NBSIR 84-2936

Diamonds and Diamond Sorting

Eleazer Bromberg and Francis Sullivan

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

September 1984

U.S. DEPARTMENT OF COMMERCE

NATIONAL BUREAU OF STANDARDS

NBSIR 84-2936

DIAMONDS AND DIAMOND SORTING

Eleazer Bromberg and Francis Sullivan

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

September 1984

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

DIAMONDS AND DIAMOND SORTING

Eleazer Bromberg and Francis Sullivan

National Bureau of Standards

I. INTRODUCTION

The Diamond sort algorithm was introduced in [1]. Like the Batcher sort [2], which it resembles in many ways, it is designed for parallel operations, which makes it well suited to vector-computer architecture. The instruction sequences are fixed, without any branches. The algorithm therefore has a fixed complexity determined solely by the number of elements to be sorted, rather than by any pattern of distribution of element values. Some timing results appeared in a report by Mossberg [3].

The Diamond sort is of special interest not only because of the unusual arrangement of its steps, but also because it introduces the concept of a Diamond as a set with a specific kind of partial ordering that is closely linked to the halving principle, which is used in the sort procedure. Unfortunately, this unusual arrangement makes it difficult to understand exactly why the algorithm produces a sorted list. The program in [1] is correct, but a more detailed description is needed to clarify how it works.

The present paper describes the Diamond sort algorithm and analyzes it in a way that differs significantly from [1]. It emphasizes a terminology of block arrays; that is, arrays with elements that are themselves subarrays of uniform length. This is particularly effective when the number of elements in an array is some power of 2, say 2(exp K), because the array may then be treated as any one of K+1 possible block arrays, by selecting a block length equal to 2(exp k), where k is an integer between 0 and K. A definition of inequality between arrays is introduced which makes it possible to carry over many of the algorithms for scalar arrays to block arrays and thus develop flexible vectorial tools. These make vector sorting procedures easier to compose and to manipulate.

Briefly stated, a diamond is a set in the form of an array with the following properties: the number of elements is some power of 2 and, for any partition of the array into a set of successive subarrays of the same length, the lower half of each subarray is, element by element, less than its upper half. Diamonds are defined and described in greater detail in Section II.

The idea of an inequality relation between two sets is introduced in Section II, and is elaborated in Section III. This is then used in Section III to describe how an arbitrary array with N elements, N = 2**K, is transformed into a diamond by K vector compare-exchanges, interlaced with rearrangements of elements so that successive compare-exchanges deal with different pairings of elements. These pairings correspond to a succession of partitions with steadily decreasing block lengths.

The definition of a diamond requires only that it be partially ordered. Full ordering is achieved by a succession of merge procedures. In keeping with the goal of parallelism in the sort algorithm, the merge procedure is designed so that it can be applied synchronously to any number of pairs of ordered arrays. It uses vector compare-exchange instructions in a manner that is implicit in the Batcher algorithm. The method is described and its validity is proved in Section IV.

It is shown in Section V that if the block array corresponding to a particular partition of a diamond is fully ordered, then the next finer partition can be treated as a pair of ordered block arrays which can be combined by the merge procedure to yield a fully ordered (refined) partition. Starting with the partition of the full diamond into two blocks, a succession of (K-1) merges leads to the full ordering of the simple-element array. The algorithm for this procedure is also presented in Section V.

The full Diamond sort algorithm is presented in Section VI. This combines the two major phases: first, transforming the given array into a diamond, and then, transforming the resulting partially ordered array into a fully ordered one.

It is assumed for convenience in the following that all elements are distinct. This restriction can readily be removed with only minor modifications. The algorithms, however, are generally valid.

II. THE DIAMOND STRUCTURE

A diamond is a dataset characterized by certain relations among its elements. Two definitions are presented: one is recursive and refers to inequality relations between subarrays; the other is index-related, with inequalities prescribed for pairs of elements with certain differences in the bit representations of their indices.

A. Recursive definition.

A diamond is any set D with the following properties:

a) D is a single element;

OR

b) D is "structurally divisible" into two disjoint subsets D0 and D1 such that
D0 and D1 are both diamonds, and
D0 < D1.

The relationship D0 < D1 is taken to mean that D0 and D1 have the same number of elements, say h, and that each element of D0 is less than or equal to the element of D1 that has the corresponding location or index. Thus, D0(i) =< D1(i) for any i in (0:h-1), or equivalently, D(i) =< D(i+h). These relations may be referred to as structural inequalities since they follow the structure of the set D; that is, the relations between values of elements of D are known for certain pairs of locations. Correspondingly, the expression "structurally divisible" means that the assignment of elements from D to the two subsets is based on the locations of the elements in D, without specific reference to their scalar values.

The set D may be a single element, or an array or subarray with an even number of elements, in which case its two subsets are its lower and upper halves. Since D0 and D1 are required to be diamonds, the above properties apply as well to their subdivisions, and therefore the number of elements in any diamond must be some power of 2.

