SIMD Algorithms For 2-D Arrays In Shuffle Networks

by

Yosi Ben-Asher
David Egozi
Assaf Schuster

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #147
February, 1988

The work of the first author was done while under partial support of IBM under grant No. N00039-84-R-0605(Q).

The work of the second author was supported under National Science Foundation grant No. DCR-8413359 while employed at the Courant Institute.

Current Address for the first and third author's: Hebrew University, Department of Mathematics and Computer Science, Jerusalem, Israel.

* A version of this paper will appear in proceedings of 16th International Symposium on Computer Architecture, May 1989.

ABSTRACT

This paper studies a set of 2-D basic algorithms for SIMD perfect shuffle networks. Those algorithms where studied in [1] and [SN], 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 $N = L*P$, we improve some algorithms to be "2-D efficient", by improving their running time from $O(L*\log P)$ to $O(L + logP)$, as N exceeds P. We give nontrivial algorithms for the following operations: Row-Reduction, Parallel-Prefix, Transpose, Packing, Smoothing and Cartesian-Product.

1. Introduction and the Model

Schwartz [1] introduced the idea of a parallel processor based on the *shuffle* connections [2]; reviewed various basic of SIMD algorithms for such an ensemble of processors, and analyzed their asymptotic time complexities. However, 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 PEi.

Our algorithms correspond to a SIMD Perfect Shuffle connected machine (PS) [1]. For simplicity, although it is not crucial, we assume that P is a power of two. The processors are ordered left to right and are numbered $0 \cdots P-1$ accordingly, so we denote the j'th processor PEj. Subgroups of processors are referred to in a "natural" manner: left half, right half, odd, even etc. There are three kinds of communication steps: shuffle (σ), unshuffle (σ^{-1}) and exchange (EX) and a processor internal (fixed-size) computation. All of these operations take one time step.

In any parallel network machine we have two possible working modes:
One in which each processor computes a local result of a sequential computation using its local data. Then an inter-processor global algorithm is carried out to compute the final result. In other words, this mode involves separate sequential work by PE's followed by a global parallel step. It is associated with column order of the array: first each column is processed, only then a "row" of partial results is being processed.

In the other mode each processor chooses an element from its local memory. Then an inter-processor algorithm computes temporal results that are distributed back among the processors. This process repeats until all elements in the local memory of every processor have been processed. This mode corresponds to interleaved processing phases. It is associated with row order of the array: in each phase the next row of the array is being processed.

We attempt to develop a set of *2-D basic* and *2-D efficient* algorithms. In 2-D Basic we mean algorithms which enable us to manipulate the elements of the 2-D array, such that it will fit our working modes. For example, this include all sort of row-column relations. These algorithms, in turn, can be used as building blocks of more sophisticated algorithms for handling 2-D arrays on shuffle networks.

We now describe what 2-D efficiency means:

Assume a 1-D parallel algorithm $(N = P)$ which requires $O(\log N)$ steps, e.g. the parallel-prefix (PP) algorithm [6]. The naive way of expanding any such algorithm to 2-D $(N = L*P)$ case, is to simulate an N processor network by the P processor network that we have. Since in each processor we have L elements, such a simulation needs $O(L*\log N) = O(L*\log L + L*\log P)$ steps. However in [3] we find a simple method of performing 2-D PP in $O(L*\log P)$ steps. This is achieved by performing 1-D PP for each row, starting with the sum of the previous rows. Thus in the 2-D case we can expect to do better then the simulation of the larger sized 1-D case. We call an algorithm 2-D efficient, if it performs better than the naive simulation of the best-known same-size 1-D algorithm. We attempt to replace the '$*\log P$' term with a '$+\log P$' term whenever this is possible, getting $O(L + \log P)$ overall runtime.

For example consider the PP by columns: First each processor computes the PP of its column elements, then apply 1-D PP to the sums of each column. Now each processor has the sum of the elements in the previous columns. Finly each processor adds this sum to its elements and achieves the PP by columns in $O(L + \log P)$.

The following principles help to achieve 2-D efficient algorithms:

1. Locality

   As demonstrated by the above PP by columns algorithm, sequential calculations at the local environment of a processor are optimal, i.e. they have $O(N/P)$ speedup.

2. Optimal use of processors

   Using a 1-D algorithm which calculates results of some binary operation on the 1-D array, the number of inactive processors double every cycle. In 2-D algorithms these processors can be used to process the next row elements.

