Optimal Algorithms for Constrained Reconfigurable Meshes

B. Beresford–Smith O. Diessel H. ElGindy

Department of Computer Science
The University of Newcastle
Callaghan NSW 2308
AUSTRALIA

{*bbs, odiessel, hossam*}*@cs.newcastle.edu.au*

Abstract

This paper introduces a constrained reconfigurable mesh model which incorporates practical assumptions about propagation delays on large sized buses. Simulations of 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 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.

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 has control over a set of 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 this parallel model of computation).

Since the first papers [9, 15], research on the reconfigurable mesh architecture has gained considerable momentum and has also received some criticism. 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 [6, 1, 3] for a survey of the various models and algorithms). Recent examples include constant time algorithms for sorting n numbers [4, 5] and for determining the convex hull of n planar points [10, 11].

A common feature of the various 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 of these models, and cast a shadow of doubt on the feasibility of implementing a massively parallel machine based on a reconfigurable bus system.

Investigation of bus delays [8, 13] has indicated that the broadcast delay is small, but that it cannot be correctly modeled by a *constant* independent of the bus size. In this paper we report on our study of a new approach to coping with bus delay and 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 $\mathcal{A}^k$, that configure the bus system into components with sizes bounded by k bus–units to run on the model.

We give a detailed description of our reconfigurable mesh model in the following section. In section 3 we present optimal algorithms for sorting on constrained reconfigurable meshes, and then present optimal convex hull algorithms for constrained reconfigurable meshes with certain aspect ratios in section 4. Lower bounds are then discussed in section 5. Finally, 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, as in figure 1. 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 PE has control over a local set of short–circuit switches which allow the four I/O ports to be connected together in any combination. The 15 possible connection configurations are depicted in figure 2.

The processors operate synchronously, in one machine cycle performing an arithmetic, logic or

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

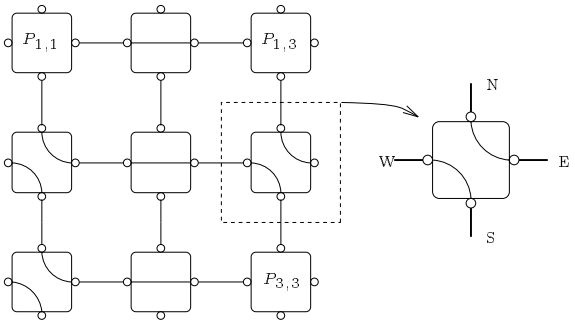

Figure 1: A reconfigurable mesh of size 3×3.

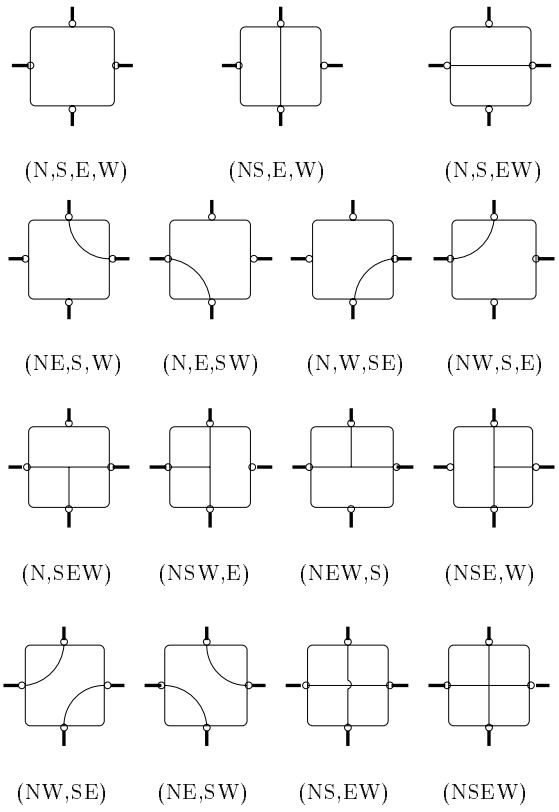

Figure 2: Reconfigurable mesh connection configurations.

control operation, setting a connection configuration, and sending (receiving) a datum to (from) each I/O port. Each processor possesses a constant number of $\Omega(\log mn)$–bit word registers allowing it to identify itself. 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 $P_{i,j} = P_{in+j}$. Block shuffled row–major ordering, as in figure 3, has the property that the first quarter of the PEs form one quadrant, the next quarter form another, and so on, with this property holding recursively, down to block length sequences of processors, which remain in row–major order.

