2-D SIMD Algorithms in the Perfect Shuffle Networks

Yosi Ben-Asher, David Egozi & Assaf Schuster

The Hebrew University

Abstract

This paper studies a set of basic algorithms for SIMD Perfect Shuffle networks. These algorithms where studied in several papers, but for the 1-D case, where the size of the problem N is the same as the number of processors P. For the 2-D case of $N = L * P$, studied by [GK-80] and [Kr-81], we improve several algorithms, achieving run time $O(L + \log P)$ rather than $O(L * \log P)$, as N exceeds P. We give non-trivial algorithms for the following 2-D operations: Row-Reduction, Parallel-Prefix, Transpose, Smoothing and Cartesian-Product.

KEYWORDS: Perfect Shuffle, Networks, SIMD.

1 Introduction

Schwartz [Sc-80] introduced the idea of a parallel processor based on the *Perfect Shuffle* connections [St-71], reviewed various basic SIMD algorithms for such an ensemble of processors, and analyzed their asymptotic time complexities. These algorithms are designed for the 1-D case, where N, the size of the problem, is restricted to be P, the number of processors. In reallity, the general case where $N > P$ is much more likely to appear. In this case each processor stores $L = N/P$ data elements in its local memory. This data structure is referred to as a *2-D array* of P columns and L rows, where the i'th column corresponds to the local memory of the processor PE_i. Algorithms for the 2-D case were introduced in [Kr-81] and [GK-80], where they are called *supersaturated*. Here we improve several of the algorithms presented there and proceed to complete the list of efficient 2-D algorithms.

We present algorithms for the following problems (in section order):

2) Row-reduction - apply some operation on all elements of each row and collect the results in first processor.

3) Parallel prefix by rows - same as row reduction, except that all partial results are also generated.

4) Transpose rows to columns and vice versa.

5) Smoothing - move elements to make all columns heights the same.

6) Cartesian product of groups of elements.

1.1 The Model

Our model is a SIMD multicomputer with shuffle-exchange interconnections. More formally, we have a set of P processors numbered $0, ..., P-1$. The j'th processor will be denoted PE_j. For simplicity, we assume that P is a power of 2. Subgroups of processors are referred to in a "natural" manner: left half, right half, odd, even etc. Each processor is connected to 3 other processors. it is easiest to describe the topology of these connections by the relations between the IDs of the connected PEs. Given the ID of a certain PE,

- the shuffle (σ) connects to the PE whose ID is a cyclic left shift of one bit,
- the unshuffle (σ^{-1}) connects to the PE whose ID is a cyclic right shift of one bit,
- the exchange (EX) connects to the PE whose ID is the complement of the least significant bit.

Note that for PE_0 and PE_{P-1} the shuffle and unshuffle connect to itself. All other processors have in and out degree 3.

At every step of the computation, all the processors perform the same operation. This can either be a send or receive on one of the interconnections lines (same one for all processors), or a (fixed size) local computation. Of course, each processor uses its own local data.

Rounds	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8	PE_9	PE_{10}	PE_{11}	PE_{12}	PE_{13}	PE_{14}	PE_{15}
round 8	5	5														
round 7	4	4	5	5												
round 6	3	3	4	4	5	5	5	5								
round 5	2	2	3	3	4	4	4	4	5	5	5	5	5	5	5	5
round 4	1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
round 3			1	1	2	2	2	2	3	3	3	3	3	3	3	3
round 2					1	1	1	1	2	2	2	2	2	2	2	2
round 1									1	1	1	1	1	1	1	1

Fig. 1. 2-D Row-Reduction on 16-Processors.

2 Row-Reduction Operation

The 2-D reduction operation calculates the result of some binary associative operation *bop* (such as +, min, etc) for each row of the $L \times P$ array. The results are stored in the first processor, thus we refer to this operation as "all-rows to one-column" operation.

In [GK-80] this operation is referred to as "summing by columns" (pp. 23) and the naive way of summing the rows sequentially, using a 1-D algorithm for each row, is used, resulting in $O(L * \log P)$ steps run time. In order to get an efficient 2-D algorithm we process the rows in a pipelined fashion. The algorithm advances in $L + \log P - 1$ rounds. The rounds are equivalent except for the following:

- The first round is a short one and consists of steps (3) and (4) only, to be described next.
- At the i'th round, $i = 1, ..., \log P$, processors $0, ..., 2^{\log P - i} - 1$ are inactive.
- At the i'th round, $L < i < L + \log P$, processors $2^{\log P + L - i}, ..., P - 1$ are inactive.

