one constraint. Since this procedure is costly in time, one needs to maximize the information obtained from each test. A scheme which, by testing few hyperplanes, allows relatively many constraints to be removed, was first proposed by Megiddo [Meg84] and subsequently expanded on by Dyer [Dye86] and Clarkson [Cla86]. This section begins with an exposition of the problem and

the required set-up as presented in [Dye86], followed by a review of the different approaches to applying the multi-dimensional search technique taken by Megiddo, Dyer and Clarkson

Given a set of n hyperplanes $h_i(x) = \left\{x \in R^d \quad a_i^T x = b_i\right\}, i \quad 1, \quad , n$ and a point x^*, it is required to determine the position of x^* relative to a fixed proportion p of the hyperplanes The point x^* is not known, however the procedure described in section 4 2 1 can determine the position of x^* relative to any hyperplane h in R^d, that is determine whether $a^T x^* \quad b$, $a^T x^* = b$ or $a^T x^* > b$. This is equivalent to determining the sign of $h(x^*)$, denoted by $sign(h(x^*))$ or simply $sign(h)$, where the sign can be negative, zero or positive An *inquiry* is an evaluation of sign for a given function in R^d When h_i is a constant function ($a_i \quad 0$), $sign(h_i)$ can be determined without any inquiries, that is without having to evaluate it

When $d = 1$, the hyperplanes are of the form $h_i(x) = x_1 + b_i$ If β denotes the median of the b_i and since in one inquiry the sign of $h(x) = x_1 + \beta$ can be determined, then

- if $sign(h(x^*))$ is positive, then the sign of $h_i(x^*)$ is also positive for at least $n/2$ values of i, for which $b_i \geq \beta$,
- if $sign(h(x^*))$ is negative, then the sign of $h_i(x^*)$ is also negative for at least $n/2$ values of i, for which $b_i \leq \beta$,
- if $sign(h(x^*))$ is zero, then the sign of $h_i(x^*)$ is positive for all values of i, for which $b_i > \beta$, negative for all values for which $b_i < \beta$ and zero for all values for which $b_i \quad \beta$

Therefore, the sign of at least half of the h_i is known after one inquiry and of all the h_i after at most $\log n$ inquiries.

When $d \geq 2$, the hyperplanes are paired so that one hyperplane in each pair has a slope greater than the median and the other one has a slope smaller than the median The slope is defined to be the slope of the line $a_{i1}x_1 + a_{i2}x_2 = b_i$, which is the line along which the hyperplane h_i intersects the (x_1, x_2) subspace For each pair of hyperplanes h_i, h_j, auxiliary hyperplanes $h_{ij}^{(1)}$ and $h_{ij}^{(2)}$ are formed so that each has a dimension which is one less than the dimension of h_i and h_j, but in different variables, that is $h_{ij}^{(1)} \quad h_{ij}^{(1)}(0, x_2, x_3, \quad , x_d)$ and $h_{ij}^{(2)} = h_{ij}^{(2)}(x_1, 0, x_3, \ldots, x_d)$. Because the auxiliary hyperplanes are $(d \quad 1)$-dimensional, the search can be applied recursively to the two collections $h^{(1)}$ and $h^{(2)}$ The hyperplanes with

$a_{i1} = 0$ or $a_{i2} = 0$ are excluded from the pairing since they are already $(d-1)$ dimensional. The original hyperplanes can be expressed in terms of the auxiliary hyperplanes

$$\begin{aligned} h_i(x) &= a_{i2} h_{ij}^{(1)}(x) + h_{ij}^{(2)} \quad \text{with } a_{i2} > 0, \\ h_j(x) &= a_{j2} h_{ij}^{(1)}(x) + h_{ij}^{(2)} \quad \text{with } a_{j2} < 0, \end{aligned}$$

to demonstrate that if $sign\left(h_{ij}^{(1)}\right)$ and $sign\left(h_{ij}^{(2)}\right)$ are both known, then the sign of at least one of h_i or h_j can be determined, in particular

- if $sign\left(h_{ij}^{(1)}\right)$ is zero, then $sign(h_i) = sign(h_2) = sign\left(h_{ij}^{(2)}\right)$,
- if $sign\left(h_{ij}^{(1)}\right)$) is positive, then
  - if $h_{ij}^{(2)}(x^*) \geq 0$, then $sign(h_i)$ is positive,
  - if $h_{ij}^{(2)}(x^*) \leq 0$, then $sign(h_j)$ is negative,
- if $sign\left(h_{ij}^{(1)}\right)$) is negative, then
  - if $h_{ij}^{(2)}(x^*) \leq 0$, then $sign(h_i)$ is negative,
  - if $h_{ij}^{(2)}(x^*) \geq 0$, then $sign(h_j)$ is positive,

Then, for each pair $h_{ij}^{(1)}$, $h_{ij}^{(2)}$ for which the search determines the location of the optimum x^*, relative to both auxiliary hyperplanes, the location is known relative to h_i or h_j and so one constraint can be eliminated

Megiddo proposed two different recursive schemes. The first scheme showed that there exist constants $A(d)$ and $B(d)$, $0 < B(d) \leq \frac{1}{2}$, which are independent of n, such that with $A(d)$ inquiries, the position of x^* can be determined relative to at least a proportion $B(d)$ of hyperplanes in R^d, as follows

- Inquire $A(d-1)$ times to obtain the position of x^* relative $B(d-1)\frac{n}{2}$ of $h^{(1)}$, where $\frac{n}{2}$ is the total number of hyperplanes $h^{(1)}$, which is equal to the number of $h^{(2)}$. (To ensure that $\frac{n}{2}$ pairs exist, Megiddo applies a transformation so that at least one of the coefficients a_{i1} and a_{i2} in all constraints is non-zero.)
- For those pairs for which the position of x^* was determined relative to $h^{(1)}$, inquire $A(d-1)$ times again to obtain position of x^* relative to $B(d-1)B(d-1)\frac{n}{2}$ of $h^{(2)}$

In effect, inquiring $A(d) = 2A(d-1)$ times, which gives $A(d) = 2^{d-1}$, results in a proportion $B(d) = \frac{1}{2}(B(d-1))^2$, with $B(d) = 2^{1-2^d}$, of pairs h_i, h_j, such that the position of x^* is known with respect to at least one (both if equal slopes) of h_i or h_j.

This scheme gives the following recurrence for the time it takes to solve a linear programming problem

$$LP_1(n,d) \leq 3 \cdot 2^{d-1} LP_1(n,d-1) + LP_1\left(\left(1-2^{1-2^d}\right)n, d\right) + O(nd)$$

For a constant $C(d) \leq 3 \cdot 2^{2^d+d-2} C(d-1)$, it can be verified that $LP_1(n,d) \leq O(n)$ with the constant of proportionality $C(d) < 2^{2^{d+2}}$.

Megiddo's second scheme recursively finds the position of x^* relative to **all** the auxiliary hyperplanes $h^{(1)}$ and $h^{(2)}$. This is done by solving two $(d-1)$-dimensional search problems with $\frac{n}{2}$ hyperplanes each, one subproblem for all $h^{(1)}$ and the other for all $h^{(2)}$. This gives an outcome relative to half of the original n hyperplanes. It remains to find an outcome for the other half. Let $Q(n,d)$ denote the number of queries required and $T(n,d)$ the additional effort required for pairing the hyperplanes and finding the median, then

$$\begin{cases} Q(n,d) &= \min\left\{n, 2Q\left(\frac{n}{2}, d-1\right) + Q\left(\frac{n}{2}, d\right)\right\} \\ Q(n,1) &= 1 + \lfloor \log_2 n \rfloor \\ Q(1,d) &= 1 \end{cases}$$

which can be solved by a technique in [Mon80] and gives $Q(n,d) = O\left(\log^d n\right)$ with a constant $C(d) = \frac{2^d}{(d-2)!}$.

The additional effort is

$$\begin{cases} T(n,d) &= 2T\left(\frac{n}{2}, d-1\right) + T\left(\frac{n}{2}, d\right) + \Theta(nd) \\ T(n,1) &= n \\ T(1,d) &= 1 \end{cases}$$

which gives $T(n,d) < d2^d n$.

With this approach, an n constraint problem in d dimensions is reduced to $\frac{n}{2}$ constraint problem in d dimensions by solving $O\left((2\log n)^d/(d-2)!\right)$ problems with n constraints in $(d-1)$ dimensions, with additional effort of $O\left(d2^d n\right)$. This leads to the following recurrence for the linear programming problem

$$LP_2(n,d) \leq c\frac{\left(2\log\frac{n}{2}\right)^d}{(d-2)!} LP_2(n,d-1) + LP_2\left(\frac{n}{2}, d\right) + O(d2^d n)$$

For a fixed d, $LP_2(n,d) = O\left(n(\log n)^{d^2}\right)$, with a constant $C(d) < \frac{2^{d^2}}{\prod_{k=1}^{d-2} k!}$.