3. Limited distribution

   When distributing data elements to N $(N > P)$ virtual processors, it is sufficient to distribute them among the P available processors only.

4. Virtual simulation

   Moving a bunch of elements from one place to another during the run of an algorithm is not always necessary, as other elements move back later. This waste can be eliminated when simulating the run first, moving the number of elements in the bunch rather than the bunch itself.

2. Sort

In the recent years there has been an enormous amount of research on parallel sorting. For an overview of results see [10] and [11]. For PS networks, two algorithms are relevant to this work:

For $L \times P$ 2-D array, if $L < P$ it is best to use the $O(\log^2 P)$ bitonic sort [18], and use the merge and split of Baudet & Stevenson [12] to extend it to the 2-D case. This sorting terminates after $O(L*\log^2 P + L*\log L)$ steps. Another possibility is to use the radix-exchange sort [20] which consists of $2\log P$ packing operations. Using the 2-D packing described in section 5, this algorithm is a

$8*L*\log^2 P + o(L*\log P)$ steps 2-D sort. When $L \geq P$ the algorithm of Cypher and Sanz [9] which optimally sorts $L*P$ elements in $O(L*\log P)$ steps is better. Since there are two cases, the value of L must be available. When L is not known, it is easily found in $O(L+\log P)$ steps, using *maximum* operation.

3. The Raw-Reduction Operation

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

The naive way to achieve 2-D reduction is to use PP on every row, which requires $O(L*\log P)$ steps. In order to get a 2-D efficient algorithm we apply the the "optimal use of processors" principle (described in the introduction).

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, $i = 1 \cdots \log P$, processors $0 \cdots 2^{\log P - i} - 1$ are inactive.
- At the i'th round, $L \leq i \leq L + logP$, processors $P - 2^{i-L} + 1 \cdots P - 1$ are inactive.

The i'th round itself consists of four steps:

(1) Every even processor sends the step (2) result to its σ^{-1} neighbor.

(2) Every processor which got new value at step (3), 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, PEj, PEj moves the value of the i'th row to PE$j-1$ using the *EX* connection. Note that for every individual processor the round number (i) is the number of rounds it is active.

(4) For every even processor compute *bop* for its i'th row value, and the value it received at step (1).

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 PE0. Let K_l (K_r) denote the left (right) half of this set. At each round steps (3) and (4), 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 added to 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 of which an element is currently processed. The table is for sixteen processors, $L = 5$. Note the results accumulating in PE0, starting with the first row at step four.

Steps	PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PE8	PE9	PE10	PE11	PE12	PE13	PE14	PE15
step 8	5	5														
step 7	4	4	5	5												
step 6	3	3	4	4	5	5	5	5								
step 5	2	2	3	3	4	4	4	4	5	5	5	5	5	5	5	5
step 4	1	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
step 3			1	1	2	2	2	2	3	3	3	3	3	3	3	3
step 2					1	1	1	1	2	2	2	2	2	2	2	2
step 1									1	1	1	1	1	1	1	1

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

The reduction operation require $4(L + \log P)$ steps. The reverse sequence of steps can be used for broadcasting a column stored at PE0 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 actually traversing the tree topdown. We name *rop* some function $rop(x,y) \longrightarrow \langle l,r \rangle$ of a local value x and an inherited value y to create two different values l,r, for the left and the right sons.

3.1. 2-D Multiple-Broadcast operation

Let each processor have a column of L elements in a 2-D array. The multiple-broadcast operation is defined as the distribution all columns to all processors. Using the reverse 2-D reduction we get a 2-D multiple broadcast algorithm in $O(L*P)$ steps:

- In $L*(P-1)$ steps all elements arrive at PE0.
- Using reverse 2-D reduction PE0 broadcasts all $L*P$ elements to the rest of the world in $4(L*P + \log P)$ steps.

4. Parallel Prefix by Raws

In the introduction it is shown that PP by columns take $O(L + \log P)$ steps where the best known algorithm for PP by rows requires $O(L*\log P)$ steps [6]. Here this gap is closed by an PP by rows algorithm which requires $O(L + \log P)$ steps only.

The algorithm uses a variant of the 2-D reduction (see section 3). 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 PE0. 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 PEn, $\cdots$, PEm then its left son contains the corresponding sum of PEn, $\cdots$, PE($(n+m-1)/2$) and the right son contains that sum for PE($(n+m+1)/2$), $\cdots$, PEm.

