Optimal Algorithms for Constrained Reconfigurable Meshes

Bryan Beresford–Smith Oliver Diessel Hossam ElGindy

Department of Computer Science

and Software Engineering

The University of Newcastle

Callaghan NSW 2308

Australia

Abstract

This paper introduces a constrained reconfigurable mesh model which incorporates practical assumptions about propagation delays on large sized buses. Simulations of AT^2 optimal reconfigurable mesh algorithms on the constrained reconfigurable mesh model are found to be non–optimal. Optimal solutions for the sorting and convex hull problems requiring less area are then presented. For the problems investigated, the constrained reconfigurable mesh model predicts a continuum in performance between the reconfigurable mesh and mesh of processors architectures.

Keywords: Parallel algorithms, reconfigurable mesh architectures, broadcasting buses, sorting, convex hull, lower bounds.

1 Introduction

The *reconfigurable mesh* architecture is a two–dimensional array of processors in which each processor is wired to its four neighbours. Each processor controls a set of local short–circuit switches which allow the inter–processor wires to be connected together to form a communication bus. All processors participating in the bus configuration have access to the data available on it, thereby reducing the communication diameter of the array to a constant. Refer to [1, 3] for a detailed description of the reconfigurable mesh model.

Since the first papers [11, 16], research on the reconfigurable mesh architecture has gained considerable momentum. A number of models have been proposed, and various techniques have been introduced to help develop *constant* running time algorithms for image processing, geometric and graph theoretic problems (refer to [7, 1, 3] for a survey of the various models and algorithms). Recent examples include constant time algorithms for sorting n numbers [4, 8, 5] and for determining the convex hull of n planar points [12, 13].

A common feature of reconfigurable mesh models is the assumption that a packet of data can be broadcast in constant time on a bus component independent of its size or length. This feature has attracted criticism and cast a shadow of doubt on the scalability of massively parallel reconfigurable machines.

Investigation of bus delays [10, 14] has indicated that the delay is small, but that it cannot be correctly modeled by a constant no matter how large the bus. In this paper we report on our study of a new approach to coping with bus delay and of its incorporation into the design of algorithms for reconfigurable meshes. The main idea is to model the propagation delay on a *bus–unit*[1] by a constant, and to only permit the class of algorithms, denoted by $\mathcal{A}^k$, which configure bus components bound in size to at most k bus–units to run on the model.

[1] A bus–unit is a segment of the bus that connects adjacent processors.

We give a detailed description of our reconfigurable mesh model in the following section. Lower bounds are discussed in section 3. In section 4 we present an optimal algorithm for sorting on a constrained reconfigurable mesh. An optimal convex hull algorithm for constrained reconfigurable meshes is presented in section 5. We conclude with some general remarks and open problems.

2 The Model

The reconfigurable mesh of size $m \times n$ consists of m rows and n columns of processing elements arranged in a grid. Each processor is connected to its immediate neighbours to the north, south, east and west, when present, and has four similarly labeled I/O ports through which it can communicate with its neighbours. Each processor possesses a constant number of $\Omega(\log mn)$–bit word registers. Processors are numbered from $P_{0,0}$ in the north–western corner, to $P_{m-1,n-1}$ in the south–eastern corner. Processors may also be numbered from P_0 to P_{mn-1} using other orderings. For example, in row major order $P_{i,j} = P_{in+j}$.

Each PE controls a set of local short–circuit switches which allow the four I/O ports to be connected together in any of 15 possible combinations. The processors operate synchronously, in one machine cycle performing an ALU operation, configuring the switches, and communicating via each I/O port. When a connection is set, signals received by a port are simultaneously available to any port connected to it. For example, when processors connect their northern and southern I/O ports, data broadcast onto the column bus so formed can be read by all of the processors in a column. The model allows concurrent reading from a bus, requires exclusive writing to a bus, and usually assumes a constant time communication delay on arbitrarily large connected bus components.