Megiddo's first approach applies one query at each level of recursion, resulting in a very small proportion of hyperplanes being eliminated and a doubly exponential constant of proportionality for the linear programming problem. In addition, part of the information obtained from the search is lost, since the signs of some of the $h^{(1)}$ are known, but not of their corresponding $h^{(2)}$. The opposite problem arises in Megiddo's second approach, in which queries are applied repeatedly until the signs of all $h^{(1)}$ and $h^{(2)}$ are known. These queries are answered by solving n constraint linear programming problems when some constraints can be eliminated.

Dyer [Dye86], showed that a constant proportion of queries can be answered (not a function of d as in Megiddo's first approach), by continuing to apply the recursion at a given recursive level and removing hyperplanes until the required proportion is eliminated. This leads to an algorithm for linear programming linear in n and with a constant of proportionality singly exponential in d. Dyer's method is a general multi-dimensional search technique, which covers the spectrum between Megiddo's first and second approach.

Let $A(d,q,p)$ be a procedure for the search problem, which takes a d-dimensional set of n hyperplanes and returns the signs of at least pn of the given hyperplanes after making at most q calls to the procedure for testing hyperplanes. Suppose r procedures $A_k(d,q_k,p_k)$ exist, $k = 1,\ldots,r$. By applying A_1, removing a fraction p_k of the h_i, then applying A_2 to the remaining fraction $1-p_k$, and so on, a procedure $A(d,q,p)$ is obtained with

$$p = 1 - \prod_{k=1}^{r}(1-p_k) \quad \text{and} \quad q = \sum_{k=1}^{r} q_k.$$

Dyer chose the following particular approach to applying this idea. Given a procedure $A(d-1,q_{d-1},p_{d-1})$ in R^{d-1}, a procedure $A'\left(d-1, kq_{d-1}, 1-(1-p_{d-1})^k\right)$ can be obtained by applying k times procedure A. Then, by combining two procedures A' and A'' get a procedure $A_1(d,q,p)$ with

$$p = \frac{1}{2}\left(1-(1-p_{d-1})^k\right)\left(1-(1-p_{d-1})^l\right) \quad \text{and} \quad q = (k+l)q_{d-1}.$$

This procedure is called a $[k,l]$ procedure in R^d.

A *scheme* results from applying a list of r $[k_i,l_i]$ procedures, $i = 1,\ldots,r$ in R^d. It

guarantees a proportion

$$p_d = 1 - \prod_{i=1}^{r} \left(1 - \frac{1}{2}\left(1 - (1 - p_{d-1})^{k_i}\right)\left(1 - (1 - p_{d-1})^{l_i}\right)\right) \qquad (1.2)$$

with

$$q_d = \sum_{i=1}^{r}(k_i + l_i)q_{d-1}.$$

inquiries. When a fixed scheme is applied recursively, based on a $A(1, 1, \frac{1}{2})$ procedure it will inquire

$$q_d = \left(\sum_{i=1}^{r}(k_i + l_i)\right)^{d-1}$$

times. Dyer then shows that for some fixed values of $\sum_{i=1}^{r}(k_i + l_i)$ the smallest of which is 9, a non-zero root can be found for $p_d = f(p_{d-1})$, (equation 4.2) In fact, a $[2, 2]$, $[2, 3]$ scheme with $\sum_{i=1}^{r}(k_i + l_i) = 9$ guarantees $p_d \geq \frac{1}{2}$ for all d, with $p_1 = \frac{1}{2}$ If $p_{d-1} \geq \frac{1}{2}$, it can be verified from equation 4.2 that $p_d \geq 1 - \left(1 - \frac{1}{2}\left(\frac{3}{4}^2\right)\right)\left(1 - \frac{1}{2}\frac{3}{4}\frac{3}{8}\right) = 1059/2048 > \frac{1}{2}$ This scheme, therefore, gives a sequence of procedures $A(d, 9^{d-1}, \frac{1}{2})$ and allows the constant of proportionality for the linear programming algorithm to be reduced from $2^{O(2^d)}$ to $O\left(3^{(d+1)^2}\right)$ The time $T(n, d)$ to solve a procedure $A(d, q_d, p_d)$ generated by a scheme $[k_i, l_i]$, $i = 1, \quad , r$ is

$$T(n, d) \leq cT(n, d - 1) + Knd,$$

where $c = \sum_{i=1}^{r}(k_i + l_i)$ Then, if $T(n, 1) \leq Kn$, $T(n, d) = O(c^{d-1}dn)$, with $c = 9$

The time to solve a linear programming problem using the above scheme is expressed by the following recurrence relation

$$LP(n, d) \leq 3 \cdot 3^{2(d-1)}(LP_2(n, d - 1) + Knd) + LP_2\left(\frac{n}{2}, d\right)$$

It can be verified by induction, that $LP_2(n, d) = O\left(3^{(d+1)^2}n\right)$ and so is linear for any fixed d, and in addition for $d = O(\sqrt{\log n})$ is polynomial.

Dyer also observed, that Megiddo's first approach corresponds to a repeated $[1, i]$ scheme with $r = \lceil \log n \rceil$. He then showed that it is sufficient to take $r = 6$ to guarantee $p \quad \frac{1}{2}$ and thus give a better running time than both of Megiddo's approaches

A similar result to Dyer's was also obtained by Clarkson [Cla86], whose algorithm corresponds to applying a $[4, 5]$ scheme and results in a similar improvement for the linear programming problem.

Chapter 5

The parallel linear programming algorithms

This chapter presents the parallel linear programming algorithm for the reconfigurable mesh and the CREW PRAM. The algorithm for 2 dimensions is presented first, followed by the general d-dimensional one. When $d = 2$, the general d-dimensional algorithm achieves the same running time as the 2-dimensional one, however both will be presented as it allows a more gradual introduction of the concepts. In the case of $d = 3$, a modification is introduced to the d-dimensional algorithm which significantly reduces its running time.

5.1 Existing parallel solutions

A number of algorithms have recently been proposed, for solving the linear programming problem in parallel for the more powerful CRCW PRAM model. Deng [Den90], developed an optimal algorithm based on the ideas in [Meg83, Dye84] to solve the linear programming problem in the plane which runs in $O(\log n)$ time with $n/\log n$ processors. Vaidya [Vai90] developed an algorithm based on the interior point methods [Chv83] which runs in $O\left(L\,(nd)^{1/4}\log^3 n\right)$ time using $O(M(d)n/d + d^3)$ processors, where $M(d)$ is the number of operations for multiplying two $d \times d$ matrices and L is bounded below by the logarithm of the largest absolute value of the determinant of any square submatrix of the coefficient matrix of the linear programming problem. Alon and Megiddo [Alo90] developed a probabilistic algorithm which solves a linear programming problem in fixed dimension almost surely in constant time

Alternately, if the linear programming problem is transformed into the dual space, a parallel solution may be found using parallel algorithms for computing the convex hull of a set of points [Dob80, Ede87]. Each constraint corresponds to a d-dimensional point in the dual space, therefore it suffices to compute the convex hull of the resulting points and test each convex hull point for optimality. Given that computing the convex hull of a set of points is reducible to sorting [Sha78] and that sorting requires $\Omega(n^{1/2})$ time on any mesh architecture (see [Sto83] and Section 2.1 of this thesis), this approach requires $\Omega(n^{1/2})$ time on the reconfigurable mesh. Unfortunately, no algorithms are yet known for solving the convex hull problem in dimensions higher than 3 on the mesh [Mil88c, Deh88].

On the CREW PRAM model, algorithms due to Aggarwal et al [Agg88] and Atallah and Goodrich [Ata86] exist which compute the convex hull of a set of points in 2 dimensions in $O(\log n)$ time using n processors. Given that the convex hull points can be tested for optimality in $O(d)$ time, and the optimal point can be chosen in $O(\log n)$ time by computing the maximum or the minimum [Val75], the linear programming problem can be solved in $O(\log n)$ time in 2 dimensions. In 3 dimensions, there exists an algorithm due to Dadoun and Kirkpatrick [Dad87] which finds the convex hull in time $O(\log^2 n \log^* n)$ using n processors, where $\log^* n$ is defined to be the least i such that $\log^{(i)} n \leq 1$, with $\log^{(i)}$ denoting the ith iteration of the log function. The linear programming problem can then be solved in $O(\log^2 n \log^* n)$ time. Nonetheless, there are as yet no parallel solutions for the convex hull problem in higher dimensions on the CREW PRAM.

5.2 The algorithm for the reconfigurable mesh in two dimensions

In this section, a parallel linear programming algorithm for 2 dimensions, based on the sequential algorithm outlined in section 4.1, is presented and shown to run in $O(\log^3 n)$ time on the reconfigurable mesh of size n. The algorithm will make use of the concepts and algorithms already presented in detail in earlier chapters.