The algorithm is based on the following 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.

(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 step (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 PE0.

(3) PE0 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.

5. 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 $1\ ,\ \cdots\ ,\ L*P$. Row order means that the numbering of elements increases first by columns and then by rows, i.e. PEi contains the following elements $\{\ i\ +\ p*j\ +\ 1\}_{j=0\ \ldots\ L-1}$. In column order PE$i$ contains $\{i*L+1\ ,\ \cdots\ ,\ i*L+L\}$.

Note that since our data type (2-D array) reflects the hardware structure, the result of the transpose operation is not the matrix transpose operation. Repeating the same transpose twice does not yield the original array, so $T_{rc}\ \neq\ T_{cr}$. An example is shown in fig. 2.

In [1] there is a brief description of the transpose operation for $P^{1/2} \times P^{1/2}$. We present a general algorithm for the 2-D case in O($L*\log P$) steps. To see that this is asymptotically optimal refer to section 6.4, the packing lower bound described there holds here too. When $L \geq P$ we could use the sorting operation as described before and get the same order of time for the transpose operation. However, our algorithm holds for any L and is of course, far better in terms of constants.

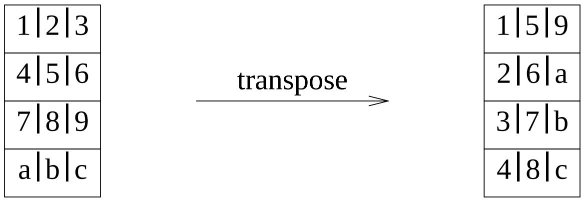

Fig. 2. 2-D T_{rc} operation for 4x3 array.

5.1. Transpose Rows to Columns T_{rc}

We denote the location of an element by $[i, j]$ $i = 0, \cdots, L-1$, $j = 0,...,P-1$, where j is the processor location and i is its local memory location. The new location is $[i', j']$ where :

$$i' = i*P + j - (j' - 1)*L \tag{1}$$

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 $\log P$ bits of their i index. Thus all the elements in a group have the same destination. The groups are ordered by their destination bits starting with j (the current processor): j , $j+1(mod\ P)$, $\cdots$, $j+P-1(mod\ P)$. Figure 3 describes this ordering for two processors, $J=000$ and $J=101$ (out of eight).

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 *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 does not change.

The algorithm advances in phases. In a single phase every processor send 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 for all of these elements we 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

group	PE=000	PE=101
1	000----	101----
2	001----	100----
3	010----	111----
4	011----	110----
5	100----	001----
6	101----	000----
7	110----	011----
8	111----	010----

Fig. 3. Group Ordering for Two Processors.

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.

We now analyse the algorithm total run time. First we count the total number of *EX* changes required for all elements residing at some processor PEj (from the above discussion this number is the same for all processors). There are $P - 1$ possible destinations, each determined by $\log P$ bits. For simplicity we add PEj itself as a destination. At average for each destination half the address bits are different from address bits of PEj. Hence the total number of *EX* steps is $L/P * P * \log P/2$. The above average also implies that half of the σ^{-1} steps are followed by an *EX* step, so finally we get a total of $(L * \log P/2) * 3$ steps for the completion of the algorithm.

5.2. Transpose Columns to Rows T_{cr}

The 2-D transpose from columns to rows can be used to scatter local results of every processor to all other processors. In this context the transpose operation serves as 2-D basic algorithm.

We use the same method as for T_{rc} with the following differences:

$$j' = \frac{j * L + i}{P} - i'$$

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 $<j, i>$.

6. Packing and Unpacking

6.1. One-Dimensional Packing

Consider data elements dispersed along a row of processors. The elements appear in order with the processors ids, at most one per processor. The 1-D packing operation moves the elements to consecutive processors, starting at PE0, preserving the original elements order. Schwartz [1] and Nassimi & Sahni [7] describe in general an algorithm for packing in Shuffle networks. We describe here in details a similar algorithm to perform packing in $4\log P$ steps, half of them for an initializing PP operation.

In order to perform packing of elements, the destination node of each element must be determined first. This is done in $2\log P$ steps by the parallel prefix algorithm. Then the algorithm proceeds with $\log P$ iterations. At each 2-steps iteration each element advances "one bit" along the shortest path to his destination, given that all elements correct address bit in the same manner (left to right or right to left).