The i'th round, $i > 1$, consists of four steps:

(1) Every even processor sends the step (4) result to its σ^{-1} neighbor.

(2) Every processor which got a new value at step (1), computes *bop* for this value with its own value of the i'th row. For the next round the result is its new value of the i'th row.

(3) For every odd processor, PE_j, PE_j moves the value of the i'th row to PE_{j-1} using the EX connection. (Note that for each individual processor the round number (i) is the number of rounds it has been active so far.)

(4) For every even processor compute *bop* for its i'th row value, and the value it received at step (3).

At each round a new row becomes "active", and its remaining number of elements starts to decrease. Consider a row, k, which is currently being processed. Note that the set of processors still containing elements of k is consecutive and starts from PE_0. Let K_l (K_r) denote the left (right) half of this set. At steps (3) and (4) of each round, K_r is decreased by half, with the remaining half stored at the even processors. At steps (1) and (2) (of the following round) this half is folded on and *bop*'ed with the right half of K_l, so K_r is "free" to proceed with the next line.

This pipeline process is demonstrated in Fig. 1, the numbers denote the row number which a PE is currently processing. The table is for sixteen processors, $L = 5$. Note the results accumulating in PE_0, starting with the first row at round 4.

The reduction operation requires $4(L+\log P)$ steps. The reverse sequence of steps can be used for broadcasting a column stored at PE_0 to the "rest of the world". This is a 2-D column broadcast operation which is useful for many algorithms.

The reverse operation is handy not only for duplication but also for generating different values. Note that 2-D reduction spans a binary tree for each row. By reversing its steps we are actually traversing the tree topdown. We name *rop* some function $rop(x, y) \longmapsto (l, r)$ of a local value x and an inherited value y to create two different values l, r, for the left and the right sons.

Let each processor store L elements in its local memory. The multiple-broadcast operation is defined as the distribution all elements to all processors. The following is an $O(L * P)$ steps multiple broadcast algorithm (Clearly $\Omega(L * P)$ steps are required): In $L * (P - 1)$ steps all elements arrive at PE_0. Then, using reverse 2-D reduction, PE_0 broadcasts all $L * P$ elements to the rest of the world in $4(L * P + \log P)$ steps.

3 Parallel Prefix by Rows

There is a trivial $O(L + \log P)$ PP by columns algorithm but the best known algorithm for PP by rows requires $O(L * \log P)$ steps [Kr-81]. We close this gap with a PP by rows algorithm which requires only $O(L + \log P)$ steps.

The algorithm uses a variant of the 2-D reduction (see section 2). Observe that the reduction algorithm virtually constructs a binary tree above the right half of the row of processors. Skipping step (4) of the reduction iterations, i.e. avoiding adding the local values of the internal nodes of that tree, we get a binary tree containing all partial sums of the right half. For the i'th row we denote that tree as T_i^R. We denote by T_i^L the tree obtained by executing the symmetric algorithm for the left half, note that the root value is stored at PE_{P-1} rather than PE_0. It is important that both trees preserve the processor order in the following sense: if an internal node contains the sum of the values of PE_n, ..., PE_m then its left son contains the corresponding sum of PE_n, ..., $PE_{(n+m-1)/2}$ and the right son contains that sum for $PE_{(n+m+1)/2}$, ..., PE_m.

The algorithm (and its validity) is based on the following easy observations:

(i) If we have such an ordered binary tree for the complete row, we could easily get 1-D PP for the i'th row. This is done by traversing the tree top down, where each father hands its inherited value to its left son and its inherited value plus its left son's value to its right son.

(ii) The same idea of (i) is correct when the two halves of the tree are available separately, and the root of the right half gets the value of the root of the left half.

(iii) Finally, note that given the sum of the lower indexed rows, the i'th row 2-D PP can be achieved by the same idea of (i) and (ii).