Given a linear programming problem in 2 dimensions and n or fewer constraints, stated as

$$\begin{array}{ll} \text{minimize} & ax_1 + bx_2 \\ \text{subject to} & a_i x_1 + b_i x_2 + c_i \leq 0, \quad i = 1, \ldots, n. \end{array}$$

it is desired to solve it on a reconfigurable mesh of size n. The processors of the mesh are labeled in the snake-like ordering.

Initially, the problem is transformed so that the objective function is equal to the y coordinate, by broadcasting the coefficients a and b to all processors and having each processor P_i apply the linear transformation $y = ax_1 + bx_2$ and $x = x_1$ to the constraint i assigned to that processor. The broadcasting can be accomplished by connecting all switches of the reconfigurable bus in order to form a global bus which allows data to be sent to all processors. The problem is then stated as

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & \alpha_i x + \beta_i y + c_i \leq 0, \quad i = 1, \ldots n, \end{array}$$

where $\alpha_i = a_i - \frac{a}{b} b_i$ and $\beta_i = \frac{b_i}{b}$. The set of constraints is partitioned into three subsets I_1, I_2 and I_3 depending on β_i being negative, positive or zero. No assumption is made about how the constraints are distributed among the processors of the mesh, with constraints from all three subsets being intermixed. It will be assumed that each processor has available the values u_1 and u_2, delimiting the interval containing the optimum value of x, and the splitter x_α, which is the current value of x being tested. The values $u_1 = \max\left\{-\frac{c_i}{\alpha_i}, i \in I_3\right\}$ and $u_2 = \min\left\{-\frac{c_i}{\alpha_i}, i \in I_3\right\}$, can each be computed by applying the max operation taking into account only constraints in I_3

Having transformed the linear programming problem, the parallel algorithm must find a minimum value of a piecewise linear convex function of x, implicitly defined by the constraints As done sequentially, the algorithm iterates by testing values of x, in a way similar to binary search, to determine if x gives the optimal solution and if not to which side of x the optimal solution may lie. At each iteration either the solution is found or at least a constant proportion of constraints are eliminated as candidates for containing the optimal solution, until the number of constraints is small and the problem can be solved directly. Each iteration involves a number of consecutive steps which operate on all the constraints in parallel

Candidates for the optimal solution are obtained by pairing constraints in order to compute the intersection points of the two lines defining the constraints. The pairing step is performed first on the constraints of I_1, then, in an identical manner, on the constraints of I_2. Pairing of constraints in I_1 can be accomplished by numbering the constraints of I_1 and pairing each odd constraint with the following even constraint. The numbering can be accomplished by assigning a 1 to each processor holding a constraint in I_1 and a 0 to all others and then performing the parallel prefix operation.

In order that the computation of the intersection points be performed efficiently, it is required that the constraints in each pair can communicate in $O(1)$ time. When the algorithm works on I_1, all processors P_j, $1 \leq j \leq n$, holding constraints in I_2, I_3 or empty, will form bus bridges between non-adjacent processors holding consecutive constraints in I_1, by connecting their switches to P_{j-1} and P_{j+1} in the snake-like ordering. These bus bridges will allow for $O(1)$ time communication between any two processors holding a pair of constraints.

Once the pairs are determined, one constraint can be eliminated from each pair of constraints defined by parallel lines and one constraint from pairs whose intersection lies outside the interval $[u_1, u_2]$, see Figure 4.2. For the remaining pairs, of which there are at most $n/2$, find an α-splitter x_α of the x-coordinates of their intersection points, with $\alpha = \frac{1}{4}$, by applying one iteration of the parallel selection algorithm. To test where, with respect to the line $x = x_\alpha$, lies the final solution, compute $g(x_\alpha) = \max_{i \in I_1} (\delta_i x_\alpha + \gamma_i)$ and $h(x_\alpha) = \min_{i \in I_2} (\delta_i x_\alpha + \gamma_i)$, where $\delta_i = -\frac{a_i}{b_i}$ and $\gamma_i = -\frac{c_i}{b_i}$. Also compute the slopes of g and h to the left and to the right of x_α, which are defined as

$$
\begin{aligned}
L_g &= max\{\delta_i : i \in I_1,\ \delta_i x_\lambda + \gamma_i = g(x_\alpha)\}, \\
R_g &= min\{\delta_i : i \in I_1,\ \delta_i x_\lambda + \gamma_i = g(x_\alpha)\}, \\
L_h &= max\{\delta_i : i \in I_2,\ \delta_i x_\lambda + \gamma_i = h(x_\alpha)\}, \\
R_g &= min\{\delta_i : i \in I_2,\ \delta_i x_\lambda + \gamma_i = h(x_\alpha)\}
\end{aligned}
$$

Each of the above functions can be computed, in turn, using the max operation. As described in section 4.1, on the basis of this information it can be determined whether x_α is feasible and whether it is optimal and if not to which side of x_α the optimum may lie.

If the problem is found to be infeasible, or the optimal solution is found, then the algorithm terminates. Otherwise, the interval $[u_1, u_2]$ can be reduced to either $[u_1, x_\alpha]$ or $[x_\alpha, u_2]$. At least αn of the intersections, as defined by the pairs of constraints, will lie outside the

new interval. Since for each such pair one constraint can now be dropped, which, including one of parallel lines dropped before, is at least $\frac{\alpha}{2}n$ constraints, there are at most $(1-\alpha/2)n$ constraints remaining.

It remains to mark processors holding constraints which are no longer to be considered as "inactive" Since the algorithm makes no assumptions about the distribution of constraints among processors, no other clean-up or compression of data is required in preparation for the next iteration of the algorithm.

The time required to complete an iteration of this algorithm is limited by the $O(\log^2 n)$ time it takes to find the splitter. All other operations (transformation of coordinates, pairing, comparisons) take $O(1)$ time, except for the parallel prefix operation which takes $O(\log n)$ time and finding the max which takes $O(\log\log n)$ time. Since at each iteration, the size of the problem is reduced by a constant proportion, the algorithm must terminate after at most $\log n$ steps Therefore, the total running time is $O(\log^3 n)$.

5.3 The algorithm for the reconfigurable mesh in d dimensions

In this section, a parallel linear programming algorithm for d dimensions, based on the sequential algorithm outlined in section 4.2, is presented and shown to run in $O(n^{1/2})$ time on the reconfigurable mesh of size n, with the constant of proportionality growing exponentially with dimension

A d-dimensional linear programming problem with n constraints can be stated as

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \geq b_i, \quad i = 1, \ldots, n. \end{aligned}$$

In order to solve this problem on the reconfigurable mesh of size n, constraints are assigned one per processor, with each processor storing the d coefficients of its constraint The processors of the mesh are labeled in snake-like ordering and the constraints are distributed among the processors in no particular order.

Initially, the problem is transformed into a subspace orthogonal to the objective function.

The constraints can be partitioned into I_1, I_2 and I_3 depending on a_{id} being negative, positive or zero, and the problem can be stated as

$$\begin{array}{lll} \text{minimize} & x_d & \\ \text{subject to} & x_d \geq \sum_{j=1}^{d-1} a_{ij}x_j + b_i, & i \in I_1 \\ & x_d \leq \sum_{j=1}^{d-1} a_{ij}x_j + b_i, & i \in I_2 \\ & \sum_{j=1}^{d-1} a_{ij}x_j + b_i \leq 0, & i \in I_3 \end{array}$$

with $|I_1| + |I_2| + |I_3| = n$. Since each processor can apply this transformation, in $O(d)$ time, to the constraint which it has stored and since all processors can do this in parallel, the transformation can be accomplished in $O(d)$ time.

The algorithm is similar to that of two dimensions in that at each iteration it removes a constant proportion of constraints until only the d essential constraints remain, the intersection of which determines the optimum. Each set I_1, I_2 and I_3 is considered in turn. Pairs of inequalities are formed, as before, beginning with the set I_1. The constraints are numbered by applying the parallel prefix operation to the mesh and each odd constraint is paired with the following even one. As before, the remaining processors form bus bridges to allow for constant time communication of the constraints in each pair.

Considering a pair of constraints i, $i+1$ in the same set:

- If $(a_{i1}, \ldots a_{i,d-1}) = (a_{i+1,1}, \ldots, a_{i+1,d-1})$, then one of the constraints is redundant and can be dropped.
- If $(a_{i1}, \ldots a_{i,d-1}) \neq (a_{i+1,1}, \ldots, a_{i+1,d-1})$, then $\sum_{j=1}^{d-1} a_{ij}x_j + b_i = \sum_{j=1}^{d-1} a_{i+1,j}x_j + b_{i+1}$ is an equation of a $(d-1)$-dimensional hyperplane which divides the space so that on one side of this hyperplane constraint i dominates constraint $i+1$, that is $\sum_{j=1}^{d-1} a_{ij}x_j + b_i < \sum_{j=1}^{d-1} a_{i+1,j}x_j + b_{i+1}$, and constraint $i+1$ dominates constraint i on the other side.