It is useful to introduce an unfolded version of this definition:

A diamond is a partially ordered array such that

EITHER
   a) it consists of a single element;
OR
   b) the number of its elements is some positive power of 2,
      AND
      for any partition of the array into a sequence of subarrays of equal length, numbered from zero on up, every element in any even-numbered subarray is less than or equal to the correspondingly located element in the next (hence odd-numbered) subarray.

It can be seen that the two definitions are equivalent by taking D to be any block element in any partition. D may therefore be any subarray in a partition, ranging from a single element in the finest partition to the full array. In the first case, it satisfies a) in either definition. Otherwise, the length of D is some power of 2 and D can be divided in half, in which case its lower half is an even-numbered subarray in the next finer partition and its upper half is the succeeding odd-numbered subarray; then if D satisfies b) by one definition, it must satisfy b) in the other.

The above relations are illustrated in the following Table 1 for the three coarsest partitions of a diamond d of N elements (N = 2**K). Each partition is assigned a level number j, such that the number of subarrays in that partition equals 2**(j+1). The length of each subarray in a partition is called hj.

THREE PARTITIONS

Level j:	0		1		2
Length	h0 = N/2		h1 = N/4		h2 = N/8
		AND		AND	
					d(0*h2;h2) < d(1*h2;h2)
			d(0*h1;h1) < d(1*h1;h1)		
					d(2*h2;h2) < d(3*h2;h2)
	d(0*h0;h0) < d(1*h0;h0)				
					d(4*h2;h2) < d(5*h2;h2)
			d(2*h1;h1) < d(3*h1;h1)		
					d(6*h2;h2) < d(7*h2;h2)

Table 1 Inequalities at first three partition levels

Generally, at any level j,

d(2*k*hj;hj) < d((2*k+1)*hj;hj)
where hj = 2**(K-j-1)
and k lies in [0:2**j-1].

There are only K partition levels; at the highest level j = K-1, h = 1, and the blocks are simple elements.

A diamond is in general only partially ordered. Taking a set S of four elements, if the relations between the lower and upper halves of S are

S(0) < S(2) and S(1) < S(3)

and the relations in the subarrays S(0:1) and S(2:3) are

S(0) < S(1) and S(2) < S(3),

then S satisfies the definition of a diamond, but the relation of S(1) to S(2) is undetermined.

B. Index-related definition.

If the indices of the elements of a set, ranging from 0 through N-1, are expressed in binary notation, a diamond can be defined as follows: Any two elements with indices that differ in only a single bit are related by the inequality that the element with the smaller index is less than the element with the larger index.

It is easy to check that the two definitions are equivalent by matching the second definition against the unfolded version of the first. A diamond may also be represented geometrically as a K-dimensional hypercube, where each vertex is identified with an element. The edges are arrows (directed arcs) parallel to the axes of the K-dimensional space and similarly oriented (in the positive direction), and any element identified with the tail of an arrow is less than the element corresponding to its head. This is illustrated in the following:

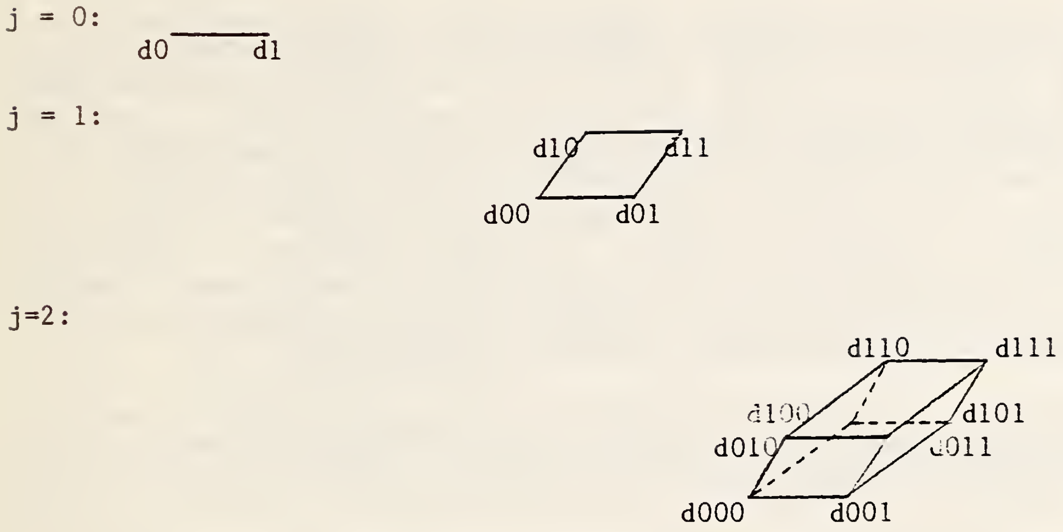


Figure 1. Cumulative block inequalities at successive partition levels

In this Figure, each node represents a block:

di = d(i*hj;hj),

where hj = N/(2**(j+1)) and the suffix i is written in its bit representation, so that the number of bits is j+1, corresponding to the partition level, and the value of i is the subarray number for that partition. Edges denote inequalities, with the node on the left of each edge less than the node on its right. Horizontal lines represent the inequalities that appear for the first time at the corresponding level, and their nodes are the lower and upper halves of single nodes at the preceding level. Oblique edges are used to denote the carryover of the structural inequalities of earlier levels. (The diamond-like shapes of these diagrams gave rise to the name of the method.)