The constant time model is infeasible for a number of reasons. Due to the finite resistance and capacitance per unit wire length, signals need to be regenerated to ensure

accurate detection, and the time to broadcast a signal along the wire is proportional to the square of its length [15]. The speed of light and the clock frequency of the machine also limit the number of processors which can be reached by a signal in one cycle. To account for these limits we propose the *k–constrained* reconfigurable mesh model that allows buses of size at most k to be formed per cycle. We use the notation $\mathcal{RM}_A^k$ to refer to a k–constrained reconfigurable mesh of area A.

A *linear* bus does not branch. Any reconfigurable mesh algorithm which only uses linear buses can be simulated in the k–constrained model by propagating signals k processors at a time. Any algorithm that broadcasts on a linear row or column bus of length l in $\Theta(1)$ time on a reconfigurable mesh can therefore be simulated by a k–constrained algorithm in $\Theta(\frac{l}{k})$ time. Efficient simulations of arbitrarily shaped linear and branching buses are under investigation.

3 Lower Bounds

Using a bisection width argument, it can be shown that the VLSI complexity for sorting n numbers on a square mesh is $\Omega(n^2)$ [15]. When the solution mesh is no longer square, the AT^2 complexity needs to be scaled by the aspect ratio (the ratio of the length of the longer to the shorter side). For a mesh with p rows and q columns, with $p \leq q$, the AT^2 lower bound to sort n numbers increases to $\Omega(\frac{q}{p} \times n^2)$, and the lower bound on the time required to sort on the mesh becomes $\Omega(\frac{n}{p})$.

As the computational complexities of the sorting and the planar convex hull problems are equivalent, their VLSI complexities are identical. The recently developed constant time reconfigurable mesh algorithms for sorting and finding the convex hull of n elements on meshes of size $n \times n$ are therefore optimal with respect to T and AT^2 complexity measures. Since these algorithms only use linear bus configurations of length $O(n)$, they can be simulated on a k–constrained reconfigurable meshes of size $n \times n$ in $O(\frac{n}{k})$ time. Although this time matches the diameter of the network, the simulations

are no longer AT^2 optimal for $k < n$. We are therefore motivated to develop solutions using less area.

4 Optimal Sorting Algorithm

The fundamental problem of sorting n items on a reconfigurable mesh of size $n \times n$ has been addressed by several authors, and constant time AT^2 optimal solutions are now well known [4, 8, 5]. Straightforward simulations of these algorithms, which use linear buses of length $O(n)$ on an $n \times n$ $\mathcal{RM}^k_{n^2}$, have $O(\frac{n}{k})$ running time and $O(\frac{n^4}{k^2})$ AT^2 complexity. Since the running time has been increased without changing the solution area, the AT^2 complexity is no longer optimal when $k < n$.

In this section, we extend these results to an AT^2 optimal algorithm for sorting on $\mathcal{RM}^k_{nk}$. Our algorithm uses the time–optimal algorithm developed by Marberg and Gafni [9] for sorting mn items on an $m \times n$ standard mesh, where $m \geq \sqrt{n}$. The algorithm in [9] uses a constant number of row and column phases and is easily adapted for sorting on the k–constrained reconfigurable mesh. Since the algorithm consists exclusively of oblivious comparison–exchange steps, it may be described (and analyzed) in terms of its action on an arbitrary input of 0's and 1's according to the 0–1 principle [6]. In outline, the algorithm consists of the following steps:

Procedure **RotateSort** [9]

1. Balance the distribution of 0's and 1's among the columns of the mesh by sorting the columns downwards, rotating each row i (i mod n) positions to the right, and again sorting columns downwards.

2. Sort the rows of the mesh to the right

3. Distribute the elements of each $\sqrt{n} \times \sqrt{n}$ block among the columns of the mesh by rotating each row i ($i\sqrt{n}$ mod n) positions to the right and sorting the columns downwards.