If the location of the optimum is known relative a given hyperplane, then one of the two constraints can be eliminated. As shown by Megiddo [Meg84], testing of a hyperplane can be accomplished by solving at most three $(d-1)$-dimensional linear programming problems

with at most n constraints each. It may easily be verified that these problems can be formulated from the original constraints on the basis of the information already available to each processor. Repeatedly forming and testing the dividing hyperplanes in order to remove one constraint per test would result in a very slow algorithm. Instead, Megiddo's and Dyer's multi-dimensional search technique described in chapter 4 can lead to an efficient parallel algorithm.

5.3.1 Applying the multi-dimensional search technique

Given a set of n dividing hyperplanes $h_i(x) = a_i^T x + b_i$, $i = 1, \ldots, n$, distributed no more than one per processor of the reconfigurable mesh, the multi-dimensional search described in section 4.2.2 is applied as follows. When $d = 1$, the hyperplanes are of the form $h_i(x) = x_1 + b_i$. The median value β of the b_i, $i = 1, \ldots, n$ can be found by applying the procedure Select of section 3.2 in $O(\log^3 n)$ time. Then the sign of $h(x) = x_1 + \beta$ can be determined in $O(1)$ time, thus making known the sign of at least half of the n hyperplanes. When $d \geq 2$, the median slope a_α of the lines $a_{i1}x_1 + a_{i2}x_2 + b_i = 0$, defined as a_{i2}/a_{i1}, is computed leaving aside hyperplanes with $a_{i1} = 0$. A transformation taking x_1 to $x_1 + a_\alpha x_2$ is applied, so that any hyperplane with the median slope has now a zero slope. Then, the hyperplanes are paired, so that each pair has one hyperplane with a positive slope and one with a negative slope, leaving aside any hyperplanes with zero slope.

This particular method of pairing hyperplanes which requires that the hyperplanes in each pair satisfy certain properties is different from the method, used earlier, for pairing constraints. Since the hyperplanes with positive and negative slopes are distributed randomly among the processors, in the worst case, it may be required that half of the hyperplanes are moved across the mesh. The pairing can be accomplished by separately numbering all hyperplanes with positive slopes and all hyperplanes with negative slopes, using the parallel prefix operation, then moving the ith hyperplane with negative slope to the processor P_{2i-1} and the ith hyperplane with positive slope to processor P_{2i}, with the processors in the snake-like ordering. It has been shown by Miller et al [Mil88a] how moving $O(n)$ values can be accomplished in $O(n^{1/2})$ time on the reconfigurable mesh.

For each pair of hyperplanes h_i, h_j, as formed above, auxiliary hyperplanes $h_{ij}^{(1)}$, $h_{ij}^{(2)}$ are

formed, where

$$h_{ij}^{(1)}(x) = (h_i(x) - h_j(x))/(a_{i2} - a_{j2})$$
$$h_{ij}^{(2)}(x) = (-a_{j2}h_i(x) + a_{i2}h_j(x))/(a_{i2} - a_{j2})$$

Since $a_{i2} > 0 > a_{j2}$, $a_{i2} - a_{j2} \neq 0$. Now, $h_{ij}^{(1)} = h_{ij}^{(1)}(0, x_2, \quad, x_d)$ and $h_{ij}^{(2)} = h_{ij}^{(2)}(x_1, 0, \quad, x_d)$ are hyperplanes in $(d-1)$ dimensions. For each pair $h_{ij}^{(1)}$, $h_{ij}^{(2)}$ for which the location of the optimum x^* is determined relative to both auxiliary hyperplanes, the location is known relative to at least one of h_i or h_j and allows one constraint per pair to be eliminated.

Three different approaches to the multi-dimensional search were described in chapter 4. Megiddo's first approach is inherently sequential in that it first performs a recursive search on all the auxiliary hyperplanes $h^{(1)}$ and then applies the recursive search to those hyperplanes $h^{(2)}$ whose sign was discovered in the first search. A parallel version of this approach will not be developed, since Megiddo's first approach is a particular case of Dyer's more general and efficient approach.

Megiddo's second scheme recursively finds the signs of all the auxiliary hyperplanes by solving two $(d-1)$-dimensional search problems, each with at most $n/2$ hyperplanes. This can be implemented on the reconfigurable mesh by separating the hyperplanes into two groups of consecutive processors in snake-like ordering, one with hyperplanes which are not a function of x_1, which includes all $h^{(1)}$'s and the original hyperplanes with $a_{i1} = 0$, and the other with all the hyperplanes which are not a function of x_2, the $h^{(2)}$'s and hyperplanes with $a_{i2} \quad 0$. It is important that the mesh be subdivided so that its diameter is always proportional to the square root of the size of the problem. This can be accomplished by subdividing the mesh horizontally at one recursive level and vertically at the next level. Again, as shown by Miller et al [Mil88a], moving $O(n)$ values can be accomplished in $O(n^{1/2})$ time. Now, the search can be applied recursively and in parallel to both sets of hyperplanes. Although the total number of queries $Q(n,d)$ will remain the same, as in the sequential application of the technique, that is $Q(n,d) = O(\log^d n)$ with $C(d) = \frac{2^d}{(d-2)!}$, there will only be $q(n,d)$ levels of recursion, where

$$\begin{cases} q(n,d) & = q\left(\frac{n}{2}, d-1\right) + q\left(\frac{n}{2}, d\right) \\ q(n,1) & = 1 + \lfloor \log_2 n \rfloor \\ q(1,d) & = 1 \end{cases}$$

which can be solved by a technique developed in [Mon80] to give $q(n,d) \quad O\left(\log^d n\right)$ with

a constant of proportionality $C(d) = \frac{1}{d!}$.

The time required for pairing of hyperplanes and finding the median is

$$\begin{cases} t(n,d) &= t\left(\frac{n}{2}, d-1\right) + t\left(\frac{n}{2}, d\right) + \Theta(n^{1/2}d) \\ t(n,1) &= n^{1/2} \\ t(1,d) &= 1 \end{cases}$$

which solved similarly gives $t(n,d) < \frac{d}{(d-2)!} n \log^{d-2} n$.

This leads to the following recurrence for the linear programming problem

$$lp(n,d) \leq c \frac{\log^d \frac{n}{2}}{d!} lp(n, d-1) + lp\left(\frac{n}{2}, d\right) + O\left(\frac{d}{(d-2)!} n^{1/2} \log^{d-2} n\right)$$

For a fixed d, $lp(n,d) = O\left(n^{1/2} \log^{d^2+d-2} n\right)$, with a constant $C(d) < \frac{d}{(d-2)! \prod_{k=1}^{d} k!}$, as can be verified by induction.

Dyer's general multi-dimensional search technique can lead to an algorithm on the reconfigurable mesh with $O(n^{1/2})$ running time with the constant of proportionality singly exponential in d. By choosing a $[2,2]$, $[2,3]$ scheme and applying each procedure in the scheme one after another, after compressing the current subproblem of size m into processors P_{ij}, $i,j \in [0, \ldots, m^{1/2}]$, a proportion $p_d \geq \frac{1}{2}$ of hyperplanes is guaranteed to be discovered after $q_d = 9^{d-1}$ inquiries. The time $t(n,d)$ to solve the multi-dimensional search, using a procedure $A(d, q_d, p_d)$ generated by a scheme $[k_i, l_i]$, $i = 1, \ldots, r$, is

$$t(n,d) \leq \sum_{i=1}^{r} (k_i + l_i)\, t(n, d-1) + K\, d\, n^{1/2},$$

which gives $t(n,d) = q_d d n^{1/2} = O(d 3^{2d-1} n^{1/2})$ for a [2,2] [2,3] scheme, as may be verified by induction. Then the time to solve the linear programming problem is expressed by the following recurrence relation

$$lp(n,d) \leq 3 \;\; 3^{2(d-1)} \left(lp(n, d-1) + K d n^{1/2}\right) + lp\left(\frac{3}{4}n, d\right).$$

Assuming inductively that $lp(n',d') < K\, 3^{(d'+1)^2} (n')^{1/2}$ for all $(d',n') < (d,n)$, it can be verified that

$$\begin{aligned} lp(n,d) &\leq 3^{2d-1}\left(K\, 3^{d^2} n^{1/2} + K d n^{1/2}\right) + K\, 3^{(d+1)^2} \left(\tfrac{3}{4}n\right)^{1/2} \\ &= K\, 3^{(d+1)^2} \left(\tfrac{1}{3^2} + \tfrac{d}{3^{d^2+2}} + \left(\tfrac{3}{4}\right)^{1/2}\right) n^{1/2} \\ &\leq K\, 3^{(d+1)^2} \left(\tfrac{1}{9} + \tfrac{1}{27} + \tfrac{3}{4}\right) n^{1/2} \\ &< K\, 3^{(d+1)^2} n^{1/2}. \end{aligned}$$

This verifies that $lp(n,d) = O(3^{(d+1)^2} n^{1/2})$ for all pairs (n,d).