The algorithm advances as follows:

(1) Use the variant of 2-D reduction to compute T_i^L for all i in $4(L + \log P)$. Repeat the same computation for T_i^R. All processors save up to $2L$ partial sums to use at phase (5) of the algorithm below.

(2) In $L+2\log P$ steps move the column of step-(1)'s results, stored in PE_{P-1}, to PE_0.

(3) PE_0 computes the total sum of each row and PP for the column of sums. This takes $2L$ steps.

(4) In $L+2\log P$ steps move the resulted column at step (3) to PE_{P-1}.

(5) Reverse the tree constructions made at step (1), computing 2-D PP for each row, as described in the above observations.

4 The Transpose Operation

The transpose operation transposes the elements of a 2-D array $L \times P$ from row order into column order (T_{rc}) or from column order into row order (T_{cr}). Suppose we have $L * P$ array elements numbered $1, ..., L * P$. Row order means that the numbering of elements increases first by rows and then by columns, i.e. PE_i contains the following elements $\{i + p * j + 1\}_{j=0,...,L-1}$. In column order PE_i contains $\{i*L+1, ..., i*L+L\}$. Note that unless $L = P$, $T_{rc} \neq T_{cr}$.

We present a general algorithm for the 2-D case in $O(L*\log P)$ steps. When $L \geq P$ we could use the 2-D sorting operation, as described in [GK-80]. However we would get the same order of time for the transpose operation. Our algorithm holds for any L and is far better in terms of constants.

We first describe the Transpose Rows to Columns operation, T_{rc}.

We denote the location of an element by $[i, j]$, where j is the processor location and i is its local memory location. The new location is $[i', j']$ where :

$$j' = \left\lfloor \frac{i * P + j}{L} \right\rfloor \quad i' = i * P + j - (j' - 1) * L \quad (1)$$

Assume for simplicity that L is a power of two. Let $< i, j >$ denote the binary representation of $i * P + j$. From (1) j' is equal to $< i, j >$ shifted right $\log L$ bits. Thus j' is equal to the $\log P$ most significant bits of $< i, j >$.

Moving from j to j' is done by $\log P$ iterations of updating the least significant bit of j (EX step), followed by a σ^{-1} step to cyclicly shift the bits to the right. Moving elements between columns, without careful selection may cause collisions and congestion while correcting the bits.

Partition the L elements of each processor into P groups of L/P elements each, such that the elements of each group have the same leftmost $\log P$ address bits. The i'th group at the j'th processor is destined to $j \oplus i$, where $\oplus$ denotes the binary *xor* operation.

The elements in the first group are destinated for the current processor, therefore only local transformations are required. For the second group in each processor we need to correct one bit, so one step is sufficient. In general, for the i'th group we need to correct up to $\lceil \log i \rceil$ bits, so $2 * \lceil \log i \rceil - 1$ steps are required. In every even step σ connection is used and in every odd step either EX connection is used or the element stays in place. We say that a *collision* occurs when in some step some processors use EX and others do not. That is, when collision do not occur during some odd step, the number of elements in

all processors do not change.

The algorithm advances in phases. In a single phase every processor sends one element of the same group i to its destination. The following phase starts when all of these elements arrive. The key observation is that all of these elements need to correct address bits at the same locations. We conclude that there can be no collisions during a phase, since all elements originate at different processors and since in odd steps either *all* of them use EX to complement the right bit or *none* of them does.

The complete algorithm sends elements by the group order. The i'th group completes after L/P phases, each phase takes at most $2 * \lceil \log i \rceil - 1$ steps. Note that an "odd" step in which no EX connection is taken may be skipped.

For $L \geq P$ the algorithm is optimal since at each step P edge traversals are taken and shortest paths are used. Counting the required address bit changes show that the algorithm terminates after $\frac{3}{2} * L * \log P$ steps.

The Transpose Columns to Rows operation, T_{cr}, can be used to scatter local results of every processor to all other processors. We use the same method as for T_{rc} with the following differences:

$$i' = \left\lfloor \frac{j * L + i}{P} \right\rfloor \quad j' = \frac{j * L + i}{P} - i'$$

Recall that j is the processor number and i is the row number. j' is the reminder $< j, i > /P$, thus j' is the $\log P$ rightmost bits of $< j, i >$.