We say that a *collision* occurs if after some step a processor contains more than a single element. Clearly, this can happen only as a result of an *EX* step, assuming no collision at the previous step. Thus when no collision occur, the algorithm run time is $4\log P$.

Denote by $dest(A)$ - the destination processor of an element A, $orig(A)$ - the processor where A originated, $loc_i(A)$ - the location of A at the end of the i'th iteration. We say two elements are adjacent when they are located in two (adjacent) processors, having an *EX* connecting them.

Consider two adjacent elements, A and B, just prior to the *EX* move of the $i+1$'th iteration. Note that $loc_i(A)$ is different from $loc_i(B)$ at the rightmost bit only. In order to get a contradiction, suppose at the *EX* move of the $i+1$'th iteration A stays in place but B joins A's processor.

- $loc_i(A)$ is identical to $loc_i(B)$ in $\log P - i - 1$ bits, starting at the second rightmost bit. These are the $\log P - i - 1$ leftmost bits of $orig(A)$ and $orig(B)$. Thus $|orig(A) - orig(B)| < 2^{i+1}$, so at most $2^{i+1} - 2$ elements could have resided between A and B initially. This implies $|dest(A) - dest(B)| < 2^{i+1}$.
- On the other hand, $loc_i(A)$ is also identical to $loc_i(B)$ in the i leftmost bits. If A is adjacent to B at the beginning of the $i+1$ iteration and B takes *EX* move but A does not, then clearly $loc_{i+1}(A)$ is identical to $loc_{i+1}(B)$ in the $i+1$ rightmost bits. These are also the $i+1$ rightmost bits of $dest(A)$ and $dest(B)$, so $|dest(A) - dest(B)| \geq 2^{i+1}$ which is a contradiction.

Note that, using same number of steps, the procedure can pack the elements starting from any processor, given that the order is preserved. The correctness proof still holds.

The unpacking is the inverse operation of packing, in which each element C has to arrive to his destination node $d(C)$. d is a permutation which preserves order, that is $A < B \implies d(A) < d(B)$. Unpacking is achieved in $2\log P$ steps using reverse paths of the corresponding packing. Note that whenever σ^{-1} is used in the packing algorithm, the unpacking algorithm uses the σ connections and vice versa.

6.2. 2-D Packing

Suppose some of the entries within the 2-D array are empty. The 2-D packing operation is defined similar to the 1-D packing described above. The 2-D array arrangement is in row order at any time (see also section 5). We know of no correct algorithm for solving this problem.

Here an $O(L * \log P)$ 2-D packing algorithm is described. The algorithm is confronted with an $O(L * \log P)$ lower bound for the operation, hence no 2-D efficient packing algorithm exists.

The 2-D packing algorithm:

(1) Perform 2-D PP in row order to get the destinations.

(2) Compute the column destination and the row destination of each element.

(3) Using 2-D PP, compare the destination row of each non null element with the destination row of the previous one in that row. Whenever the row destinations differ, assign this element with $ch = ch.previous + 1$, otherwise $ch = ch.previous$. The first non null element in every row always gets $ch = 1$ (see fig 6). PE($P - 1$) also stores a column of the highest *ch* (0, 1 or 2) of every row.

(4) Perform 2-D column broadcast of the "highest *ch*" column of PE($P - 1$). (Note that symmetry of the network allows us to refer to PE($P - 1$) as PE0 and vice versa).

(5) Denote by ch_i the highest *ch* of the *i*'th row. We perform *L* rounds, one for each row. At the *i*'th round we perform ch_i 1-D packings, where ch_i is at most 2. Note that the PP phase of the 1-D packing is superfluous and thus may be omitted. An element of the *i*'th row participates in the *j*'th pack of the *i*'th round if and only if its $ch = j$. This takes at most $4 * L * \log P$ steps.

The total run time of the algorithm is $4 * L * \log P + o(L * \log P)$ steps.

Steps	PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7
row destination	2	2	2	2	2	2		3
changes check	1	1	1	1	1	1		2
row destination		1	1		2	2		2
changes check		1	1		2	2		2
row destination	1	1	1			1	1	1
changes check	1	1	1			1	1	1

Fig. 4. 2-D Packing: An Example of Step (3).

6.3. 2-D Unpacking

The 2-D unpacking is the inverse operation of 2-D packing. Each element $C = <col, row>$ is destined to $d(C)$, where d is a permutation which preserves the 2-D row order, i.e. $A \le B \implies d(A) \le d(B)$.