It is possible to apply the recursive procedures to the two $(d-1)$-dimensional sets of hyperplanes in parallel after separating the two, so long as the proportion p_{d-1} of signs discovered is strictly greater than $\frac{1}{2}$. This would guarantee that at least $p_{d-1} - \frac{1}{2}$ proportion of pairs would have the signs of both auxiliary hyperplanes known. This would increase $\sum_{i=1}^{r}(k_i + l_i)$ in order to guarantee the same proportion could be maintained at each stage. It would not however result in an asymptotic improvement for the constant of proportionality for the parallel linear programming algorithm, since that could only be achieved if $\sum_{i=1}^{r} \max\{k_i, l_i\} = 1$, which could only be true if $r = 1$. However, this could not guarantee a proportion $p_d > \frac{1}{2}$. Although the theoretical benefits of this approach are uncertain, in practice the signs of more pairs of hyperplanes than $p_{d-1} - \frac{1}{2}$ are likely to be discovered, thus resulting in a greater proportion of constraints being removed.

Given that the reconfigurable mesh model assumes a constant number of registers available to each processor, it is necessary to verify that the number of registers required by the algorithm does not exceed this constant. Originally constraints are distributed one per processor of the reconfigurable mesh, with each processor storing the d coefficients of its constraint plus some additional constant amount of information. The only time during the algorithm at which constraints are moved from one processor to another occurs during the pairing of hyperplanes as part of the multi-dimensional search. The i*th* hyperplane with negative slope is moved to the processor P_{2i-1} and the i*th* hyperplane with positive slope to the processor P_{2i}. This causes the hyperplanes to accumulate in some processors of the mesh. Given that at most one additional hyperplane arrives at any processor during a given pairing and that the multi-dimensional search procedure has depth of at most d recursive calls, at most d constraints can arrive at any processor of the mesh. As the dimension d is considered constant, this verifies the requirement for a constant number of registers at each processor.

As the pairing of hyperplanes requires $\Omega(n^{1/2})$ time, employing data movement and selection algorithms which run in $o(n^{1/2})$ time has no effect on the asymptotic running time of the linear programming algorithm. However, since such algorithms (max, parallel prefix, sorting) are known to run in $O(n^{1/2})$ time on a mesh computer with no broadcasting,

the d-dimensional linear programming algorithm can be implemented to run on any mesh architecture in the same $O(n^{1/2})$ time as on the reconfigurable mesh.

5.3.2 The special case when $d = 3$

In 3 dimensions, given n constraints distributed no more than one per processor of the reconfigurable mesh, the constraints are paired to obtain $n/2$ two-dimensional hyperplanes (lines), to which then the multi-dimensional search is applied. During the multi-dimensional search, the hyperplanes are paired to give two sets of 1-dimensional auxiliary hyperplanes (points) at which time the recursion bottoms-out (see section 4.2.2).

It is apparent that the limiting step of the algorithm is pairing of hyperplanes, which in the worst case takes $O(n^{1/2})$ time. It is possible to improve the time it takes to pair hyperplanes to $O(n^{1/3})$, by considering the following modification in the way that procedure $A(2, q_2, p_2)$ is applied in 2 dimensions to solve the multi-dimensional search. Let the mesh be subdivided into $n^{1/3}$ blocks of $n^{1/3} \times n^{1/3}$ size. Each block will have at most $n^{2/3}$ lines. For all blocks in parallel, apply the procedure A to the hyperplanes in that block. Procedure A will be applied to a block in the same fashion that it was applied to the entire mesh in the d-dimensional algorithm, but completes in $O(n^{1/3})$ time since the diameter of the blocks is of $O(n^{1/3})$. Each line returned by procedure A can be tested (see section 4 2.1) by solving at most three 2-dimensional linear programming problems. The tests, which take $O(\log^3 n)$ time each, can be applied one at a time using the whole mesh and will complete in $O(n^{1/3} \log^3 n)$ time. For the proportion p_2 of hyperplanes for which the procedure A determined the sign, one constraint can be eliminated (see section 4.2). Since groups were assigned disjoint subsets of constraints, the total number of constraints which can be eliminated is simply multiplied by the number of groups, to give the required proportion p_2 The time $t(n, 2)$ to solve the multi-dimensional search in 2 dimensions, using a procedure $A(2, p_2, q_2)$ generated by a scheme $[k_i, l_i]$, $i = 1, \ldots, r$, is

$$t(n, 2) \leq \sum_{i=1}^{r} (k_i + l_i)\, t(n, 1) + K n^{1/3},$$

which gives $t(n, 2) = O(n^{1/3})$ Then the time to solve the linear programming problem is

expressed by the following relation

$$lp(n,3) \leq q_2 n^{1/3} t(n,2) + K n^{1/3} + t\left(\frac{3}{4}n, 3\right),$$

where $q_2 = \sum_{i=1}^{r}(k_i + l_i)$, which gives $lp(n,3) = O(n^{1/3}\log^3 n)$.

Since the number of tests required becomes $n^{1/3}$, this scheme does not extend into higher dimensions, in fact, it results in a worse running time than the d-dimensional algorithm of section 5.2.

5.4 The algorithm for the CREW PRAM

The d-dimensional parallel linear programming algorithm can be adapted to run in $O(\log^d n)$ time on the CREW PRAM model, with the constant of proportionality exponential in d. Initially, each processor is assigned one of the n constraints, with the constraints distributed in no particular order Once the subsets I_1, I_2 and I_3 are determined, the constraints can be rearranged, in constant time, so that processors $P_1, \ldots, P_{|I_1|}$ are responsible for constraints in I_1, processors $P_{|I_1|+1}, \ldots, P_{|I_1|+|I_2|}$ are responsible for constraints in I_2 and processors $P_{|I_1|+|I_2|+1}, \ldots, P_{|I_1|+|I_2|+|I_3|}$ are responsible for constraints in I_3 In this way, the algorithm can work in parallel on all pairs of constraints When the multi-dimensional search is applied to the hyperplanes $h(x)$ obtained from pairing constraints, the hyperplanes can be paired and auxiliary hyperplanes $h_{ij}^{(1)}$ and $h_{ij}^{(2)}$ can be formed The procedure can then be applied recursively to the two sets of $(d-1)$-dimensional auxiliary hyperplanes Since pairing of hyperplanes can be accomplished in $O(1)$ time, after they have been enumerated with the parallel prefix operation which takes $O(\log n)$, and since median can be found in $O(\log n)$ time, the time $t(n,d)$ to solve the multi-dimensional search using a procedure $A(d, q_d, p_d)$ generated by a scheme $[k_i, l_i]$, $i = 1, \ldots, r$ is

$$t(n,d) \leq \sum_{i=1}^{r}(k_i + l_i)\, t(n, d-1) + K\, d \log n,$$

which gives $t(n,d) = q_d d \log^d n = O(d 3^{2d-1} \log^d n)$, for a [2,2] [2,3] scheme, as may be verified by induction Then the time to solve the linear programming problem is expressed by the following recurrence relation

$$lp(n,d) \leq 3 \cdot 3^{2(d-1)}\left(lp(n, d-1) + K d \log n\right) + lp\left(\frac{3}{4}n, d\right).$$

Assuming inductively that $lp(n', d') < K\, 3^{(d')^2} \log^{d'} n'$ for all $(d', n') < (d, n)$, it can be verified that

$$\begin{aligned} l(n,d) &\leq 3^{2d-1}\left(K\, 3^{d^2} \log^{d-1} n + K\, d \log n\right) + K\, 3^{(d+1)^2} \log^d \left(\tfrac{3}{4} n\right) \\ &= K\, 3^{(d+1)^2} \left(\tfrac{\log^{d-1} n}{3^2} + \tfrac{d \log n}{3^{d^2+2}} + \log^d \tfrac{3}{4} n\right) \end{aligned}$$

Let $c = -\log \frac{3}{4}$, then it remains to show that

$$\begin{aligned} \log^d n &> \tfrac{\log^{d-1} n}{3^2} + \tfrac{d \log n}{3^{d^2+2}} + (\log n - c)^d \\ &> \tfrac{\log^{d-1} n}{3^2} + \tfrac{d \log n}{3^{d^2+2}} + \log^d n - dc \log^{d-1} n + \tbinom{d}{2} c^2 \log^{d-1} n - \tbinom{d}{3} c^3 \log^{d-3} n + \ldots \end{aligned}$$

Which is true since

$$\frac{\log^{d-1} n}{3^2} + \frac{d \log n}{3^{d^2+2}} < dc \log^{d-1} n + \binom{d}{2} c^2 \log^{d-1} n - \binom{d}{3} c^3 \log^{d-3} n + \ldots,$$

for all $d > 1$. This verifies that $lp(n, d) = O(3^{(d+1)^2} \log^d n)$ for all pairs (n, d).