4. Balance the distribution of 0's and 1's among the rows of each $\sqrt{n} \times n$ horizontal slice of the mesh by sorting the rows to the right, rotating each column i within the slice (i mod $\sqrt{n}$) positions downwards, and again sorting the rows to the right.

5. Repeat step 3.

6. Perform 3 iterations of *Shearsort*

7. Sort the rows of the mesh to the right

end **RotateSort**

Each step of the algorithm involves a constant number of sorts and cyclic rotations on the data contained, alternately, in rows and columns. A rotation is no harder than a sorting step since the resulting permutation can be achieved by sorting. *RotateSort* uses a constant number of the following basic operations:

1. Sort columns with m items in $O(m)$ time;

2. Sort or rotate rows (in either direction) with n items in $O(n)$ time;

3. Rotate the columns of each $\sqrt{n} \times n$ horizontal slice of the $m \times n$ mesh in $O(\sqrt{n})$ time.

RotateSort therefore requires $O(m+n)$ steps to sort mn items.

We now show how sorting n items on a $\mathcal{RM}^k_{nk}$ of size $k \times n$ can be achieved in $O(\frac{n}{k})$ time. This result may be used in the generalization to k–constrained meshes with arbitrary aspect ratio. For ease of explanation, it will be assumed that k divides n.

Lemma 1 *Let $\mathcal{RM}^k_n$ be a reconfigurable linear array of processors $P_0, ..., P_{n-1}$ and let P_{ik} contain item x_i, $0 \leq i < \frac{n}{k}$. Items $x_0, \ldots, x_{\frac{n}{k}-1}$ can be sorted in $O(\frac{n}{k})$ time by a straightforward simulation of odd–even transposition sort.*

Lemma 2 *k items stored in the first row of $\mathcal{RM}^k_{k^2}$ of size $k \times k$ can be sorted in $O(1)$ time since algorithms [4, 8, 5] use linear buses of length $O(k)$.*

To sort n items stored in the first row of $\mathcal{RM}^k_{nk}$ of size $k \times n$ we arrange the items so that x_{sk+j} is in processor $P_{j,sk}$ for $0 \leq s < \frac{n}{k}$ and $0 \leq j < k$. Each row j then contains $\frac{n}{k}$ items at locations (j, sk) for $0 \leq s < \frac{n}{k}$. The operations of *RotateSort* can be simulated on an $\mathcal{RM}^k_{nk}$ of size $k \times n$ to give an algorithm *SORT(n, k)* with the following times for the phases (for our exposition we replace rows by columns and vice versa):

1. Sorting (or rotating) rows with $\frac{n}{k}$ items: $O(\frac{n}{k})$ time by *Lemma 1* (and since rotation is no harder than sorting);

2. Sorting (or rotating) columns (in either direction) with k items: $O(1)$ time by *Lemma 2* since there is a $k \times k$ block available to sort each column;

3. A rotation, dependent on the size of k as follows:

   (a) if $\frac{n}{k} \geq \sqrt{k}$, rotating rows with $\sqrt{k}$ items in each $k \times k\sqrt{k}$ vertical slice of the $k \times n$ mesh, which takes $O(\sqrt{k})$ time by *Lemma 1*;

   (b) if $\frac{n}{k} < \sqrt{k}$, and hence $k > \sqrt{\frac{n}{k}}$, rotating columns with $\sqrt{\frac{n}{k}}$ items in each $\sqrt{\frac{n}{k}} \times n$ horizontal slice of the $k \times n$ mesh, which takes $O(1)$ time using the $\mathcal{RM}^k_{\sqrt{nk}}$ submesh of size $\sqrt{\frac{n}{k}} \times k$ available to each column.