The index-based definition and the geometrical representation are both of interest in studying symmetries in the structure of a diamond. Let the indices be transformed by an arbitrary permutation of their K bits, and rearrange the elements to conform to the corresponding new values of their indices. Clearly the inequalities continue to be related to the differences in indices and are not affected by this transformation. Hence the new arrangement of the elements continues to be a diamond, equivalent to its predecessor diamond. This is equivalent to a renumbering of the axes in the geometric representation, which will not change the directions of any arrows between vertices.

III. CONSTRUCTION OF A DIAMOND

Any set S of N elements, where N = 2**K, can be transformed into a diamond by a sequence of K steps which can be understood in terms of operations at successive partition levels from 0 through K-1. At each level, even-numbered blocks are assigned, in order, to one block array, x, and odd-numbered blocks are assigned to block array y. Matching elements in x and y are compared, and interchanged if necessary to establish the inequality x < y. This yields the relation formulated for the partition at that level. This procedure is effective because, as will be shown, inequalities established at any step remain valid through all subsequent steps. When all partitions have been processed, the resulting array is a diamond, since all of the inequalities characterizing a diamond are satisfied.

Two vector operations are used: One is the assignment of subarrays to x and y appropriate to the current partition level, and the other is a compare-exchange transformation.

A) The compare-exchange operation and its properties:

A compare-exchange takes input arrays, say a and b, each of length n, and transforms them into arrays A and B of the same length, with A < B. This applies the instruction

```
if(a(i) > b(i)) interchange their values
```

to the n pairs of elements. Since the instruction can be carried out independently for each pair, the compare-exchange transformation can be represented as a vector operator:

```
E(a,b,n) -> (A<B).
```

This operation is designated COMPAR(a,b,n) in the formal statement of the algorithm in the last Section.

It should be noted that

```
A(i) = min(a(i),b(i)), and
B(i) = max(a(i),b(i)).
```

Paired inequality relations existing in a and in b are retained under the compare-exchange operation; viz.,

```
Theorem III:  If a,b are arrays, each of length n,
               & a(i) < a(j) and b(i) < b(j) for some i and j
               & E(a,b,n) -> (A<B),
              then
                 A(i) < A(j)
               & B(i) < B(j).

Proof:   A(i) = min(a(i),b(i)) =< a(i) < a(j)
   Also,                       =< b(i) < b(j).
   Hence, A(i) < min(a(j),b(j)) = A(j).
```

A similar proof holds for $B(i) < B(j)$.

This theorem can immedately be generalized to sets of paired inequality relationships:

```
Corollary 1:  If I and J are sets of indices, each of length m,
               & a(I) < a(J) and b(I) < b(J),
              then
                 A(I) < A(J),
               & B(I) < B(J).
```

```
Proof:  For any k in (0:m-1), set i = I(k) and j = J(k), and
        apply the Theorem.
```

The above Corollary allows the Theorem to be extended to block arrays. If m divides n, sets a, b, A, and B may be represented as block arrays a0, b0, A0, and B0, each with n/m block elements. Setting

$Ii = (i*m:i*m + m - 1)$, with i in $(0:n/m-1)$,

define

$a0(i) = A(Ii)$,

with similar representations for the other block arrays. Then the following Corollary is a block version of the Theorem.

```
Corollary 2:  If a0 and b0 are block arrays
                 with k block elements each,
                 with each block element of length m,
             &   a0(i) < a0(j)  and b0(i) < b0(j)
                 for some i and j
             &   E(a0,b0,(k*m)) -> (A0<B0),
             then
                 A0(i) < A0(j) and B0(i) < B0(j).
```

It now follows simply that ascending arrays remain ascending under a compare-exchange:

```
Corollary 3:  If a and b are ascending arrays,
              then
                A and B are ascending arrays.
```

```
Proof:  Take any i,j in (0:n-1), with i < j, and
        apply the Theorem.
```

Similarly,

```
Corollary 4:  If a0 and b0 are block arrays,
                 each with k block elements,
                 and each block of length m,
```

& the blocks are ascending in both arrays,
then
  A0 and B0 are ascending block arrays.

Proof: Take any i,j in (0:k-1), with i < j, and apply Corollary 2.

Note that while any set of scalars must have at least one maximal and one minimal element, the same is not true of a set of blocks. If, however, a linear sequence of inequality relations has been established for a set of blocks, then that set may be said to have a maximal and a minimal (block) element. These distinctions are observed without explicit reference in the remarks and proofs that follow.