We would like to show a lower bound of $\Omega(L * \log P)$ for the transpose operation. We call two processors *far* when at least $\log P - 3$ edge traversals are necessary to reach one from the other. Since the network degree is at most 3, each processor has more than $P/4$ of the other processors far from it. If $L = P$ the transpose is a 2-D permutation, where every processor sends an element to all other processors. In such a case a total of at least $P * P/4 * \log P$ edge traversals are necessary for the completion of the transpose operation. Hence the operation terminates after at least $P/4 * \log P$ steps, since at most P such edge traversals are possible at a single step. Thus the lower bound follows for $L \geq P$.

5 The Smoothing Operation

This operation moves elements among columns, to make the height of each column the same. It can serve as a load-balancing operation. Denote by $L = L_{max}$ the length of the highest column, L_{min} the length of the lowest one. Denote by D_{max} the biggest difference of all column size differences at odd-even pairs (i.e. pairs connected by an EX). An $O(L + \log L * \log P)$ approximate smoothing algorithm for certain families of expanders was presented in [UP-87].

Smoothing is also packing, so smoothing can be completed in $O(L * \log P)$ using the packing algorithm from [GK-80]. However, since the preservation of order is not necessary, we hope to do better. First, a rather straightforward $O(L * \log P)$ algorithm is presented.

5.1 Simple Smoothing Algorithm

The algorithm incorporates (at most) $\log P$ iterations. At the i'th iteration:

(i) Find D_{max}, broadcast to all processors. Use EX connections to move elements such that every pair of odd-even processors has an equal size column. This takes $D_{max}/2$ steps.

(ii) Find L_{min} and L_{max} and broadcast these to all processors. If $L_{min} = L_{max}$ then stop. Each processor now recomputes its column size to be the actual current value minus L_{min}. L_{max} is updated accordingly. In L_{max} steps each processor uses the σ^{-1} connection to send his column.

The correctness of the algorithm is due to the fact that the σ^{-1} connections "unshuffles" 2 consecutive odd-even pairs onto new 2 odd-even pairs. Hence the number of equal height columns doubles at each iteration.

Suppose only one column is of size L and the rest are zero's. Then after $2 * L + 4 * \log^2 P$ steps the size of each column is L/P. At the other extreme case, let the left half of the processors have columns of size L and the right half has columns of size zero. It is easy to see that at every phase there exist columns of size L and columns of size zero, so in this case the algorithm takes $L * \log P + 4 * \log^2 P$ steps.

This algorithm is subject to several "small" improvements. However it has a fundamental characteristic which makes it slow: it is nonadaptive in the sense that elements move in one direction only on σ^{-1} edges, full columns are being moved in cases where this may be superfluous and local information only is taken into account.

5.2 The Support Algorithm

The following "Support" algorithm seems to be free of the Simple smoothing algorithm flaws. We had no success in proving its optimality, so we present simulation results (see section 5.3). All the cases that

we checked were smoothed by the Support algorithm (up to 3 times) faster than by the Simple algorithm. For any reasonable scope of P, Support proves to be very efficient.

Let L_{AV} denote the final column height. The algorithm consists of $\log L_{AV}$ identical phases, each involving $\log P$ rounds. After the first phase every processor has at least half $(L_{AV}/2)$ of its final set of elements. In every subsequent phase, the number of elements missing at the lowest column processor is decreased by half. Moreover, the height of the highest column is also decreased by half at the termination of each phase, so in terms of the highest column size, the Support algorithm performance is twice the performance of the first phase. We describe in details the first phase.

At the beginning of the phase, the EX connections are used for equalizing the sizes of "pairs" of odd-even columns. Now each column is split into two halves. We call one collection of all halves of all columns H_1 and the other H_2. Note that

$$|H_1| = |H_2| \tag{1}$$

and that is how it is going to stay throughout the phase. Note also that, for both sets

The total number of elements residing at odd processors is equal to that of even processors. (2)

Although the actual set of elements of H_j change during the run of algorithm, the notion of which elements belong to H_1 at each step and which to H_2 will be obvious.