Theorem 1 *If n items are stored in the first row of a $\mathcal{RM}^k_{nk}$ of size $k \times n$, then algorithm* SORT(n, k) *sorts the items correctly in $O(\frac{n}{k})$ time, which is AT^2 optimal.*

Proof: The correctness and time complexity follow from [9] and the above arguments. The time is optimal since it matches the diameter and bisection lower bounds for $\mathcal{RM}^k_{nk}$ of size $k \times n$. The algorithm matches the AT^2 lower bound for sorting n items on a mesh of size $k \times n$, which is $\Omega(\frac{n}{k} \times n^2)$. ∎

5 Optimal Convex Hull Algorithm

The convex hull of a set of planar points is the smallest convex polygon containing the set. The polygon may be considered to consist of two convex chains of points: one lying

above the line through the most westerly and easterly extreme points, the upper hull, and a similarly defined lower hull. The problem of computing the convex hull may be solved by identifying, in sequence, the vertices of the two chains, and concatenating the two lists. We shall describe methods for computing the upper hull, which can be used with straightforward substitutions to find the lower hull.

Two algorithms have recently been proposed for computing the convex hull of n planar points in constant running time on an $n \times n$ reconfigurable mesh [12, 13]. Both methods form linear buses of $O(n)$ length. Straightforward simulations on an $n \times n$ $\mathcal{RM}^k_{n^2}$, in which signals are propagated k processors at a time, have $O(\frac{n}{k})$ running time. However, the AT^2 complexity of such simulations is $O(\frac{n^4}{k^2})$, which is no longer optimal for $k < n$.

In this section we present an optimal algorithm to solve the convex hull problem for $\mathcal{RM}^k_{nk}$. We employ the divide–and–conquer technique together with efficient merging steps. The method used for each merging step, which requires computing the supporting line of two separable convex polygons, is chosen to suit the aspect ratio.

We use the procedure *SupportLine*, adapted from Nigam and Sahni [12], in our algorithms to compute the line of support between two vertically separable upper hulls, L and R, whose extreme points are in general position. The procedure computes the endpoints of the line of support between R and L, each of size n at most, on a reconfigurable mesh of size $n \times n$ in a constant number of steps.

Procedure **SupportLine** [12]

1. Arrange the points of L, one per row, in the first column of the mesh and broadcast the points along each row.

2. Arrange the points of R, one per column, in the last row of the mesh, and broadcast the points along each column.

3. Each processor containing a point from L and R determines the slope of the line from its point of R to its point of L.

4. Within each column, each processor containing a slope record checks whether the slope is locally minimal by checking its neighbours.

5. Identify the maximum of the slope records found in the previous step using the method of [11].

end **SupportLine**

Lemma 3 *[12] Procedure* SupportLine *correctly computes the line of support between two n–sized vertically separable upper hulls in general position on a reconfigurable mesh of size $n \times n$ in constant time.*

The procedure forms buses of length $O(n)$, and therefore requires $O(\frac{n}{k})$ time to complete on a k–constrained reconfigurable mesh of size $n \times n$.

The extreme points on the upper hull of a set of planar points for RM^k_{nk} of size $k \times n$ can be computed as follows (for simplicity, it will be assumed that k divides n):

Procedure **UpperHull**

1. Load the n points onto the first row of the mesh.

2. Sort the points in order of increasing x–coordinate using the algorithm *SORT(n,k)* of section 4.

3. Set the switches of the $\mathcal{RM}^k_{nk}$ to partition the mesh into $\frac{n}{k}$ components of size $k \times k$ each. Blocks of processors compute the upper hull of their subsets independently using the algorithm of Nigam and Sahni [12]. Upon completion the extreme points are assigned labels in order of their appearance on the upper hull and compressed into contiguous processors ready for merging.