An $O(L * \log P)$ 2-D unpacking algorithm consists of the following steps:

(1) & (2)

These steps are similar to steps (3) & (4) of the 2-D packing algorithm, correspondingly.

(3) This step is similar to step (5) of the 2-D packing algorithm, except replace 1-D packing with 1-D unpacking. Note that the highest *ch* here can exceed 2.

We claim that the algorithm run time is $O(L * \log P)$, same as packing. For row i $i = 1 \cdots L$: let A_i denote the number of different destination rows, and X_i the number of destination rows that do not appear in any previous source row. All X_i are mutually exclusive. Since there are no more than L destination rows, $\sum_{i=1}^{L} X_i \le L$. Since each row can have at most one destination row common to the previous row, $A_i \le X_i + 1$. Summing A_i's over all rows:

$$\sum_{i=1}^{L} A_i \le \sum_{i=1}^{L} (X_i + 1) \le 2L$$

Thus, step (3) takes at most $2L$ iterations of 1-D unpacking, so the algorithm total runtime is $4 * L * \log P + o(L * \log P)$.

6.4. 2-D packing lower bound

We call two processors *far* when at least $\log P - 3$ edges traversals are necessary to reach one from the other. It is an easy exercise to show that any arbitrary processor has more than $P/4$ of the other processors far from it. It is also not hard to construct a 2-D packing permutation where every processor sends an element to all other processors, e.g. the 2-D array where an entry is empty if and only if it is a diagonal entry. In such a case a total of at least $P * P/4 * \log P$ edge traversals are necessary for the completion of the packing operation. Hence the operation terminates after at least $P/4 * \log P$ steps, since only P such edge traversals are possible at a single step. Thus the lower bound follows for certain values of L.

7. The Smoothing Operation

This operation moves elements among columns, to make the height of each column the same. It can serve as a load-balance operation. Denote by $L_{\max}$ 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*). Note that $D_{\max} \le L$.

Smoothing is a special case of packing, so it can be completed in $O(L * \log P)$. However, since the preservation of order is not necessary, we hope to do better. We first present a rather streightforward algorithm.

7.1. 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. This is completed in at most $L_{\max}$ steps.

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 $\sim 2*L + 4*log^2 P$ 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 $\sim L*\log P + 4*\log^2 P$ 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.

7.2. 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 7.3). 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. The general question wether a 2D-efficient smoothing algorithm exists, remains open.

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 hight 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 round, 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

$$\textit{the total number of elements residing at odd processors.bris equal that of even processors.} \tag{2}$$

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 be 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 $0, 2^m, \ldots, P-2^m$. The phase advances in rounds: at the end of the i'th round $H_{i(2)} = H_{imod2}$ is evenly distributed among the elements of S_i. In other words, if $L_{l,i(2)}$ is current number of elements in PEl belonging to $H_{i(2)}$, then for all $k = 0, 2^{i'}, \ldots, P-2^{i'}$

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

Clearly after $\sim \log P$ 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 $k = 0, 2^{i'}, \ldots, P-2^{i'}$

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

and

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

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.

Figure 5 describes the first 2 rounds of this process. "*" denotes all processors, "1*0" denotes all processors starting with "1" and ending with "0", i.e. all even processors at the left side. The "US" (σ^{-1}) edges demonstrate the support operations between H_1 and H_2. At the beginning of round $i+1$ this property is used so $H_{i(2)}$ "supports" $H_{i+1(2)}$ such that (3) holds, 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 hight i' of a complete tree of hight $\log P$. 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