Chapter 6

Applications

The continuing popularity of linear programming, both in applications and as a research topic, can be attributed to the fact that a great many practical problems can be expressed as linear programming problems and efficiently solved using an established technique such as the Simplex method [Dan63] or Megiddo's and Dyer's algorithm [Meg84, Dye86].

Although it is not clear whether Megiddo's and Dyer's algorithm can compare in efficiency to the Simplex method, their algorithm appears to be very practical for a small number of dimensions. For $d = 3$, Megiddo [Meg84] states that the current computational experience is very successful. In higher dimensions, even though the algorithm is exponential in d, Dyer [Dye86] showed that it is able to take advantage of sparsity, that is for sparse problems the proportion of hyperplanes discovered during the multi-dimensional search increases. In addition, Megiddo pointed out that although, when testing a hyperplane, three $(d-1)$-dimensional problems with possibly n constraints each need to be solved, the number of constraints is usually much smaller in two of the problems. That is, assuming a non degenerate case, two of the three problems have no more than d constraints each. Megiddo also observed that another practical speedup can be realized by choosing a random 3-sample or 5-sample instead of finding the median (or splitter), since the selection is repeated many times and each repetition is independent. The same factors apply in the case of the parallel algorithm and should result in better practical performance than that given by the worst case time analysis.

Any problem which can be formulated as a linear programming problem may be solved using the algorithm developed in this thesis, nonetheless the fact that the running time

is exponential in dimension limits the applicability of the algorithm to problems with small dimension The sequential linear programming algorithms by Megiddo and Dyer are similarly limited

The most obvious area of applications is that of operations research, in which there arise problems such as allocation of resources, planning and scheduling of production and inventory These, however, can often be very large in both number of constraints and dimension and therefore it may not be practical to solve them using the parallel algorithm.

Some other problems, which can be solved using the proposed parallel algorithm are those which can be shown to be reducible to linear programming [Dob80]. For these problems, it is sufficient to show that they are reducible in polynomial time to linear programming and it follows that they can be solved by the parallel algorithm proposed in this thesis. Of course, it is desirable, that the time complexity of the reduction process is in the same order of complexity as solving the linear programming problem.

6.1 The linear separability problem

Two point sets are *linearly separable* if and only if there exists a hyperplane such that all points of one set lie on one side of the hyperplane and all points of the other set lie on the other side of the hyperplane [Sha78]. Recognizing whether two sets are linearly separable and finding a separating hyperplane has applications in statistics and in pattern recognition for the purpose of classifying data points using linear functions [Sha78]. Dobkin and Reiss [Dob80] have shown that this problem in d dimensions is equivalent to linear programming in d variables, that is linear separability is reducible to linear programming and linear programming is reducible to linear separability It can therefore be solved sequentially in linear time, in fixed dimension, with Megiddo's algorithm [Meg84] Preparata and Shamos [Pre85] have shown how to find the $(d-1)$-dimensional separating hyperplane. Given two sets of points $S_1 = \left\{\left(a_1^{(i)}, \ldots, a_d^{(i)}\right) : i = 1, \ldots, |S_1|\right\}$ and $S_2 = \left\{\left(a_1^{(i)}, \ldots, a_d^{(i)}\right) : i = |S_1|+1, \ldots, |S_2|\right\}$, with $|S_1| + |S_2| = n$, the separating hyperplane $p_1 x_1 + \ldots + p_d x_d + p_{d+1} = 0$, if one exists, must satisfy the conditions

$$\begin{aligned} p_1 a_1^{(i)} + \quad + p_d a_d^{(i)} + p_{d+1} &\leq 0, \qquad 1 \leq i \leq |S_1| \\ p_1 a_1^{(i)} + \ldots + p_d a_d^{(i)} + p_{d+1} &\geq 0, \qquad |S_1|+1 \leq i \leq n. \end{aligned}$$

Many separating hyperplanes may exist. Since solving the above linear program with n constraints on the reconfigurable mesh can be accomplished in $O(n^{1/2})$ time and in $O(\log^d n)$ time on the CREW PRAM, one separating hyperplane can also be found, if it exists, in $O(n^{1/2})$ and $O(\log^d n)$ time respectively.

When one of the sets contains only one point, that is $S_1 = \{P_0\}$, the problem of finding a separating hyperplane is known as *point-set separability* and was also shown to be equivalent to the problem of linear programming [Dob80] Megiddo [Meg83] stated that this problem in R^d can be solved by linear programming in $d-1$ variables as it requires finding a hyperplane passing through P_0 which has all the points of S_2 lying to one side of it Therefore, the point-set separability problem, which also determines if P_0 is extreme with respect to the points of S_2 [Dob80, Meg83] can be solved in $O(n^{1/2})$ on the reconfigurable mesh and in $O(\log^d n)$ time on the CREW PRAM.

6.2 Circular separability and the digital disk

Another problem which can be solved by the linear programming algorithm is the circular separability problem. O'Rourke and Rao Kosaraju [O'R85] have shown that the circular separability problem in 2 dimensions reduces to linear separability in 3 dimensions and stated that in general spherical separability in d dimensions reduces to linear separability in $d+1$ dimensions Since linear separability is equivalent to linear programming, they concluded that by Megiddo's algorithm [Meg83] the circular separability problem can be solved in $O(n)$ time with the constant of proportionality doubly exponential in dimension The constant of proportionality can immediately be reduced to singly exponential due to Dyer's improved multi-dimensional search technique [Dye86]

As defined in [O'R85], two sets of points $S_1 = \{(x_i, y_i) \mid i \in I_1\}$ and $S_2 = \{(x_i, y_i) \mid i \in I_2\}$ in R^2, with $|S_1| + |S_2| = n$, are *circularly separable* if there exists a circle C, such that each point of S_1 is interior to or on the boundary of C, while each point of S_2 is exterior to or on the boundary of C Transforming points (x, y) in R^2 into points of the form $(x, y, x^2 + y^2)$ in R^3 creates a one-to-one correspondence between circles

$$(x - A)^2 + (y - B)^2 = R^2$$

in the original space and planes

$$ax + by + (x^2 + y^2) = c$$

in the transformed space, as long as $c \geq -(a^2 + b^2)/4$, with $A = -a/2$, $B = -b/2$ and $R^2 = c + (a^2 + b^2)/4$. Therefore, finding a separating plane in R^3 gives a separating circle in R^2. If the points from the two sets S_1 and S_2 are assigned one per processor of the reconfigurable mesh or the CREW PRAM, the mapping $(x, y) \rightarrow (x, y, x^2 + y^2)$ takes constant time and the linear separability problem can be solved in $O(n^{1/3} \log^3 n)$ time on the reconfigurable mesh and in $O(\log^3 n)$ time on the CREW PRAM, by reduction to linear programming. The separating circle, if one exists, can therefore be found in $O(n^{1/3} \log^3 n)$ time on the reconfigurable mesh and in $O(\log^3 n)$ time on the CREW PRAM.

The circular separability problem generalizes into *spherical separability* in d dimensions, where a transformation from d into $d + 1$ dimensions creates a correspondence between separating hyperspheres in d dimensions and separating hyperplanes in $d + 1$ dimensions, with $O(d)$ time required to perform the transformation in parallel on the reconfigurable mesh Therefore, applying the procedure for solving linear separability problems (see section 6.1) to the transformed points, the spherical separability problem in d dimensions can be solved in $O(n^{1/2})$ time on the reconfigurable mesh with the constant of proportionality exponential in d and in $O(\log^{d+1} n)$ time on the CREW PRAM. It should be noted, that the spherical separability problem being discussed, as defined in [O'R85], is different from the problem of finding spherical separation, as defined in [Dob80]. The later problem is a version of the linear separability problems, which requires finding a separating hyperplane between two sets of points, with the points restricted to lying on the unit hypersphere.

O'Rourke and Rao Kosaraju [O'R85] have shown that the circular separability problem, which has applications in pattern recognition and image processing, can be applied to solving the digital disk recognition problem in linear time A *digital disk* is defined as a set of lattice points (points with integer coordinates) which are contained inside or on some circle. Given a set S of points with integer coordinates, the digital disk recognition problem requires determining if S forms a digital disk [Kim84] Let S_1 be the set of n points on the boundary of S Let S_2 be the set of points representing all pixels exterior to S, but adjacent horizontally,

vertically or diagonally to a point in S_1, of which there are at most $8n$. There exists a circle which encloses S_1 and excludes S_2 if and only if S is a digital disk [O'R85]. Thus, the digital disk problem can be solved in $O(n^{1/3}\log^3 n)$ time on the reconfigurable mesh and in $O(\log^3 n)$ time on the CREW PRAM.