4. Merge the $\frac{n}{k}$ disjoint upper hulls by performing $O(\log(\frac{n}{k}))$ parallel merging stages as in figure 1. During each stage, odd–numbered components are paired with the following even–numbered components and their upper hulls, denoted by L

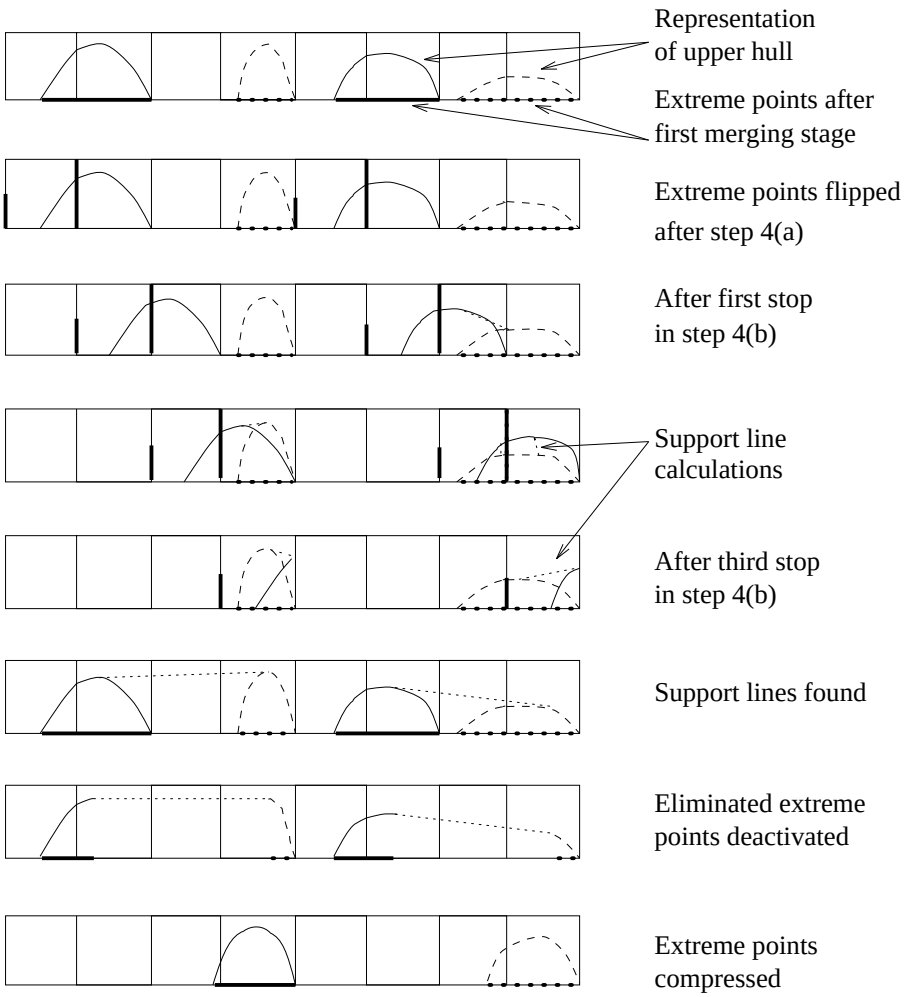

Figure 1: A second merging stage of procedure *UpperHull*.

and R, are merged. Details of merging two upper hulls during the i^{th} stage, $1 \leq i \leq \log(\frac{n}{k})$ are as follows:

(a) Partition the upper hull of each L into at most 2^{i-1} contiguous segments of at most k extreme points each, and flip the extreme points of each segment into the leftmost column of each $k \times k$ block. The extreme points of each R are left in the last row.

(b) Pipeline the segments of L from left to right, by repeatedly advancing the groups k columns at a time. At each stop, compute the support line from the segment of R to the segment of L using procedure *SupportLine*, and store

the endpoints of the line of support with minimum slope for each segment of R.

After $2 * 2^{i-1}$ stops, each segment of R will know its support line with the upper hull of L.

(c) Among the support lines of segments in R with the upper hull in L, we select the one with the maximum slope as the support line between L and R.