Let i' denote $\log P - i$. Let S_i denote the collection of all sequences of $2^{i'}$ consecutive processors. By a "sequence" of processors of length 2^m, we always mean that counting starts at one of the processors $0, 2^m, ..., P - 2^m$. The phase advances in rounds: at the end of the i'th round $H_{i(2)}$ ($H_{i \bmod 2}$) is evenly distributed among the elements of S_i. In other words, if $L_{l,i(2)}$ denotes the number of elements in PE_l belonging to $H_{i(2)}$, then for all $k = 0, 2^{i'}, ..., P - 2^{i'}$ (zero-sum for sequence starting at PE_k)

$$\sum_{l=0}^{2^{i'}-1} (L_{k+l,i(2)} - L_{AV}/2) = 0. \tag{3}$$

Clearly after $\log P$ rounds smoothing of one of the H sets is completed.

During the i'th round we use elements of $H_{i-1(2)}$ to "support" $H_{i(2)}$ such that (3) holds. Note that (3) implies (1). Then $H_{i(2)}$ uses the EX connections so that (2) holds. Thus, at the end of the i'th round, both (3) and (2) hold for $H_{i(2)}$. This implies a strengthened versions of (3): for all $k = 0, 2^{i'}, ..., P - 2^{i'}$ (zero-sum for even processors in the sequence starting at PE_k)

$$\sum_{l=0}^{2^{i'}/2-1} (L_{k+2l,i(2)} - L_{AV}/2) = 0, \tag{4}$$

and (zero-sum for odd processors in the sequence starting at PE_k)

$$\sum_{l=0}^{2^{i'}/2-1} (L_{k+2l+1,i(2)} - L_{AV}/2) = 0. \tag{5}$$

Finally observe that, using σ^{-1} connections, a subsequence of odd (or even) processors of some sequence in S_i is connected to a single sequence of S_{i+1}. Also every sequence in S_{i+1} is "covered" by such subsequence. At the beginning of round $i+1$ this property is used so $H_{i(2)}$ "supports" $H_{i+1(2)}$ such that (3) holds for $i+1$, and so on and so forth.

We now describe how $H_{i(2)}$, obeying (5) and (4) at the beginning of the $i+1$'th round, "supports" $H_{i+1(2)}$ so that (3) holds. Consider a forest which is all subtrees of height i' of a complete tree of height $\log P$. Each subtree calculates the difference between the total number of elements of $H_{i+1(2)}$ in a sequence of S_{i+1} to its corresponding total number of elements of $H_{i(2)}$ in the corresponding odd (or even) subsequence of S_i. This difference determines how many elements should $H_{i(2)}$ move to $H_{i+1(2)}$ (or vice versa) through the σ^{-1} connections. Going "back", down the tree, each father splits the work of moving elements between its children and according to their partial sums. This procedure of "control" takes at most $\sim 8i'$ steps. The total time spent in the control procedure throughout the run of the algorithm is $O(\log^2 P * \log L_{max})$.

Our last observation is that the control procedure may run independent of moving the elements. It is actually performing a simulation of the algorithm itself, given the initial number of elements at each processor as input and giving the total flow and direction of flow at every edge as output. Thus, to improve the total run time, we first perform the control procedure, saving the data about the movement of elements on every edge. Then, for each edge, a number which is the total sum of elements traversing it is calculated, where two opposite traversals cancel each other. Only then the flow of elements starts: a processor sends an element using an out-edge if (and only if) it has an element and the edge "control number" is positive. The intuition motivating this two parts separation is to let the control procedure gather global information which pays in optimizing the flow part.

5.3 Simulation Results

We know the control procedure takes $O(\log^2 P * \log L_{max})$ steps. The question remaining is: how fast is the flow part? The flow is carried out in an optimal pipeline, what about the involved paths? To gain more information we ran simulations of the algorithm, measuring the time of the flow part only. Early results showed that the worst initial distributions consist of $P/2$ columns of height $L = L_{max}$ and the rest of height 0. Since we are interested in worst case analysis, these are the initial distributions we have chosen, where the location of the full/empty columns is randomly chosen.