In parallel applications, the input to the digital disk recognition problem may not be represented as a set of points, but rather may consist of an $n^{1/2} \times n^{1/2}$ digitized image distributed one per processor on a reconfigurable mesh of size n, where processor $P_{i,j}$ contains the value of the pixel (i, j). The figure represented by the digitized image has at most $4n^{1/2}$ boundary points, at most two in any row or column. Since the reconfigurable mesh is of size n, a modified version of the linear programming algorithm can solve this problem of size $n^{1/2}$ in $O(\log^3 n)$ time. That a modified algorithm can be designed, can easily be verified, since the linear programming algorithm for the CREW PRAM, which runs in $O(\log^3 n)$ time in 3 dimensions, can be utilized here. Supposing that the memory of the PRAM is mapped onto the processors in the first column of the reconfigurable mesh, the reconfigurable bus and the remaining processors of the mesh provide for constant time communication between the processors in the first column. For example, if P_{i0} is required to send a value to P_{j0}, it can send it to P_{i1} in a row broadcast, then to P_{j1} in a column broadcast and finally to P_{j0} in another row broadcast. Subdividing the communication in this way into three steps ensures that a large number of processors can communicate simultaneously using distinct subbusses on the mesh. Miller et al [Mil88b] state without proof, that the smallest enclosing circle of the points can be found in $\Theta(1)$ time. If all the pixels inside the smallest enclosing circle correspond to data points, then the smallest enclosing circle provides a separating circle and the figure is an image of a digital disk.

6.3 The Euclidean one-center problem

Megiddo [Meg83] presented a linear time algorithm for finding the *minimum spanning circle*, which is the smallest circle enclosing n given points in R^2. This problem is also known as the Euclidean one-center problem [Meg83], where the objective is to find a point p_0 (the center of the smallest enclosing circle), whose greatest distance to any point of the set

$S = \{p_1, \ldots, p_n\}$, with $p_i = (x_i, y_i)$, is minimized. The point p_0 can be characterized as

$$\min_{p_0} \max_{i} (x_i - x_0)^2 + (y_i - y_0)^2,$$

however, by introducing a variable z, the following problem can be formulated

$$\begin{array}{ll} \text{minimize} & z \\ \text{subject to} & z \geq (x_i - x)^2 + (y_i - y)^2 \qquad i = 1, \ldots, n. \end{array}$$

Because the constraints are quadratic, this is not a linear programming problem but the constraints can also be expressed as

$$z \geq -2x_i x - 2y_i y + c_i + (x^2 + y^2),$$

where $c_i = x_i^2 + y_i^2$. Preparata and Shamos [Pre85] stated that since the surface represented by a set of constraints of this form is a convex function, the method of eliminating constraints in the 3-dimensional linear programming algorithm [Meg83] remains applicable. Indeed, while the specifics of the two algorithms differ, the structure of Megiddo's algorithm for the minimum enclosing circle problem is almost identical to his 3-dimensional linear programming algorithm Assuming that the n points of the set S are distributed one per processor of the reconfigurable mesh or the CREW PRAM, the minimum enclosing circle can be found in $O(n^{1/3} \log^3 n)$ and $O(\log^3 n)$ time respectively.

The method extends to higher dimensions using the techniques of Megiddo [Meg84] with Dyer's multi-dimensional search [Dye86] to find the smallest hypersphere enclosing n points in R^d in linear time sequentially, when d is fixed Therefore, the Euclidean one-center problem in R^d can be solved in $O(n^{1/2})$time on the reconfigurable mesh and in $O(\log^{d+1} n)$ time on the CREW PRAM

6.4 Finding the smallest separating circle

Megiddo [Meg83] showed that his method for solving linear programming problems can be extended to apply towards solving quadratic programming problems in R^3, which require minimizing a convex quadratic function subject to linear constraints, in linear time O'Rourke and Rao Kosaraju [O'R85] have shown that finding the smallest circle separating two sets of points in the plane, $S_1 = \{(x_i, y_i) \quad i \in I_1\}$ and $S_2 = \{(x_i, y_i) : i \in I_2\}$, can

be expresses as a quadratic programming problem and hence, can be solved with Megiddo's technique. After mapping the points into R^3 (see Section 5.2), the problem can be expressed as

$$\begin{array}{ll} \text{minimize} & \frac{a^2}{4} + \frac{b^2}{4} + c \\ \text{subject to} & ax_i + by_i + (x_i^2 + y_i^2) \leq c, \qquad i \in I_1, \\ \text{subject to} & ax_i + by_i + (x_i^2 + y_i^2) \geq c, \qquad i \in I_2 \end{array}$$

The 3-dimensional linear programming algorithm can be extended, as described by Megiddo [Meg83] for the sequential algorithm, to solve problems of this form in $O(n^{1/3}\log^3 n)$ time on the reconfigurable mesh and in $O(\log^3 n)$ time on the CREW PRAM. If S_2 is empty, this also gives an alternative procedure for finding the minimum spanning circle [O'R85] (see section 5.2).

Megiddo [Meg84] stated that the linear programming algorithm in d dimensions can be extended to give a linear time algorithm for the quadratic programming problem. Therefore, the smallest separating sphere can be found in $O(n^{1/2})$time on the reconfigurable mesh and in $O(\log^{d+1} n)$ time on the CREW PRAM as any of the required changes take only $O(d)$ time in parallel.

It has been shown that finding the largest separating circle requires $\Theta(n \log n)$ time sequentially [O'R85].

6.5 Other applications

Dyer [Dye86] developed a generalized sequential algorithm to solve the *weighed* Euclidean one-center problem which runs in linear time, with the constant of proportionality $3^{(d+2)^2}$. The algorithm is based on the techniques presented in [Dye84] and [Meg83], and the multi-dimensional search technique presented in [Dye86].

It should be noted, that linear programming can be applied to solving other problems, many of which are discussed in [Dob80]. Yet another interesting application, which requires solving many linear programming problems with relatively few constraints, can be found in computing the dual of the d-dimensional Voronoi diagram, as described in [Avi83].

Chapter 7

Conclusions

This thesis has demonstrated that the linear programming problem in R^d can be solved in parallel on the reconfigurable mesh architecture and on the CREW PRAM. The parallel algorithm presented for solving the linear programming problem was based on a sequential technique due to Megiddo and Dyer [Meg83, Meg84, Dye84, Dye86] which demonstrated that linear programming can be solved in linear time in the number of constraints when the dimension is fixed. The parallel algorithm runs in $O(\log^3 n)$ time in R^2, in $O(n^{1/3}\log^3 n)$ time in R^3 and in $O(n^{1/2})$ time in R^d on the reconfigurable mesh architecture. The simplified version of the algorithm runs in $O(\log^d n)$ time on the CREW PRAM. The constant of proportionality is exponential in d.

This thesis has also demonstrated that the selection problem can be solved on the reconfigurable mesh in poly-logarithmic time The parallel selection algorithm presented was based on a sequential algorithm due to Munro and Paterson [Mun80] designed to select from a file when only a limited amount of internal storage is available for computation. The parallel algorithm runs in $O(\log^3 n)$ time, however a splitter can be obtained after only one iteration of the algorithm, that is after $O(\log^2 n)$ steps

Whether the running times achieved by any of the parallel algorithms presented are optimal remains an open question It is clear, however, that the speedup achieved by the parallel algorithm, which is defined as the ratio of the worst-case running time of the best sequential algorithm known to the worst-case running time of the parallel algorithm [Akl85], is non-optimal When solving a problem using n processors in parallel, as was done, the optimal speedup is of $O(n)$. However, even in the case of the selection algorithm, which

forms a part of the linear programming algorithm, a non-optimal speedup of $O(n/\log^4 n)$ was achieved. Although it is rarely possible to achieve optimal speedup, especially when designing algorithms for architectures based on the distributed memory model, in which the structure of the parallel architecture limits the flow of data, it remains to be seen whether faster algorithms can be obtained for the two problems addressed in this thesis.

A number of possibilities exist for attempting to improve the efficiency of the algorithms presented. Since the selection algorithm views the reconfigurable mesh as a linear array of processors with a reconfigurable bus (a one-dimensional equivalent of the reconfigurable mesh), it is possible that a faster algorithm can be obtained by better exploiting the mesh connections. This would have an immediate effect of reducing the running time of the linear programming algorithm in 2 and 3 dimensions, but not the general d-dimensional one. The running time of the d-dimensional algorithm is limited by the $O(n^{1/2})$ time it takes to pair hyperplanes and so only a radically different approach to the multi-dimensional search technique employed in the d-dimensional algorithm could result in an improvement.

Akl [Akl85] defines efficiency as the ratio of the worst case running time of the fastest known sequential algorithm for the problem to the cost of the parallel algorithm, which is the product of the running time of the parallel algorithm and the number of processors used. The efficiency achieved by the linear programming algorithm running on the CREW PRAM model is $O(n/(n\log^d n))$. It can be improved by a factor of $O(\log n)$ if fewer than n processors are used. For example, in 2 dimensions, if each processor is initially assigned $\log n$ constraints, the linear programming problem can be solved in the same $O(\log^2 n)$ time, but with efficiency of $O(n/(n\log n))$, since it is known that the max, the parallel prefix and a splitter can each be computed in $O(\log n)$ time using $n/\log n$ processors on the CREW PRAM [Val75, Lad80, Bre74].