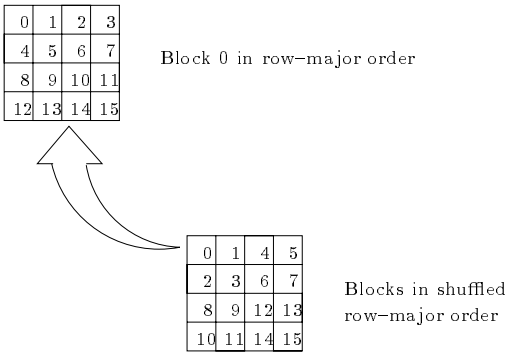

Figure 3: Block shuffled row–major order.

When a connection is set, signals received by a port are simultaneously available to any port connected to it. For example, if processors connect their northern and southern I/O ports by closing the appropriate switches as in the configuration (NS,E,W), data "broadcast" onto the "column bus" 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.

Unfortunately, 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. 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 a *k–constrained* reconfigurable mesh model in which connected buses of size at most k can be formed in any cycle. We use the notation $\mathcal{RM}_A^k$ to refer to a k–constrained reconfigurable mesh of area A.

A *linear* bus is a bus which never branches, thereby excluding configurations of the form (NSE, W) and (NSEW). Any reconfigurable mesh algorithm which uses only linear buses can be simulated by the k–constrained model by propagating signals k processors at a time. Any algorithm, which in $O(1)$ time broadcasts on a linear bus of length l, can therefore be simulated by a k–constrained algorithm in $O(\frac{l}{k})$ time. The simulation of a time optimal algorithm does not produce a time optimal solution for the k–constrained reconfigurable mesh, and unless the area of the mesh is reduced, the algorithm is no longer optimal according to the AT^2 metric popularized by Ullman [14].

3 Optimal Sorting Algorithms

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, 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 AT^2–optimal algorithms for sorting on $\mathcal{RM}^k_{nk}$ with varying aspect ratio (the ratio of the longer to the shorter side). The algorithms we use are extensions of those in [7]. In [7], an algorithm is given for sorting mn items on an $m \times n$ (standard) mesh of processors in time $O(m+n)$ using only a constant number of row and column operations. Assuming $p \leq q$, we show that this algorithm can be adapted to give a $O(\frac{p+q}{k})$–time algorithm for sorting n items on a $\mathcal{RM}^k_{nk}$ of size $p \times q$, which is AT^2–optimal. For ease of explanation, it will be assumed that k divides p, q and n, and that p and q divide n. Although these assumptions are fairly restrictive for practical applications, the results can be generalised, and are asymptotically identical.

3.1 Sorting on $\mathcal{RM}^k_{nk}$ of size $k \times n$

We first 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 will be used in the generalisation to k–constrained meshes with arbitrary aspect ratio.

Lemma 1 *Let $\mathcal{RM}^k_n$ be a linear array of processors $P_0, ..., P_{n-1}$ and let P_{ik} contain item x_i, $0 \leq i < \frac{n}{k}$. Then the items $x_0, ..., x_{\frac{n}{k}-1}$ can be sorted in $O(\frac{n}{k})$ time.*

Proof: A straightforward simulation of odd–even transposition sort on a linear array will sort the x_i. ∎

Lemma 2 *Sorting k items stored in the first row of $\mathcal{RM}^k_{k^2}$ of size $k \times k$ can be done in $O(1)$ time.*

Proof: It is easy to check that the algorithm in [5] for sorting k items on a $k \times k$ reconfigurable mesh uses only linear buses of length $O(k)$. Hence, the same algorithm can be used for the k–constrained reconfigurable mesh $\mathcal{RM}^k_{k^2}$ of size $k \times k$ to sort in constant time. ∎

In order to sort n items stored in the first row of $\mathcal{RM}^k_{nk}$ of size $k \times n$ we assume in the following that the items have been moved 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 $0 \leq s < \frac{n}{k}$.