The scope of P, achievable by our Sun-3 workstation, is limited to 2^{17}. However, this range turned out sufficient in getting a clear tendency of the results which, if does not hold for all P, is probably kept up to at least $P = 2^{25}$. We tried 2 sizes of L: 1024 and 4096. To normalize, the total number of steps was divided by L. For each value of $\log P = 8, ..., 17$ the average of 10 experiments is presented. Most deviations from the averages presented are small: the highest of them, 0.55, is a real exception.

The results suggest a linear growth of the flow part run-time and $\log P$, so the algorithm behaves asymptotically as $L * \log P$. Note, however, that the constants make the algorithm competitive with a $5 * L$ algorithm at the feasible range of P (taking the control part into account). Moreover, when the number of columns is much less or much more then $P/2$ or when it is not a zero/full-column distribution then the performance is a lot better. Results of the simulations are presented in figure 2.

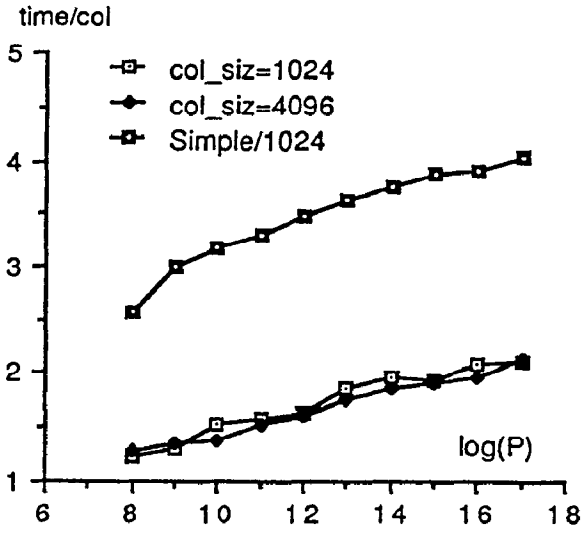

Figure 2: Simulation Results

5.4 A Smoothing Lower Bound

To achieve a smoothing lower bound one needs a better insight of the Perfect Shuffle structure. This can be done as follows: the $P = 2^n$ nodes of the network are split into $n+1$ disjoint sets: $PS_0, ..., PS_n$. PS_i is the set of processors having exactly i set address bits. The relative size of the sets follows the binomial distribution. The links between the sets PS_i and PS_{i+1} are exactly all the EX edges originating at nodes in PS_i having a "0" rightmost address bit. Analogously, the PS_i to PS_{i-1} links originate at nodes in PS_i having a set rightmost address bit. σ edges always connect processors in the same PS-set. Clearly, to move from a processor in PS_i to a processor in PS_j $(i < j)$, one has to visit at least two processors at each of $PS_{i+1}, ..., PS_{j-1}$ in that order.

Suppose half of the processors, say, all the processors having at most $n/2$ set address bits, have a full column size L_{max} and the other half are "empty". Then smoothing requires $L_{max}/2 * P$ elements be moved through $PS_{n/2}$. However, there are only

$$\binom{n-1}{n/2} \approx \frac{1}{2} * \frac{1}{\sqrt{2\pi n}} * 2^n = \frac{1}{2\sqrt{2\pi n}} * P$$

edges connecting $PS_{n/2}$ to $PS_{n/2-1}$, a bottleneck of factor $\frac{1}{\sqrt{\log P}}$. Hence $\Omega(L * \sqrt{\log P})$ steps are necessary to complete the smoothing operation.

Consider a node of the PS network and all the nodes it can reach using σ connections only. This defines a "ring" of processors which we call a *necklace*. The internal structure of the PS-sets consists of disjoint, disconnected necklaces. Suppose bits are moving through a necklace in the following fashion: a necklace edge can carry 1 bit per second, an even necklace node is the exit of 1 bps and an odd necklace node is the entrance of 1 bps. Then the number of bps moving through the necklace is exactly the number of all the odd-even pairs of necklace processors which are neighbors. This observation, however, applied to the total flow through $PS_{n/2}$, does not change the asymptotics of the previous estimate.

5.5 The Dup Operation

The 2-D dup operation duplicates each element, e, producing extra copies according to a tag number $(e.c)$ associated with e. The resulting total number of elements is $D = \sum_{\text{all } e} e.c$. Note that there is no requirement on the final distribution of the copies. Let $D_i = \sum_{e \text{ in } PE_i} e.c$ and L_i the number of elements in PE_i before duplication.