B) Procedure for constructing a diamond:

The procedure for creating a diamond out of a given array S with N elements (N = 2**K), consists of steps based on the partitions at levels 0 through K-1 in order. At each step, with all even-numbered subarrays assigned to x and all odd-numbered subarrays assigned to y, a compare-exchange will produce the inequality relations prescribed for that level in the unfolded definition, and retain all preceding structural inequalities. The sequence of K vector compare-exchanges as described thus leads to a diamond. It is necessary, however, to consider how the assignment of elements to x and y are to be carried out.

The first step (at level 0) is straightforward: assign the first n= N/2 elements of S to x and the last n to y. Subsequently, however, there must be transfers of some elements between x and y at the start of each level. Conceptually, the blocks of x and y are interleaved, x leading, after each compare-exchange, to reconstitute a transformed, partially ordered array S. Then S is repartitioned, with new subarrays of half the length of the preceding step, and a new assignment of even- and odd-numbered subarrays to a new x and y is carried out. It should be noted that the new even-numbered subarrays are the lower halves of the preceding larger subarrays, and the new odd-numbered subarrays are their upper halves.

This reconstitution need not actually be carried out, however. Since the repartitioning consists in simply dividing in half all of the subarrays of the "old" partition, this division can be imposed on the blocks of the old x and y. Then the old x and y may be considered to have twice as many blocks, corresponding to the blocks of the new partition, but not yet in proper place. To assign them properly, let the new blocks of x and y be given numbers corresponding to their position in x or in y, and interchange every even-numbered new block in y with the next higher odd-numbered new block in x.

It can be seen that this rearrangement yields the desired result, as it sends the lower half of all old blocks of x and y (the even-numbered new blocks of old x and y) into a new x and the upper halves of these old blocks into a new y. It is only necessary to interchange alternate new blocks because the lower halves of the old blocks of x should stay in x, and the upper halves of the old blocks of y should stay in y.

If h is taken to be the subarray length in the new partition, the operation transforming the old x and y into new X and Y is denoted by

Z(x,y,n,h) -> (X,Y),

and is labelled ALTBLK(x,y,n,h) in the code of Section VI.

After the step at level K-1, the output arrays X and Y consist of subarrays of length 1; hence of indivisible elements.. Following the conceptual model, the full set S should then be obtained by interlacing their elements. This step is unnecessary, however. It can be seen that the arrays x and y, corresponding to the even and odd indexed elements of S, are each diamonds, and, since x < y, the concatenation of x and y yields a full diamond which is equivalent to the conceptual model, and may be used in its place.

The procedure for transforming a given set S = (x,y) to a diamond can now be formulated succinctly:

```
n = N/2
x <- S(0;n)
y <- S(n;n)
(x,y) <- compare-exchange(x,y,n)
m = n
while m > 1 [m = m/2
             (x,y) <- alternate block exchange(x,y,n,m)
             (x,y) <- compare-exchange(x,y,n)
            ]
end while
S(0;n) = x
S(n;n) = y
```

IV. VECTOR MERGE OF A PAIR OF SORTED ARRAYS

The transformation of a diamond into a fully ordered array consists of a cascade of merge operations. As described in the next Section, this starts with the parallel merging of N/4 pairs of sorted 2-element subarrays, and continues with parallel mergers of smaller numbers of pairs of longer subarrays until it merges a single pair of N/2-element subarrays. In order to take advantage of the speed of a vector processor, the conventional merge algorithm used in serial mode can not be used because this would require a different number of branches for different pairs, thus destroying the synchronization of streamed data needed for parallel operations. It is necessary, instead, to use vector instructions that keep all pairs in lockstep.

The present Section presents a vector algorithm for the merger of a single pair x,y of sorted arrays with n elements each, and with x < y. The number of elemental compare instructions required is (n*k*(k-1)/2 +n-1), where n = 2**k. By contrast, the number of compare instructions in the serial algorithm lies between n-1 and 2n-3 (the positions of smallest x and largest y are known). However, the time required for the vector merge procedure may be substantially less than that for the serial mode, because of the parallelism of streamed comparison and exchange operations.

The vector merge algorithm is developed first for a pair of sorted arrays with simple elements. This is then extended to cover the merging of a pair of block arrays. The latter case has a special feature arising from the fact that when two blocks are compare-exchanged they will in general be transformed, since compare-exchange is fundamentally a simple-element process, and some pairs of matching elements in the pair of blocks will be exchanged, while others will stay put. As a result, the merger of two block arrays yields two arrays in which the blocks are in the desired order, but they may not be identical with the input blocks, although the simple elements composing any pair of matching input blocks will also compose the corresponding output blocks.

A. MERGING ARRAYS OF SIMPLE ELEMENTS

The two simple-element arrays to be merged, x and y, have the following properties:

a) x and y are both of length n (equal to some power of 2);

b) x and y are given as increasing arrays;

c) x < y.

The procedure consists solely of the repeated application of a vector operation called reverse-compare-exchange with offset m, as m varies from n/2 down to 1, being reduced by one-half at each step.