In Marberg and Gafni [7] an optimal $O(m+n)$–time algorithm is given for sorting on an $m \times n$ standard mesh (where $\sqrt{n} \leq m$). The algorithm uses a constant number of row and column phases and is easily adapted for sorting on the k–constrained reconfigurable mesh. In outline, the algorithm consists of the following steps:

Procedure **RotateSort [7]**

1. Distribute the range of data contained in the mesh among its columns
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
4. Distribute the range of data contained in each $\sqrt{n} \times n$ slice of the mesh among the rows of the slice
5. Distribute the elements of each $\sqrt{n} \times \sqrt{n}$ block among the columns of the mesh
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 fixed–length sequence of sorts and rotations on the data contained, alternately, in rows and columns. A rotation results in a left (right) shift of each datum in a row (column). The datum at the end of a row (column) is moved to the opposite end as a result of the rotation, simulating a wraparound connection. Since a rotation permutes the data, it can be emulated by sorting the row (column). A rotation therefore takes as much time as it does to sort.

The algorithm of Marberg and Gafni [7] uses only the following basic operations:

1. Sort columns with m items;
2. Sort or rotate rows (in either direction) with n items;
3. Rotate the columns of each $\sqrt{n} \times n$ slice of the $m \times n$ mesh. Each such column has $\sqrt{n}$ items.

In the context of the $\mathcal{RM}^k_{nk}$ of size $k \times n$, the operations of the algorithm in [7] 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 operation dependent on the size of k as follows:

   (a) if $\frac{n}{k} \geq \sqrt{k}$, rotating rows with $\sqrt{k}$ items in each $k \times k\sqrt{k}$ slice of the $k \times n$ mesh which takes $O(\sqrt{k})$ time since rotation on a linear array is no harder than sorting on a linear array using *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 $\sqrt{\frac{n}{k}} \times n$ slice of the $k \times n$ mesh which can be done in $O(1)$ time for each column by using the $\mathcal{RM}^k_{\sqrt{nk}}$ of size $\sqrt{\frac{n}{k}} \times k$ which is available.

The total time is then $O(\frac{n}{k})$.

It follows from the above argument that:

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.*

3.2 Sorting on $\mathcal{RM}^k_{nk}$ of size $p \times q$

For a $\mathcal{RM}^k_{nk}$ of size $p \times q$ where $p \leq q$ the operations of the Marberg and Gafni algorithm lead to a corresponding algorithm *SORTPQ(n,k)* for sorting n items. In this case the mesh is divided into slices of size $p \times k$. By *Theorem 1*, the column operations can be done in $O(\frac{p}{k})$ time using the processors in each slice. The row and column phases can then be seen to take the following times:

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

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

3. A rotation operation dependent on the size of p and q as follows:

   (a) if $\frac{q}{k} \geq \sqrt{p}$, rotating rows with $\sqrt{p}$ items in each $p \times k\sqrt{p}$ slice of the $p \times q$ mesh which takes $O(\sqrt{p})$ time since rotation of a linear array is no harder than sorting a linear array (*Lemma 1*);

   (b) if $\frac{q}{k} < \sqrt{p}$ and hence $p > \sqrt{\frac{q}{k}}$ rotating columns with $\sqrt{\frac{q}{k}}$ items in each $\sqrt{\frac{q}{k}} \times q$ slice of the $p \times q$ mesh which takes $O(\sqrt{\frac{q}{k}}/k)$ time using the $\mathcal{RM}^k_{\sqrt{qk}}$ of size $\sqrt{\frac{q}{k}} \times k$ which is available for each column.

The total time is then $O(\frac{p+q}{k})$.

From the above argument it follows that:

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

4 Convex Hull Algorithms

The convex hull of a set $\mathcal{S}$ of n planar points is defined as the smallest convex region which contains all the points. The problem of computing the convex hull is that of identifying the extreme points which form the boundary of this convex region. The convex hull can be considered to consist of a convex chain of points lying above the line joining the westernmost and easternmost extreme points, the upper hull, and a similarly defined lower hull. To solve the convex hull problem, it suffices to compute the upper and lower hulls separately and then to concatenate the two chains. 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 [10, 11]. 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 not optimal for $k < n$.

In this section we present two optimal algorithms to solve the convex hull problem for $\mathcal{RM}^k_{nk}$ with differing aspect ratios. Both algorithms 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. For simplicity, it will be assumed that k divides n and $n \div k$ is a power of 2. Should this not be the case, the data could be padded, until n is sufficiently large, without affecting the asymptotic performance of the algorithm.