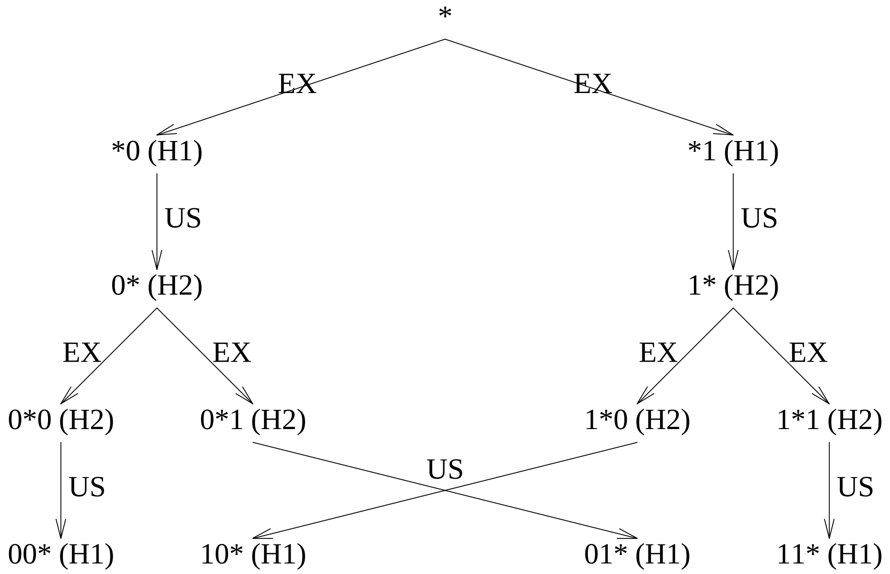

Fig. 5: the Support algorithm

moving elements between its children and according to their partial sums. This procedure of "control" takes $\sim 6i'$ 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.

7.3. Simulation Results

We know the control procedure takes $O(log^2 P * \log L_{\max})$ steps. The question remained 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 $\sim P/2$ 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*, achieveable by our Sun-3 workstation, is limited to 2^{17}. However, this range turned out to be sufficient in getting a clear tendency of the results. Even if this tendency does not hold for all *P*, it does , probably, for up to $P = 2^{25}$ at least. We tried 2 sizes of L: 1024 and 4096. To normalize, the total number of steps was devided by L. For each value of $\log P = 8, \ldots, 17$ the average of 10 experiments is presented. Most deviations from the averages presented were small: the highest of them, 0.55, was 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. The simulation results are presented in figure 6, where T is the number of steps divided by L the coulmn hights. One can see the difference between the simple-algorithm and the help algorithm.

7.4. 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 $D_i = \sum_{e \text{ in } PEi} e.c$ and L_i the number of elements in PEi before duplication.

For the 1-D case Schwartz [1] 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.

log(P)@T-help@T-help@T-simple			
	L=4096	L=1024	L=1024
8	1.23	1.29	2.57
9	1.30	1.36	2.99
10	1.54	1.38	3.18
11	1.59	1.54	3.31
12	1.63	1.61	3.47
13	1.86	1.76	3.63
14	1.96	1.85	3.77
15	1.94	1.90	3.89
16	2.10	1.96	3.91
17	2.12	2	4.37

Fig. 6: Smoothing simulation results.

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 with a tag $e.c$ 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 $O(\mathrm{Smooth(L)} + D/P)$, where L is the size of the highest column *before* duplication and Smooth(L) is the cost of smooth (see section 7.5). When $D/P > L$, this algorithm is 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.

7.5. 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 simpler smoothing algorithm (section 7.1). 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.
- For $D_i > D_{i+1}$, 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*-hights 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.

8. 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 of the final distribution of pairs.

In [1] there is an algorithm for 1-D CP for groups of size at most $P^{1/2}$. 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.

8.1. Cartesian Product by Groups

Given list of CPs CPBG = { $A_1 \times B_1, \cdots, A_r \times B_r$ }, 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. $r << P$ and $r << L$.
- 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.

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

(i) All groups at the "*B*-side" are broadcasted to all PEs, not repeating 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: $e.A_j.<1 \cdots |L_j|>$. 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 7.5). Whenever the smooth operation requires a spliting of $e.A_j.<x \cdots y>$ into e_1 and e_2, the range of e is split into disjoint "segments": $e_1.A_j.<x \cdots z>$ and $e_2.A_j.<z+1 \cdots y>$. The result of this step is an even distribution of the "pairing" work among all processors.

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

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

9. Discussion and Summary

The following table summarizes performance of 2-D efficient algorithms presented in this work. Only dominating terms are given.