For the 1-D case Schwartz [Sc-80] suggested to use PP to determine the place of each element in the new array, then use sorting to move each element to its place, and finally PP by groups to duplicate each element in the space between every two original elements.

For the 2-D case smoothing is used to equally distribute the duplication work among the processors. Then D/P more steps are required to complete the duplication at each processor locally. Note that in the course of the smoothing operation it may be necessary to duplicate e, tagged $e.c$, to e_1 and e_2 and assign them with new tags $e_1.c'$ and $e_2.(e.c - e_1.c')$. This duplication might cause the size of (real) columns, L_i, to increase which might increase the smoothing runtime too. However, in the next subsection a way is shown how to equalize D_i and D_j for two adjacent columns, where L_i and L_j increase by at most 1. Hence the real column size increases by at most $\log P$ so the total dup algorithm runtime is $O(\text{Smooth}(L) + D/P)$, where L is the size of the highest column *before* duplication and $\text{Smooth}(L)$ is the cost of the smoothing operation. When $D/P > L$, this algorithm performs better than the minimum of $O(D/P * \log P + \log^2 P)$ steps required by adopting the 1-D algorithm to the 2-D case.

5.6 Dup - Smoothing

Given two adjacent processors, PE_i and PE_j, it remains to show that their "D-heights" can be equalized such that their "L-heights" increase by at most 1. It is easier to see for the Simple smoothing algorithm (section 5.1). Before the beginning of the algorithm each column is sorted by tags. The columns are kept sorted from this point on, throughout the algorithm. Consider step (i) where all odd-even pairs of processors equalize their column D-heights. Let PE_i, PE_{i+1} be such a pair.

- While the L-heights are different, move elements from the higher column to the lower one, preserving the internal column order sorted by tags.
- W.l.o.g. $D_i > D_{i+1}$. Denote $Top(i)$ the element with the largest tag in PE_i's column and $Bottom(i+1)$ the element with smallest tag in PE_{i+1}'s column. Obviously $Top(i) > Bottom(i+1)$. Preserving the internal column order sorted, PE_i and PE_{i+1} exchange these two elements. Keep exchanging until D_{i+1} becomes larger then or equal to D_i. If $D_{i+1} > D_i$, then the last element moved from PE_i to PE_{i+1} can be split into two, such that the return of one of them to PE_i makes the D-heights equal.

For the new step (i) of the simple smoothing algorithm, the final L-heights are at most the average of the L-heights of initial columns plus one. Moreover, the preservation of order is easily kept, using a merge-like scan of the columns, in $O(L)$ steps. The overall algorithm runtime is $O(L * \log L + L * \log P + \log^2 P)$, regardless of the D-heights.

6 Cartesian Product

Assume that the elements in the 2-D array are divided into two groups: A and B. Then the Cartesian product (CP) $A \times B$ operation produces all pairs a, b, such that a belongs to A and b belongs to B. There is no restriction on the final distribution of pairs.

In [Sc-80] there is an algorithm for 1-D CP for groups of size at most $P^{\frac{1}{2}}$. A naive adaptation of this algorithm to the 2-D case requires at least $O(\frac{|A|*|B|}{P} * \log P)$ steps. Suppose $|A| > |B|$ and $|A| > P$, and assume A is smoothed, i.e. each processor has $\frac{|A|}{P}$ of A's elements. The following algorithm achieve 2-D CP of A and B in $O(\frac{|A|*|B|}{P} + |B|)$ steps:

(i) Use 2-D broadcast to produce a copy of each element of B in each processor. This takes at most $2 * (|B| + \log P)$ steps.

(ii) Locally, each processor creates a, b for every a which belongs to A's elements stored in it and every b in B. This takes $\frac{|A|*|B|}{P}$ steps.