4.1 Convex Hull Computation on $\mathcal{RM}^k_{nk}$ of size $k \times n$

We use procedure *SupportLine* 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 expects L and R to be of size at most n, and computes the left and right endpoints of the line of support from R to L on a reconfigurable mesh of size $n \times n$ in a constant number of steps.

Procedure **SupportLine**

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

2. Similarly, 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 the point of R to the point of L.

4. Within each column, each processor containing a slope record checks whether the slope is locally minimal by checking its neighbours. Convexity ensures only one slope per column is identified.

5. Identify the maximum of the slope records found in the previous step using the method of [9].

end **SupportLine**

Theorem 3 *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.*

Proof: At the completion of Step 3, each point in R has determined the slope of the line to each point of L. For a given point in R, these are arranged in a column. The general position of points ensures that only one processor (point) in the column achieves a minimum slope, and the convexity of L ensures the processor containing this point can be identified by checking the values contained in the neighbouring processors to the north and south. By the end of Step 4, each point in R has identified a tangent to L. The tangent with maximum slope is the sought after line of support.

Steps 1 and 2 take $O(1)$ time since in the reconfigurable mesh model broadcasts along arbitrarily long buses takes a constant amount of time. Step 3 is an arithmetic step taking $O(1)$ time. Communication with neighbouring processors in Step 4 and the following comparisons require $O(1)$ time. Step 5 requires $O(1)$ time by Proposition 3 of Miller et al [9]. The total time for procedure *SupportLine* is therefore $O(1)$. ∎

Corollary 1 *Procedure* SupportLine *requires $O(\frac{n}{k})$ time in the k–constrained reconfigurable mesh model.*

Proof: Broadcasting buses of length $O(n)$ are formed, hence $O(\frac{n}{k})$ time is required to simulate procedure *SupportLine* in the k–constrained reconfigurable mesh model. ∎

We compute the extreme points on the upper hull of a set of planar points for this mesh organization as follows:

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 3.

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 then compute the upper hull of their subsets independently using the algorithm of Nigam et al [10]. 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 4. During each stage, odd–numbered components are paired with the following even–numbered components and their upper hulls, denoted by L 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:

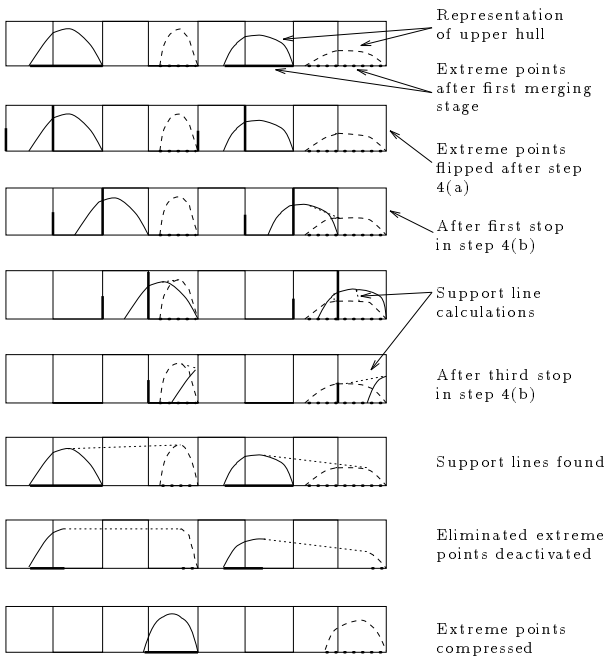

Figure 4: A second merging stage of procedure *UpperHull*.

   (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 4 *Procedure* UpperHull *correctly computes the upper hull of a set $\mathcal{S}$ 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 $O(\frac{n}{k})$. ∎

4.2 Convex Hull Computation on $\mathcal{RM}^k_{nk}$ of size $\sqrt{nk} \times \sqrt{nk}$

It is easy to see that any algorithm with $o(\frac{n}{k})$ running time must be executed on a $\mathcal{RM}^k_{nk}$ with smaller diameter. In this section we develop an optimal algorithm for a square $\mathcal{RM}^k_{nk}$ of size $\sqrt{nk} \times \sqrt{nk}$, which follows the same outline as that of procedure *UpperHull* of section 4.1. The algorithms differ mainly in the arrangement of blocks and in the method used to compute the support line of two disjoint upper hulls. Before describing the algorithm, we present the geometric properties and the basic mesh operations that are essential to the efficiency of the algorithm.