The final output arrays will be in alternating order with x leading; that is,

x(i) < y(i) < x(i+1) < y(i+1) for i in (0:n-2),

The procedure will therefore be called a shuttle-merge. The arrays will be said to be in shuttle order. The fully sorted array of 2n elements can be obtained from the output arrays by an interlace of their elements (a perfect shuffle).

The reverse-compare-exchange with offset m is a special version of the compare-exchange operation, from which the first m elements of x and the last m elements of y are excluded. The paired elements in the individual comparisons are x(i+m) and y(i), for all i in (0:n-m-1); but in a reversal of the order relation of x to y, the operation is used to assign the lesser value to y(i) and the greater to x(i+m). Thus, in the output,

$$y(i) < x(i+m), \quad \text{for all } i \text{ in } (0:n-m-1).$$

This could be represented in terms of the compare-exchange as

$$(x,y) \leftarrow E(y(0;n-m),x(m;n-m),n-m),$$

but instead it will be written generally as

$$(x,y) \leftarrow R(x,y,n,m).$$

This operation has the following properties under the above conditions a), b), and c):

b') The output arrays x and y continue to be in ascending order, regardless of the value of m, and

c') the inequality $x < y$ continues to hold despite the reverse assignments, if m is appropriately chosen.

These properties will now be proved.

First: It is shown that if the input arrays x and y are both ascending arrays, then the output arrays (which will be called xo and yo whenever there is any possiblity of confusion) are also ascending. The restriction that the elements of x and y be distinct is retained and again it can readily be eliminated with only minor modifications in the discussion and proof.

The elements of xo and yo are defined as:

$$\begin{aligned} xo(i) &= x(i) && \text{for } i \text{ in } (0:m-1) \\ &= \max(x(i),y(i-m)) && \text{for } i \text{ in } (m:n-1), \text{ and} \end{aligned}$$

$$\begin{aligned} yo(i) &= \min(x(m+1),y(i)) && \text{for } i \text{ in } (0:n-1-m) \\ &= y(i) && \text{for } i \text{ in } (n-m:n-1). \end{aligned}$$

a) xo(0:m-1) is ascending, since it is unchanged from x(0:m-1), which is given as ascending.

b) xo(m:n-1) is ascending, since for any i,j in (m:n-1) and $i < j$, x and y ascending implies that

$$\begin{aligned} x(i) &< x(j) =< \max(x(j),y(j-m)) = xo(j), \text{ and} \\ y(i-m) &< y(j-m) =< \max(x(j),y(j-m)) = xo(j); \end{aligned}$$

and therefore,

$$xo(i) = \max(x(i),y(i-m)) < xo(j).$$

c) To combine a) and b), consider that since $m-1 < m$, it follows that

$xo(m-1) = x(m-1) < x(m) =< \max(x(m),y(0)) = xo(m).$

Since xo(m-1) is the largest element of the ascending subarray xo(0:m-1), and xo(m) is the smallest element of the ascending subarray xo(m:n-1), the full array xo(0:n-1) must be ascending.

It can be shown similarly that yo is ascending.

Second: If m is chosen properly, the order relationship x < y on input holds for the output arrays; that is, that xo(i) < yo(i) for all i. The first part of the proof shows that x < y when m = n/2, and the second part extends the results to include the cases m < n/2.

a) For i in (0:m-1), with m =< n/2,
since the first m elements of x are unchanged,

```
  xo(i) = x(i) < y(i),    because x < y,
&         x(i) < x(i+m),  because x is ascending.
```

Therefore,
$xo(i) < \min(x(i+m),y(i)) = yo(i).$

b) For i in (n-m:n-1), with m =< n/2,
since the last m elements of y are unchanged,

$yo(i) = y(i);$

```
furthermore, since x(i) < y(i)
               and y(i-m) < y(i),
it follows that
```

$xo(i) = \max(x(i),y(i-m)) < y(i) = yo(i).$

```
Note:  At this point, if m is taken equal to n/2,
         a) shows that xo(i) < yo(i) for i in (0:n/2-1) and
         b) shows that xo(i) < yo(i) for i in (n/2:n-1).
       Therefore xo(i) < yo(i) for all i, and
                 xo < yo.
```

Some inequality relationships among subarrays of xo and yo should be noted: The reverse-compare-exchange with offset m = n/2 yields the relationship

$$yo(0;m) < xo(m;m),$$

since these are the subarrays that were compared. Furthermore, the following shuttle order holds since xo < yo:

$$xo(0;m) < yo(0;m) < xo(m;m) < yo(m;m);$$

that is, alternating subarrays of length m in xo and yo are in ascending order, with xo leading.

c) Having proved the inequality xo < yo for an offset m = n/2, the case of an offset m' < n/2 is now to be considered. The results of preceding cases a) and b) are applicable for m' as well as for m. On the other hand, the interval (m':n-1-m') is no longer empty, as it is for m = n/2, and it becomes necessary to consider the relationship of xo(i) and yo(i) when i lies in that interval. In order to prove that xo(i) < yo(i), that is,

```
    that  max(x(i),y(i-m')) < min(x(i+m'),y(i)),
```

it is necessary to show that four inequalities hold: each of the elements on the left must be less than either of the elements on the right. Since x and y are ascending, it follows that

```
      x(i)    < x(i+m')
  &   y(i-m') < y(i)
```