End points of the support line, which uniquely identify the upper hull of $L \cup R$, are communicated to the processors of both L and R. The remaining extreme points are identified, assigned labels in order of their appearance on the upper hull, and compressed ready for the next merging stage.

end **UpperHull**

Theorem 2 *Procedure* UpperHull *correctly computes the upper hull of a set S of n planar points on a $\mathcal{RM}^k_{nk}$ of size $k \times n$ in $O(\frac{n}{k})$ time, which is AT^2 optimal.*

Proof: During the i^{th} merging stage, consider the j^{th} segment from the right hull, $R_j, 1 \leq j \leq 2^{i-1}$, to be the upper chain of vertices of a convex polygon forming a line of support, $r_j l_k$, with each segment of the left, $L_k, 1 \leq k \leq 2^{i-1}$, also considered to be the upper chain of vertices of some convex polygon. Since the L_k together form the upper hull, L, the line of support from R_j to L must be the line of support $r_j l_k, 1 \leq k \leq 2^{i-1}$, with minimum slope, because otherwise there is some point of L above this line, contradicting the definition of a supporting line. Considering the supporting lines from each R_j to L, the line of support from R to L must be the line with maximum slope, since otherwise some point of R contradicts the definition of a supporting line.

The running time of procedure *UpperHull*, $CH(n,k)$, can be described by the following recurrence relation:

$$CH(n,k) = LOAD(n) + SORT(n,k) + INIT + Merge(n,k) \quad (1)$$

$$Merge(n,k) = \sum_{i=1}^{\log(\frac{n}{k})} Merge(k2^{i-1}, k) \quad (2)$$

During the i^{th} merging stage of step 4, the procedure merges upper hulls L and R, each of which is the upper hull of $k2^{i-1}$ points. The operations of routing k elements within a block of size $k \times k$ (step 4(a)) and of applying procedure *SupportLine* to compute the line of support between sets of k points in a $k \times k$ block (in step 4(b)) require constant running time. Routing all the segments of L through $O(k2^i)$ columns of the mesh that contain points of L and R requires $O(2^i)$ pipelined time. Thereafter, the remaining extreme points can be identified and resequenced in constant time, and the remaining extreme points can be compressed in $O(2^i)$ time. Therefore, $Merge(k2^{i-1}, k)$ of (2) is performed in $O(2^i)$ time. It follows directly that the running time of step 4 is $O(\frac{n}{k})$.

Since *LOAD(n)*, the running time of step 1, and *INIT*, the running time of step 3, require $O(1)$ time each, and *SORT(n,k)* takes $O(\frac{n}{k})$ time, it follows that the time complexity of *CH(n,k)* is $O(\frac{n}{k})$. This is optimal since the solution time and area match the lower bound for sorting on $\mathcal{RM}^k_{nk}$ of size $k \times n$. ∎

6 Concluding Remarks

Optimal T and AT^2 algorithms for sorting and planar convex hull computation for the RM^k_{nk} of size $k \times n$ have been presented in this paper. Extensions of these algorithms to other aspect ratios can be found in [2], where:

1. an $O(\frac{q}{k})$ time algorithm for sorting n items on a RM^k_{nk} of size $p \times q, p \leq q$, and
2. an $O(\sqrt{\frac{n}{k}})$ time algorithm for computing the convex hull of n planar points on a RM^k_{nk} of size $\sqrt{nk} \times \sqrt{nk}$

are presented. Bridging the extreme cases, for convex hull computations, remains an open problem. We believe a convex hull algorithm for arbitrary aspect ratios would be of practical use in general purpose computing environments where the mesh area available to a task may be limited.