4.2.1 Preliminaries

Lemma 3 [2] *Let P and Q be two* disjoint *convex polygons and the line passing through the vertices $p \in P$ and $q \in Q$ be their upper support line, and let $P' = (a_1, a_2, \ldots, a_m)$ and $Q' = (b_1, b_2, \ldots, b_n)$ be convex sub-polygons of P and Q respectively. If the line passing through a_i and b_j is the upper support line of P' and Q', then at least one of the following statements is true:*

1. *p is a vertex of the chain joining a_{i-1} and a_i*
2. *p is a vertex of the chain joining a_i and a_{i+1}*
3. *q is a vertex of the chain joining b_{j-1} and b_j*
4. *q is a vertex of the chain joining b_j and b_{j+1}*

Lemma 4 *Let P and Q be two sorted sequences such that all elements of Q are larger than those of P. If the two sequences are arranged in row major order of two adjacent submeshes of size $\sqrt{m} \times \sqrt{m}$ in a k–constrained reconfigurable mesh such that there is one row of data for every k rows of processors, then the sorted sequence $P \cup Q$ can be arranged into row major order in the combined mesh of size $\sqrt{m} \times 2\sqrt{m}$ in $O(\frac{\sqrt{m}}{k})$ time.*

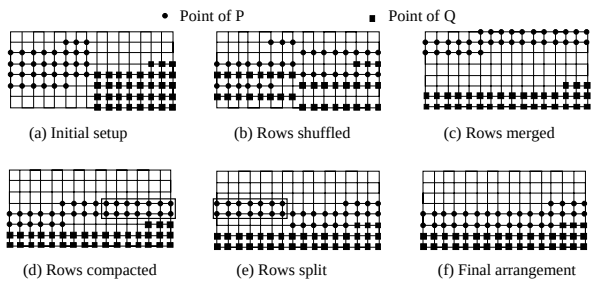

Figure 5: Merging sorted sequences P and Q.

Proof: (refer to figure 5 for illustration) We maintain $k-1$ free rows of processors for every row of data throughout the proof. Sequences initially setup as in figure 5(a) are arranged into row–major order as in figure 5(f) as follows:

1. Shuffle every second row to the opposite side of the combined mesh as in figure 5(b).

2. Merge the rows corresponding to sequence P towards the top of the combined mesh. Similarly, merge those from Q towards the bottom as in figure 5(c).

3. Move the sequence P down to the sequence Q as in figure 5(d).

4. Split the rows of P so as to fill the hole left in the last row of Q, whilst preserving the sequence of P as in figure 5(e).

5. Fill the hole left in each row of P with the segment from the row above.

Each step can be completed in $O(\frac{\sqrt{m}}{k})$ time in the k–constrained reconfigurable mesh model by pipelining rows of data k rows of processors at a time in steps 2, 3 and 5, and k–sized subsequences of data k columns of processors at a time in steps 1 and 4. ∎

A similar statement can easily be derived if the sequences P and Q are originally stored in rectangular meshes and need to be arranged into a square mesh.

Lemma 5 *If n points are stored in row–major order in a k–constrained mesh of size $m \times m$ with one row of data to every k rows of processors, the $\frac{m^2}{k^2}$ subsequences of k points can be moved into block shuffled row–major order in $O(\frac{m}{k})$ time.*

Proof: The arrangement can be achieved in $\log(\frac{m}{k}) - 1$ phases.

In the first phase, the points in the upper half of the upper right quadrant of the mesh are moved to the upper left quadrant, while the points of the lower half of the left upper quadrant are moved to the upper right quadrant. This step can be achieved by flipping sequences of k points from the bottom row of a $k \times k$ block into the left (right) column of a $k \times k$ block and pipelining the points k columns at a time to the left (right) in $\frac{m}{2k}$ steps.

The rows of points originally in the upper half of the left quadrant are now expanded to assume the position of every second row of data in the upper left quadrant, starting with the first row, while the rows of points originally from the upper half of the right quadrant are also expanded to take up every second data row in the upper left quadrant, albeit starting with the second row. Similarly, the rows of points originally in the lower half of the left quadrant are expanded to take up every second data row in the upper right quadrant starting with the first row, while the rows of points left in the lower half of the upper right quadrant are expanded to take up every second data row of the upper right quadrant beginning with the second data row. This movement takes $O(\frac{m}{4k})$ time.