6.1 Cartesian Product by Groups

Given list of CPs CPBG= $\{A_1 \times B_1, ..., A_r \times B_r\}$, the Cartesian product by group (CPBG) operation produces all pairs of all the products in CPBG. The following assumptions are natural for many cases:

- r is "sufficiently" small, i.e. $r \ll P$ and $r \ll L$.
- The CPBG is initially known for all processors, including group sizes. Here we also assume that CPBG is given such that all processors discriminate groups appearing at the "B-side" from those appearing at the "A-side". The CPBG is stored in the following data structure: for every "A-group" A_j there is an ordered list L_j containing all "B-groups" that are "taken product" with A_j in CPBG, i.e. G is in L_j if and only if $A_j \times G$ belongs to CPBG.
- The groups are mutually disjoint, i.e. elements of different groups may have same value but always different "id"s.

Operation	Perfect Shuffle Time	Lower Bound
Row-Reduction	$4(L+\log P)$	
Parallel-Prefix	$20(L+\log P)$	
Transpose (T_{rc} and T_{cr})	$1.5*L*\log P$	$\Omega(L*\log P)$
Smoothing	$L*\log P$	$\Omega(L*\sqrt{\log P})$
Dup	$D/P+O(L*\log P)$	D/P
Cartesian Product of A and B	$\frac{\lvert A\rvert*\lvert B\rvert}{P}+5*\lvert B\rvert$	$\frac{\lvert A\rvert*\lvert B\rvert}{P}$
Cartesian Product by Groups	$L*\log P+\sum(\lvert B_i\rvert+\frac{\lvert B_i\rvert*\lvert A_i\rvert}{P})$	$\sum\frac{\lvert B_i\rvert*\lvert A_i\rvert}{P}$

Figure 3: Table of Results

The following CPBG algorithm is a close variant of the CP algorithm:

(i) All groups at the "B-side" are broadcasted to all processors, not repeating the same group-broadcast twice. Note that this enables an ordering of the elements of each "B-group" consecutively, which is equivalent at all processors.

(ii) Let e be an element of a group A_j where A_j is at the "A-side". Let $|L_j|$ denote the total number of elements of all groups in L_j. e is associated with a label containing its group id and a range: $e.A_j. < 1,...,|L_j| >$. The range of e corresponds to the (ordered) elements from the (ordered) groups of L_j with which it is to be paired.

(iii) The labeled elements are "smoothed" by a variant of the smoothing incorporated in the dup operation (section 5.6). Whenever the smooth operation requires a splitting of $e.A_j. < x,...,y >$ into e_1 and e_2, the range of e is split into disjoint "segments": $e_1.A_j. < x,...,z >$ and $e_2.A_j. < z+1,...,y >$. The result of this step is an even distribution of the "pairing" work among all processors.

(iv) Each processor generates all pairs of "A-elements" with their "B-partners", specified by their labels.

The total number of steps required is at most $L_A * \log P+\sum_{i=1}^{r}(|B_i|+\frac{|B_i|*|A_i|}{P})$ where L_A is the highest column at the "A-side".

7 Discussion and Summary

Figure 3 summarizes performance of 2-D algorithms presented in this work. Only dominating terms are given. It turned out that the efficient 2-D algorithms are those related to problems which may involve some computation of values but do not involve (sets of) permutations. On the other hand, problems for which the best algorithm is provably inefficient, such as the packing or transpose operations, clearly are special cases of 2-D permutations. It is interesting to refer to smoothing not as a specific 2-D permutation, but rather as a collection of such, from which one has to be chosen. The question remains open wether smoothing can be done more efficiently.

References

[Sc-80] J. T. Schwartz: *Ultracomputers*, ACM TOPLAS 2, pp. 484-521, 1980.

[St-71] Harold S. Stone: *Parallel Processing with the Perfect Shuffle*, IEEE Trans. C-20, pp. 153-161, 1971.

[Kr-81] Clyde P. Kruskal: *Upper and Lower Bounds on the Performance of Parallel Algorithms*, Ph.D. Dissertation, Courant Inst., NYU, 1981.

[GK-80] A. Gottlieb and C. Kruskal: *Supersaturated Ultracomputer Algorithms*, Ultracomputer note #11.

[Be-88] Y. Ben-Asher: *M&P A Non Procedural Parallel programming Language Based on Set Theory*, Ph.D. dissertation, Hebrew University 1988.

[UP-87] E. Upfal and D. Peleg: *The Generalized Packet Routing Problem*, IBM Research Report, RJ 5529 (56428), 1987.