Furthermore, since x < y, it follows that

```
      x(i)    < y(i).
```

The fourth inequality that is required,

```
      y(i-m') < x(i+m')   for i in (m':n-m'-1),
```

is not automatically satisfied; it is, rather, an additional condition that must be imposed. It is useful to reformulate this condition by replacing (i-m') by i:

```
      y(i) < x(i+2*m')  for i in (0:n-2*m'-1).
```

Thus it follows that the output arrays xo and yo will be ascending and will satisfy xo < yo if the same is true of the input arrays and if, in addition,

```
      y(0;n-2*m') < x(2*m';n-2*m').
```

This can be restated in the form:

If two arrays satisfy the conditions a), b) and c), and in addition if there is some integer m such that

```
      d) y(0;n-m) < x(m;n-m)
```

then the output arrays xo, yo given by

```
      (xo,yo) <- R(x,y,n,m')
```

will both be ascending and xo < yo if m'=m/2.

Furthermore, it can readily be seen that the subarrays of length m' in xo and yo now lie in shuttle order, since

```
      xo(i) < yo(i) < xo(i+m'),
```

where the first inequality follows from xo < yo, and the second is the result of the reverse-compare-exchange. Thus,

$$xo(0;m')<yo(0;m')<xo(m';m')< \ldots <xo(n-m';m')<yo(n-m';m');$$

The sequence that leads to the merging of the two input arrays can now be formulated. It consists, as previously stated, of a succession of reverse-compare-exchanges in which the first offset m is n/2, and subsequent offsets are each one-half of the preceding value, so that the conditions are satisfied at each step for the output pairs to be ascending, with xo < yo. This sequence ends after the offset m = 1. At that point the subarrays are all of length 1 and the shuttle order may be stated as

$$x(0) < y(0) < x(1) < y(1) <\ldots< x(n-1) < y(n-1);$$

that is, the operations for sorting are complete, and a single sorted array can be obtained by interlacing the elements of x and y.

The shuttle-merge process can also be described in a different manner that may help to obtain further insight into its mechanics, and that show it to be, in a sense, a multiple binary search procedure. This alternative view is only sketched here, for the particular case of two 8-element arrays.

Because of the known character of the input arrays, the order rank of each element among all 16 elements is partially determined at the start, and the shuttle-merge provides the additional information required for final determination of the ranking. Since x and y are each sorted, the rank of each element with respect to the other 7 elements of its array is known at all times; the relation x < y provides additional information at the start. In the accompanying diagram, y(2) is taken as a typical element, with lesser values as determined by the input conditions marked with a minus (-) sign, and larger values marked with a plus (+). Since all elements are assumed to be unequal, this leaves 6 possible slots where y(2) may fit, all marked with a question (?) mark. (It should be noted that the two elements, y(0) and x(7), have the maximum number of possible slots, namely 8, at the start, and the two elements x(0) and y(7) have predetermined positions because they must be the minimum and maximum of the 16 elements.)

```
- - -? ? ? ? ? ?
x x x x x x x x
y y y y y y y y
- - © + + + + +
```

The next figure shows the effects of the sequence of reverse-compare-exchanges, with offsets of 4, 2, and 1.

```
Offset   Input                    Output

      - - -? ? ? ? ? ?            - - -? ? ? ?+ +
 4    x x x x x x x x             x x x x x x x x
              y y y y y y y y             y y y y y y y y
              - - © + + + + +             - - © + + + + +

      - - -? ? ? ?+ +             - - -? ?+ + + +
 2    x x x x x x x x             x x x x x x x x
          y y y y y y y y             y y y y y y y y
          - - © + + + + +             - - © + + + + +

      - - -? ?+ + + +             - - - + + + + +
 1    x x x x x x x x             x x x x x x x x
        y y y y y y y y             y y y y y y y y
        - - © + + + + +             - - © + + + + +
```

The first operation cut the maximum number of possible slots for any element to 4, here applicable to the value of y(2), possibly transferred from x(7) by the reverse-compare-exchange. The second operation cut the maximum number of possible slots to 2, and the last leaves only a single slot, corresponding to the final rank of y(2) in shuttle order, between x(2) and x(3).

The shuttle-merge procedure for merging two sorted arrays x and y, with x < y, can be outlined as follows:

```
m <- n
while m > 1,  [m <- m/2
               (x,y) <- R(x,y,n,m)
              ]
end
```

B. MERGING BLOCK ARRAYS

The logical basis for the foregoing discussion and proofs applies to block arrays just as well as to arrays of simple elements, on the understanding that the quantities n and m refer to the number of block elements in each array and the number of block elements to be used as offsets, respectively. This follows readily from Corollaries 2 and 4 in Section III. The merge procedure then yields a shuttle-ordering of the blocks in the output block arrays xo and yo. It should be noted, however, that the set of blocks in the combined output is not the same as the set of blocks in the combined input, as they are in general subject to transformation in the course of applying compare-exchanges.