Identical steps are simultaneously carried out on the lower half of the mesh and are applied recursively to each quadrant down to meshes of size $4k \times 4k$.

The time to shuffle the data is therefore:

$$Shuffle(m) = \sum_{i=2}^{\log(\frac{m}{k})} \frac{2^i}{2} + \frac{2^i}{4} + 2 \quad (3)$$

$$= O(\frac{m}{k}) \quad (4)$$

∎

4.2.2 The Algorithm

The main idea of the algorithm is to partition the set of points into $\frac{n}{k}$ groups and compute the upper hull of each group in a block of size $k \times k$. The $\frac{n}{k}$ disjoint upper hulls are then combined into larger groups in parallel merging stages. After each stage, the extreme points of each group are arranged into row–major order in the combined square or rectangular submesh. Such arrangement facilitates the efficient execution of the operations required for merging two upper hulls, namely, the identification

of equally spaced points, and the routing of subsequences through the constrained mesh.

Details of the procedure are as follows:

Procedure **UpperHullSM**

1. Load the n points by repeatedly presenting the first row with $\sqrt{nk}$ points and pipelining them through the mesh k rows at a time.

2. Sort the points into row major order on increasing x–coordinate using the procedure *SORTPQ(n,k)* of section 3.

3. Shuffle k–sized subsequences of points into block shuffled row–major order using the method of *Lemma 5*.

4. 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 then compute the upper hull of their subsets independently using the algorithm of Nigam et al [10]. Upon completion, extreme points are assigned labels in order of their appearance on the upper hull.

5. Compute the final upper hull by performing $O(\log(\frac{n}{k}))$ parallel stages to merge the $\frac{n}{k}$ disjoint upper hulls generated in the previous step as in figure 6. During each stage, odd–numbered components are paired with the following even–numbered components and their upper hulls, denoted by L and R, are merged. Recall that prior to the i^{th} stage, $1 \leq i \leq \log(\frac{n}{k})$, extreme points of L (R) are stored in row major order one row of data for every k rows of processors in a block of size $k2^{\lfloor \frac{i-1}{2} \rfloor} \times k2^{\lfloor \frac{i}{2} \rfloor}$.

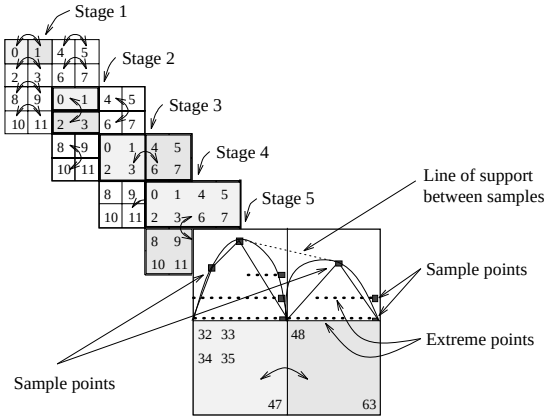

Figure 6: A fifth merging stage of procedure *UpperHullSM*.

Details of merging two upper hulls are as follows:

(a) Compute end points of the support line of L and R, which uniquely identify the upper hull of $L \cup R$, as follows:

i. Select a sample of elements, which partitions L into equal length segments, by choosing a point from each row in the block of processors storing L. Route the sample to the block storing R. The entire set of elements is selected if there is only one row of data.

A sample is simultaneously selected from R in a similar fashion.

ii. Compute end points of the support line of the two samples, using procedure *SupportLine*.

iii. If possible, identify the segments of L or R, which contain the endpoints of the support line of L and R as described in [2].

iv. Repeat with the newly identified segment(s) until the support line is computed.

(b) End points of the support line are communicated to processors of both L and R. The extreme points of $L \cup R$ are labeled and arranged into row major order using the method of *Lemma 4*.

end **UpperHullSM**

Theorem 5 *Procedure* UpperHullSM *correctly computes the upper hull of a set S of n planar points on a $\mathcal{RM}^k_{nk}$ of size $\sqrt{nk} \times \sqrt{nk}$ in $O(\sqrt{\frac{n}{k}})$ time, which is AT^2 optimal.*

Proof: Correctness of the procedure follows from simple geometric arguments.