With the algorithms derived using our model, we are able to predict the running time of the sorting and convex hull problems as functions of the problem size, the

degree of constraint, k, and the aspect ratio. As is to be expected, we observe a continuum in performance from the standard mesh of processors, for which $k = 1$, to the usual reconfigurable mesh model, for which k is arbitrarily large. We were able to achieve AT^2 optimality in the k–constrained reconfigurable mesh model for the computational problems we investigated by scaling the standard mesh area required to solve the problem by k, choosing an aspect ratio which matched the diameter of the network to its bisection width, and finding time–optimal procedures to solve the problem. Whether or not this is a general prescription for developing T and AT^2 optimal algorithms for constrained reconfigurable meshes requires further study. It can be argued that the constrained reconfigurable mesh model is asymptotically no faster than the standard mesh of processors model, since for large n it is at best k times faster. Furthermore, the k–constrained reconfigurable mesh requires k times as many processors as the standard mesh to achieve this speedup.

Further work is needed to determine how to handle non–linear buses, and find general techniques for developing optimal $\mathcal{A}^k$.

Acknowledgments:

This research is supported by a grant from the Australian Research Council.

References

[1] H. H. Alnuweiri, M. Alimuddin, and H. Aljunaidi. Switch models and reconfigurable networks: Tutorial and partial survey. In *Proceedings of the Workshop on Reconfigurable Architectures,* 8th International Parallel Processing Symposium, Los Alamitos, California, Apr. 1994. IEEE Computer Society.

[2] B. Beresford-Smith, O. Diessel, and H. ElGindy. Optimal algorithms for constrained reconfigurable meshes. In *Proceedings of the Eighteenth Australasian Com-*

puter Science Conference, pages 32–41, Adeleaide, SA, Feb. 1995. (Also available by ftp from `ftpcs.newcastle.edu.au/pub/reconfig/papers/cm.ps`).

[3] J.-w. Jang, H. Park, and V. K. Prasanna-kumar. A bit model of reconfigurable mesh. In *Proceedings of the Workshop on Reconfigurable Architectures,* 8th International Parallel Processing Symposium, Los Alamitos, California, Apr. 1994. IEEE Computer Society.

[4] J.-w. Jang and V. K. Prasanna-kumar. An optimal sorting algorithm on reconfigurable mesh. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 130–137, Los Alamitos, California, 1992. IEEE Computer Society.

[5] A. Kapoor, H. Schröder, and B. Beresford-Smith. Constant time sorting on a reconfigurable mesh. In *Proceedings of the 16th Australian Computer Science Conference*, pages 121–132, Brisbane, Qld, Feb. 1993. Griffith University.

[6] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

[7] H. Li and Q. F. Stout. *Reconfigurable Massively Parallel Computers.* Prentice Hall Publishers, Englewood Cliffs, New Jersey, 1991.

[8] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang. Sorting in O(1) time on an $n \times n$ reconfigurable mesh. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice (Proceedings of European Workshop on Parallel Computing)*, pages 16–27, Amsterdam, 1992. IOS Press.

[9] J. M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3(4):561–572, Oct. 1988.

[10] M. Maresca and H. Li. Connection autonomy in SIMD computers: A VLSI implementation. *Journal of Parallel and Distributed Computing*, 7(2):302 – 320, Oct. 1989.

[11] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678–692, June 1993. (A preliminary version of this paper was presented at *5th MIT Conference on Advanced Research in VLSI*, 1988).

[12] M. Nigam and S. Sahni. Computational geometry on a reconfigurable mesh. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 86–93, Los Alamitos, California, Apr. 1994. IEEE Computer Society.

[13] S. Olariu, J. L. Schwing, and J. Zhang. Optimal convex hull algorithms on enhanced meshes. *BIT*, 33(3):396–410, July 1993.

[14] D. B. Shu and J. G. Nash. The gated interconnection network for dynamic programming. In *Concurrent computations : algorithms, architecture, and technology* Proceedings of the 1987 Princeton Workshop on Algorithm, Architecture, and Technology Issues for Models of Concurrent Computation, pages 645–658, Princeton, New Jersey, 1988. Princeton University.

[15] J. F. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[16] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash. The image understanding architecture. *International Journal of Computer Vision*, 2:251–282, Jan. 1989.