It is desirable to change the notation in the references to n and m in E and R, the compare-exchange and the reverse-compare-exchange, however, so that the calls will refer to the total numbers of simple elements involved. Thus, if n is taken to be the number of simple elements in each of the x,y arrays, and if the length of each block is h, so that k = n/h and N = 2*n, then the general expression for a reverse-compare-exchange with (block) offset m of two arrays x and y, each with n simple elements is

```
R(x,y,n,j*h)    for j in [1:k/2],
```

and the procedure for a shuttle merge of a pair of block arrays with blocks of length h, written shuttle-merge(x,y,n,h) follows:

```
m <- N/2
while m > h, [m <- m/2
              (x,y) <- R(x,y,n,m)
             ]
```

It should be noted that, because the relations between blocks also represent relations between corresponding elements of the blocks, the merging of a pair of arrays with block elements of length h represents also the simultaneous merging of h distinct pairs of simple arrays. Furthermore, the effect of the merge procedure is such as to retain any order relationship between these h arrays that may exist between them at input.

V. SORTING A DIAMOND

In describing the process of sorting a diamond, the introduction of order relationships among sets makes it possible to treat sets as entities in chains of inequalities. Thus, for example, sets A, B, C and D, all of length n, may be said to be perfectly ordered when

$$A < B < C < D.$$

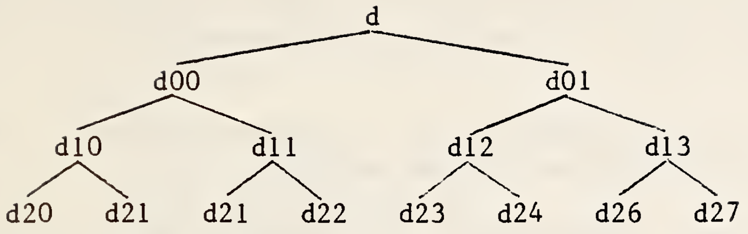
This would mean that

$$A(i) < B(i) < C(i) < D(i)$$

for all i in (0:n-1). This statement does not carry any general implications about the relative order of the simple elements in the individual sets. (Take for example, (1,2,3,4) < (9,8,7,6).)

The sorting of a diamond consists in traversing the partitions in succession, ordering the component subarrays at each level in turn, in such manner as to retain all the previous order relationships that were inherent in the diamond structure.

It is helpful in describing the process to present the relations between the partitions of a diamond as the following tree:



where each node represents a subarray which is itself a diamond; the nodes at each level represent the subarrays of the partition corresponding to that level; the two children of each node represent the lower and upper halves of that node; and the child to the left is less than the child to the right. In order to maintain the terminology introduced previously, the nodes d00 and d01 will be considered to lie at level 0, with the following level numbers defined accordingly. There are therefore K-1 levels below d.

The symbols dji are introduced for compactness in referring to the i-th diamond in the partition at level j, corresponding to the subarray

dji = S((i-1)*hj;hj),

where hj is, as before, the length of the individual subarrays at level j; namely, hj = 2**(K-j-1).

The two diamonds at level 0 form a (block-) ordered pair. Proceeding inductively, assume that the k = 2**(j+1) diamonds of length hj at level j are ordered, in the block sense. This means that

dj0 < dj1 < dj2 < ... < dj(k-1).

Because these are diamonds, the next level divides each dji into two diamonds,

dji = (dj'i' < dj'(i'+1)),

where j' = j+1, and i' = 2*i.

Applying the assignment pattern used earlier, the smaller of each pair of diamonds is assigned to array x' and the larger to array y':

x' = (dj'0 < dj'2 < dj'4 < ... < dj'(2*k-2)), and
y' = (dj'1 < dj'3 < dj'5 < ... < dj'(2*k-1)).

Considering x' and y' as block arrays, with dj'i' as the block elements, it can be seen that they are each ordered arrays, (ordered with respect to the blocks), with k block elements each, and with x' < y'. The array pair x' and y' satisfies the conditions for a shuttle-merge, and the output xo' and yo' of that procedure will have diamond blocks in shuttle order.

It is therefore possible, starting at level 0, with d00 < d01, to proceed sequentially through successive levels to level K-1, to obtain a pair of arrays that are in shuttle order with blocks of length 1; that is, with the simple elements in shuttle order. A perfect shuffle then yields the fully ordered array.

As in Section III, rearrangements in the transition from one level to the next are carried out by an alternate block exchange.