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

$$\begin{aligned} CHSM(n,k) &= Load(n) + Sortpq(n,k) \\ &\quad + Shuffle(\sqrt{nk}) + Init \\ &\quad + Merge(n,k) \qquad (5) \end{aligned}$$

$$Merge(n,k) = \sum_{i=1}^{\log(\frac{n}{k})} Merge(k2^{i-1}, k) \qquad (6)$$

During the i^{th} merging stage of step 5, the procedure merges upper hulls L and R, each of which is the upper hull of $k2^{i-1}$ points stored in a mesh of size $k2^{\lfloor \frac{i-1}{2} \rfloor} \times k2^{\lfloor \frac{i}{2} \rfloor}$.

The operations of selecting a sample of L and routing it to the submesh storing R requires $O(2^{\lfloor \frac{i}{2} \rfloor})$ time. Procedure *SupportLine* requires $O(2^{\lfloor \frac{i}{2} \rfloor})$ time, and the identification of at least one row of L or R containing an endpoint of the support line, according to *Lemma 3*, requires further $O(2^{\lfloor \frac{i}{2} \rfloor})$ time. The geometric property of *Lemma 3* ensures that step 5(a) is executed at most four times, thus $O(2^{\lfloor \frac{i}{2} \rfloor})$ time is sufficient to complete step 5(a). Using the method of *Lemma 4*, step 5(b) can be completed in $O(2^{\lfloor \frac{i}{2} \rfloor})$ time as well. Therefore, *Merge*$(k2^i, k)$ of (6) can be performed in $O(2^{\lfloor \frac{i}{2} \rfloor})$ time. It follows directly that the running time of step 5 is $O(\sqrt{\frac{n}{k}})$.

To load, sort, and rearrange the data requires *Load*(n) = *Sortpq*(n,k) = *Shuffle*$(\sqrt{nk})$ = $O(\sqrt{\frac{n}{k}})$ time, and it takes *Init* = $O(1)$ time to perform the initial convex hull computations. It follows that the time complexity of *CHSM*(n,k) is $O(\sqrt{\frac{n}{k}})$. ∎

5 Lower Bounds

Using a bisection width argument, it can be shown that the VLSI complexity for sorting n numbers on a square mesh is $\Omega(n^2)$ [14]. As sorting is linear time reducible to the convex hull problem [12], it takes at least as much time to solve the convex hull problem as it does to sort. The recently developed constant time reconfigurable mesh algorithms for sorting and finding the convex hull of n elements on meshes of size $n \times n$ [4, 5, 10, 11] are therefore optimal with respect to T and AT^2 complexity measures. Since these algorithms only use linear bus configurations, they can be simulated on a k–constrained reconfigurable mesh of size $n \times n$ using $O(\frac{n}{k})$ time, which is no longer AT^2 optimal for $k < n$. We are therefore motivated to develop solutions using less area.

When the solution mesh is no longer square, the AT^2 complexity needs to be scaled by the aspect ratio [14]. 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 $\Omega(\frac{n}{p})$. Our algorithms for sorting and for finding the convex hull of n planar points on a $\mathcal{RM}^k_{nk}$ of size $k \times n$ run in $O(\frac{n}{k})$ time, which matches the lower bound when k replaces p. Both algorithms have an AT^2 complexity of $O(\frac{n}{k} \times n^2)$, which matches the lower bound when k and n are substituted for p and q. $SORTPQ(n,k)$ can be shown to be T and AT^2 optimal by a similar argument. The second convex hull algorithm for the $\mathcal{RM}^k_{nk}$, of size $\sqrt{nk} \times \sqrt{nk}$, has a running time of $O(\sqrt{\frac{n}{k}})$ and an AT^2 complexity of $O(n^2)$, both of which are optimal.

6 Concluding Remarks

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. 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 $\mathcal{A}^k$. We are also interested in developing a convex hull algorithm for arbitrary aspect ratios, which we see as being of practical use in general purpose computing environments where the mesh area available to a task may be constrained.

Acknowledgements

This research is supported by grants from the University of Newcastle RMC, the Australian Research Council, and an Australian Research Council Postgraduate Scholarship.

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] M. J. Atallah and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, Oct. 1988.

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

[7] 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.

[8] 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.

[9] 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).

[10] 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.

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

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

[13] 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.

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

[15] 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.