The assumption that the number of available processors can be equal to n, the size of the problem, is not a realistic one. It is possible, that a different algorithm could be arrived at by reducing the number of processors required, where the solution time for the problem would depend on the number of processors available. Such an algorithm should attempt to improve the efficiency without increasing the running time. This, however, would require an enhanced model of the reconfigurable mesh architecture, where each processor could store

large amounts of data. It should be noted, that in order to obtain an optimally efficient parallel algorithm, an alternate selection algorithm would be required, as the sequential version of the current algorithm does not run in $\Theta(n)$ time

The parallel linear programming algorithm presented in this thesis has achieved a substantial improvement in running time over what can be accomplished sequentially. The parallel algorithm makes use of a selection algorithm, also presented in this thesis, which achieves a poly-logarithmic running time on the reconfigurable mesh. It has been shown that a number of problems can be solved by employing the parallel linear programming algorithm, including linear separability, circular separability and digital disk and that the technique can be extended to solve quadratic programming problems.

Bibliography

[Agg88] A. Aggarwal, B. Chazelle, L. Guibas, C.Ó. Dúnlaing and C.K. Yap, "Parallel computational geometry," *Algorithmica*, **3**, 293-327 (1988).

[Aho83] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data structures and algorithms*, Addison-Wesley, Reading, Massachusetts (1983)

[Akl85] S.G. Akl, *Parallel sorting algorithms*, Academic Press, Orlando, Florida (1985)

[Alo90] N. Alon and N. Megiddo, "Parallel linear programming in fixed dimension almost surely in constant time," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, **2**, 574–582 (1990)

[Ata86] M.J. Atallah and M.T. Goodrich, "Efficient parallel solutions to some geometric problems," *Journal of Parallel and Distributed Computing*, **3**, 492–507 (1986)

[Avi83] D. Avis and B.K. Bhattacharya, "Algorithms for computing d-dimensional Voronoi diagrams and their duals," *Advances in Computing Research*, **1**, edited by F.P. Preparata, JAI Press, 159–180 (1983)

[Bha88] B.K. Bhattacharya, "Circular separability of planar point sets," *Computational Morphology*, edited by G.T Toussaint, North-Holland, 25–39 (1988)

[Blu72] M. Blum, R.W Floyd, V.R Pratt, R.L. Rivest, R.E Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, **7**(4), 448-461 (1972)

[Bre74] R.P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, **21**(2), 201-206 (1974)

[Chv83] V. Chvátal, *Linear Programming*, W.H Freeman and Company, New York (1983)

[Cla86] K.L. Clarkson, "Linear programming in $O\left(n \times 3^{d^2}\right)$ time," *Information Processing Letters,* **22**, 21–24 (1986).

[Col85] R. Cole, "A parallel median algorithm," *Information Processing Letters,* **20**, 137–139 (1985)

[Col86] R. Cole, "Parallel merge sort," *27th Annual Symposium on Foundations of Computer Science,* 511–516 (1986).

[Col88] R. Cole, "An optimally efficient selection algorithm," *Information Processing Letters,* '**26**, 295-299 (1987/88)

[Coo82] S. Cook and C Dwork, "Bounds on the time for parallel RAM's to compute simple functions," *Proc. of the 14th Annual ACM Symposium on Theory of Computing,* 231–233 (1982).

[Dan63] G.B. Dantzig, *Linear programming and extensions,* Princeton University Press, Princeton, New Jersey (1963).

[Dad87] N. Dadoun and D G Kirkpatrick, "Parallel processing for efficient subdivision search," *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry,* (1987), pp 205-214

[Deh88] F Dehne, J-R Sack and I. Stojmenović, "A note on determining the 3-dimensional convex hull of a set of points on a mesh of processors," *Proceeding of the 1st Scandinavian Workshop on Algorithm Theory,* in Lecture Notes in Computer Science **318**, 154–161 (1988)

[Den90] X Deng, "An optimal parallel algorithm for linear programming in the plane," *Information Processing Letters,* **35**(4), 213–217 (1990).

[Dob80] D P Dobkin and S P. Reiss, "The complexity of linear programming," *Theoretical Computer Science,* **11**, 1-18 (1980)

[Dye84] M E Dyer, "Linear time algorithms for two- and three-variable linear programs," *SIAM Journal on Computing,* **13**(1), 31–45 (1984).

[Dye86] M.E. Dyer, "On a multidimensional search technique and its application to the Euclidean one-centre problem," *SIAM Journal on Computing*, **15**(3), 725–738 (1986).

[Ede87] H. Edelsbrunner, *Algorithms in combinatorial geometry*, Springer-Verlag, Berlin (1987).

[ElG90] H. ElGindy, "Improved convex hull computation on the reconfigurable mesh architecture," manuscript (1990).

[Fre83] G.N. Frederickson, "Tradeoffs for selection in distributed networks," *Proceedings 2nd ACM Symposium on Theory of Computing*, 154–160 (1983)

[Kim84] C.E. Kim and T.A. Anderson, "Digital disks and a digital compactness measure," *Proceedings 16th Annual ACM Symposium on Theory of Computing*, 117–123 (1984).

[Kle72] V. Klee and G.J. Minty, "How good is the simplex algorithm?" *Inequalities-III*, edited by O. Shisha, Academic Press, New York, 159–175 (1972)

[Knu73] D.E. Knuth, *The art of computer programming*, Vol 3, Addison-Wesley, Reading, Massachusetts (1973)

[Lad80] R.E. Ladner and M J. Fischer, "Parallel prefix computation," *Journal of the ACM*, **27**(4), 831–838 (1980)

[Lee84] D T. Lee and F.P. Preparata, "Computational geometry - a survey," *IEEE Transactions on Computers*, **33**(12), 1072–1101 (1984)

[Meg83] N. Megiddo, "Linear-time algorithms for linear programming in R^3 and related problems," *SIAM Journal on Computing*, 759–776 (1983)

[Meg84] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *Journal of the Association for Computing Machinery*, **31**(1) 114–127 (1984)

[Mil88a] R. Miller, V.K Prasanna-Kumar, D.I. Reisis and Q.F. Stout, "Meshes with reconfigurable buses," *MIT Conference on Advanced Research in VLSI*, 163–178 (1988)

[Mil88b] R Miller, V.K Prasanna-Kumar, D.I. Reisis and Q.F. Stout, "Data movement operations and applications on reconfigurable VLSI arrays," *Proceedings of the International Conference on Parallel Processing*, **1**, 205–208 (1988).

[Mil88c] R Miller and Q F Stout, "Efficient parallel convex hull algorithms," *IEEE Transactions on Computers*, **37**(12), 1605–1618, (1988).

[Mil87] R. Miller and Q.F. Stout, "Mesh computer algorithms for line segments and simple polygons," *Proceedings of the International Conference on Parallel Processing*, 282–285 (1987)

[Mon80] L. Monier, "Combinatorial solutions of multidimensional divide-and-conquer recurrences," *Journal of Algorithms*, **1**, 60–74 (1980).

[Mun80] J.I. Munro and M.S Paterson, "Selection and sorting with limited storage," *Theoretical Computer Science*, **12**, 315–323 (1980).

[O'R85] J. O'Rourke and S Rao Kosaraju, "Computing Circular Separability," The Johns Hopkins University, Technical Report No. JHU/EECS-85/05, (1985)

[Pla89] C.G Plaxton, "On the network complexity of selection," *Proceedings IEEE 30th Annual Symphosium on Foundations of Computer Science*, 396-401 (1989).

[Pra87] V.K Prasanna Kumar and C.S. Raghavendra, "Array processors with multiple broadcasting," *Journal of Parallel and Distributed Computing*, **4**, 173–190 (1987).

[Pre79] F.P. Preparata and J. Vuillemin, "The cube-connected cycles," *Proceedings IEEE 20th Annual Symphosium on Foundations of Computer Science*, 140–147 (1985).

[Pre85] F P. Preparata and M I Shamos, *Computational geometry—an introduction*, Springer-Verlag, New York (1985)

[Sha78] M.I. Shamos, *Computational geometry*, Doctoral Thesis, Department of Computer Science, Yale University, New Heaven, Conneticut (1978).

[Sto83] Q.F. Stout, "Mesh-connected computers with broadcasting" *IEEE Transactions on Computers*, **9**, 826-830 (1983).

[Tho77] C.D. Thompson and H T Kung, "Sorting on a mesh connected parallel computer," *Communications of the ACM*, **20**(4), 263–271 (1977).

[Vai90] P. M. Vaidya, "Reducing the parallel complexity of certain linear programming problems," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, **2**, 583–589 (1990)

[Val75] L.G. Valiant, "Parallelism in comparison problems," *SIAM Journal on Computing*, **4**(3), 348-355 (1975).