The procedure can now be outlined for the transformation of a given diamond d, divided into two arrays x and y, corresponding to d00 and d01 respectively, into a fully sorted array:

```
n <- N/2
m <- N/2
while m > 1, [m=m/2
              (x,y) <- alternate block exchange(x,y,n,m)
              (x,y) <- shuttle-merge(x,y,n,m)
             ]
```

VI. THE DIAMOND SORT ALGORITHM

A full code follows, written in Fortran, with the vector operations appearing as subroutine calls. The code is in working form, but the vector-operation subroutines have not been optimized for vector computer use since this would depend greatly on the particular hardware and software facilities available. It is written, rather, to present the full Diamond sort procedure in clear and succinct form. The CALL's are introduced primarily to highlight the points at which parallel operations are called for because of their critical importance in determining the effectiveness of the program.

```
      SUBROUTINE DIAMND(X,Y,N)
      REAL X(0:N-1),Y(0:N-1)
C  SUBROUTINE CALLED BY: CALL DIAMND(S(1),S(1+NT/2),NT/2)
******
C  CREATE DIAMOND
      M=N
   10 CONTINUE
      CALL COMPAR(X,Y,N)
      IF(M.GT.1)THEN
         M=M/2
         CALL ALTBLK(X,Y,N,M)
         GO TO 10
      END IF
C  DIAMOND COMPLETE.
      M=N/2
C  REPEAT SHUTTLE-MERGE TILL M = 1.
   20 CONTINUE
      J=N
   21 CONTINUE
      IF(J.GT.M)THEN
```

```
         J=J/2
         CALL RVCOMP(X,Y,N,J)
         GO TO 21
      END IF
      IF(M.GT.1)THEN
         M=M/2
         CALL ALTBLK(X,Y,N,M)
         GO TO 20
      END IF
C  INTERLEAVE X AND Y
      CALL SHUFFL(X,Y,N)
      RETURN
      END
**********
      SUBROUTINE VECOPS
      REAL X(0:N-1),Y(0:N-1),U,V
C  COMPARE-EXCHANGE
      ENTRY COMPAR(X,Y,N)
      DO 10 I=0,N-1
         U=X(I)
         V=Y(I)
         IF(U.GT.V)THEN
            X(I)=V
            Y(I)=U
         END IF
  10  CONTINUE
      RETURN
C  ALTERNATE BLOCK EXCHANGE
      ENTRY ALTBLK(X,Y,N,M)
      DO 20 I=0,N-2*M,2*M
         DO 19 J=I,I+M-1
            U=Y(J)
            Y(J)=X(J+M)
            X(J+M)=U
  19  CONTINUE
  20  CONTINUE
      RETURN
C  REVERSE-COMPARE-EXCHANGE WITH OFFSET M
      ENTRY RVCOMP(X,Y,N,M)
      DO 30 I=0,N-M-1
         U=X(I+M9
         V=Y(I)
         IF(V.GT.U)THEN
            X(I+M)=V
            Y(I)=U
         END IF
  30  CONTINUE
      RETURN
******************
C  PERFECT SHUFFLE
******************
      SUBROUTINE SHUFFL(X,Y,N)
      REAL X(0:N-1),Y(0:N-1)
      REAL T(0:4095)
C  THE DIMENSION OF T ALLOWS THE SUBROUTINE TO
C  ACCOMMODATE N = 8192
      DO 10 I=0,N/2-1
         T(I)=X(N/2+I)
   10 CONTINUE
      DO 20 I=N/2-1,0,-1
         X(2*I)=X(I)
         X(2*I+1)=Y(I)
   20 CONTINUE
      DO 30 I=0,N/2-1
         Y(2*I)=T(I)
         Y(2*I+1)=Y(N/2+I)
   30 CONTINUE
      RETURN
      END
```

REFERENCES

1. H.K.Brock, B.J.Brooks, and F.Sullivan: Diamond, A Sorting Method for Vector Machines; BIT, 21-2, (1981),142-152.

2. D.E.Knuth: The Art of Computer Programming, Vol.3, Sorting and Searching; Addison-Wesley Publishing Company, Reading, Massachusetts (1973).

3. B.Mossberg: Sorting on the Cyber 205; Symposium on Cyber 205 Applications, Colorado State University, Fort Collins, Colorado (1982).

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 84-2936	2. Performing Organ. Report No.	3. Publication Date September 1984

4. TITLE AND SUBTITLE

Diamonds and Diamond Sorting

5. AUTHOR(S)

Dr. Eleazer Bromberg and Dr. Francis Sullivan

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

The present paper describes and analyzes the Diamond sort algorithm. The algorithm is designed for parallel operations, which makes it well suited to vector-computer architecture. The instruction sequences are fixed, without any branches. The algorithm therefore has a fixed complexity determined solely by the number of elements to be sorted, rather than by any pattern of distribution of element values. The Diamond sort is of special interest not only because of the unusual arrangement of its steps, but also because it introduces the concept of a Diamond as a set with a specific kind of partial ordering that is closely linked to the halving principle, which is used in the sort procedure.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Parallel and vector machines, sorting, complexity, algorithms

13. AVAILABILITY

[X] Unlimited
[ ] For Official Distribution. Do Not Release to NTIS
[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

23

15. Price

$7.00