Operation	Perfect Shuffle Time
Raw-Reduction@$4(L + \log P)$	
Multiple-Broadcast	5*L*P
Parallel-Prefix@$20(L + \log P)$	
Transpose Rows to Columns@$1.5 * L * logP$	
1-D Packing@$4 * \log P$ ($2 * \log P$ for Unpacking)	
2-D Packing and Unpacking@$4 * L * \log P$	
Smoothing@$L * \log P$ (no matching lowerbound)	
Dup@$O(L * \log P + D/P)$	
Cartesian Product of A and B@$\frac{\lvert A \rvert * \lvert B \rvert}{P} + 5 * \lvert B \rvert$	
Cartesian Product by Groups@$L * \log P + \sum (\lvert B_i \rvert + \frac{\lvert B_i \rvert * \lvert A_i \rvert}{P})$	

We gave several 2-D basic algorithms, several of them where 2-D efficient. In some cases it was not possible for us to do better then $O(L * \log P)$. It turned out that the 2-D efficient algorithms, such as row reduction or parallel prefix, are those related to problems which do not involve 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. Note that smoothing is not a spesific 2-D permutation but rather can be considered as a collection of such, from which one has to be chosen. The question remained open wether smoothing can be done 2-D efficiently.

An interesting model to compare with is the Layered Shuffle. This model incorporates $\log P$ layers of P swithches, each connected in a shuffle pattern to the next level. The last and first levels are connected to the PS network, so the leveled structure is cyclic. Switches are simple: a "limited" routing capability and one buffer of size $O(\log P)$ bits. On such a machine the cases of 2-D permutations considered, packing, unpacking, T_{rc} and T_{cr}, become 2-D efficient. $O(L + \log P)$ algorithms follow from the algorithms presented above. Smoothing is efficient as a special case of packing.

The set of algorithms that we have chosen to focus on, are such that efficiently support data manipulation on a SIMD machine. Such algorithms can be used as a basic operations on such a machine. The fact that several of these algorithms are provably optimal, can motivate two other lines of research: one is to implement a SIMD machine with these algorithms as instructions of some programming language. The second is to expand the MIMD PRAM machine to support SIMD mode, and to switch between these modes.

10. Acknowledgements

The authors wish to thank Marc Snir for his help, Yue-Sheng Liu for helpful discussions.

11. References

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

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

[3] C.P. Kruskal, L. Rudolph and M. Snir, *Techniques for Parallel Manipulation of Sparse Matrices*, to appear in ________________

[4] Allan Gottlieb, *An Overview of the NYU Ultracomputer Project*, Ultracomputer Note #100, N.Y.U., 1986.

[5] Hockney and Jesshope, *Parallel Computers*, Adam Hilger Ltd. Bristol, 1980.

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

[7] Ginosar R. and Egozi D., *Topological Comparison of Prefect Shuffle and Hypercube*, to appear in Int. J. of Parallel Programming.

[8] Nassimi D. and Sahni S., *Data Broadcasting in SIMD Computers*, IEEE Trans. on Computers, vol. c-30, no. 2, pp. 101-107, Feb. 1981.

[9] Cypher R. and J.L.C. Sanz, *Cubesort: An Optimal Sorting Algorithm for Feasible Parallel Computers*

[10] S. Akl, "Parallel Sorting Algorithms", Academic Press, N.Y. ,1985.

[11] D. Bitton, D.J DeWitt, D.K.Hasiao, J. Menon, *A Taxonomy Of Parallel Sorting*, Computing Surveys, Vol. 16, no. 3, Sep. 1984, pp. 287-318.

[12] G. Baudet, D.Stevenson, *Optimal Sorting Algorithm for Parallel Computers*, IEEE Transaction on Comp. vol. c-27, no. 1 Jan. 1987, pp. 84-87.

[13] Nassimi D. and Sahni S., *Parallel Permutation and Sorting Algorithms and a New Generalized-Connection-Network*, Univ. of Minnesota, Mineapolis, Tech. Rep. 79-18, 1979.

[14] Ginosar R. and D. Egozi, *Algorithmic Comparison of Hypercube and Perfect-Shuffle-Nearest-Neighbor Networks*, to appear in Int. J. of Parallel Programming.

[15] F.T. Leighton, *Complexity Issues in VLSI*, MIT Press, Cambridge, MA, 1983.

[16] C.P. Kruskal and M. Snir, *A Unified Theory of Interconnection Network Structure*, Ultracomputer note #106, 1986.

[17] D. Mitra, *Randomized Parallel Communication on an Extension of the Omega Network*, J. of the ACM, October 1987.

[18] K. E. Batcher, *Sorting Networks and their Applications*, Proc. 1968 Spring JCC, AFIPS Press, pp. 307-314.

[19] G. Bleloch, *Scans as Primitive Parallel Operations*, Proceedings of ICPP 1987, pp. 355-362.

[20] D. E. Knuth, *Sorting and Searching*, Eddison-Wesley, Reading, Mass., 1973.

Table of Contents