JOURNAL OF COMPLEXITY 4, 330–355 (1988)

VLSI-Sorting Evaluated under the Linear Model

H. SCHRÖDER

Computer Science Laboratory, Australian National University, Canberra, ACT 2601, Australia

1. DIFFERENT MODELS OF VLSI COMPUTATION

Sorting has been one of the central problems of computation in the last 100 years. In 1880 the census data of the U.S.A. could not be evaluated. Motivated by this Hermann Hollerith developed a sorting machine. One hundred of such machines were able to evaluate 1890 census data (Knuth, 1973).

Sorting is also a theoretically interesting problem with significant practical relevance. Around 1960 this led to the development of a series of fast sorting algorithms for sequential machines. Around 1970 more than a quarter of the world computer capacity was involved in sorting.

Today, due to the fast growing use of data bases, the need for fast sorting algorithms has become even stronger. The development of VLSI technology allows for the realization of a new class of sorting algorithms that are by several magnitudes faster than those algorithms designed for sequential machines. This in turn started around 1980 the development of a new series of sorting algorithms that exploit the parallelism allowed for by the new technology (Bilardi, 1984; Thompson, 1983).

The practical use of these new algorithms will depend heavily on technological development. Similarly their theoretical evaluation depends heavily on the underlying hardware model (Thompson, 1983; Chazelle and Monier, 1981; Kung, 1982).

A VLSI chip can be viewed as a computation graph whose vertices are information processing devices and whose arcs are wires, that is, electrical connections responsible for information transfer as well as for power supply and distribution of timing waveforms. A given computation graph is to be laid out in conformity with the rules dictated by technology (Bilardi, 1984).

The main difference of the hardware models is the way they model signal propagation time. Most authors evaluate VLSI algorithm under the constant or logarithmic model, i.e., the signal propagation time is assumed to be at most proportional to the logarithm of the wire length. Some authors postulate that the linear model should be used at least for future technology (Chazelle and Monier, 1981; Vitanyi, 1984a,b), since it is predicted that signal propagation time increases dramatically in relation to gate switching time (Mangir, 1983; Thompson and Raghavan, 1984). In the linear model signal propagation time is assumed to be proportional to the wire length.

In this paper we investigate the impact of basing evaluation of VLSI sorting algorithms on the linear model. In Section 2 basic notations are introduced and the constant and linear models are defined.

In Section 3 the known lower bounds for VLSI sorting under the constant model are cited and the impact of the linear model on lower and upper bounds for AT^2 and AP^2 is investigated. Here new AT^2 and AP^2 optimal sorting algorithms are introduced. These are based on a comparison exchange cell that processes keys of length k as $\sqrt{k} \times \sqrt{k}$ bit matrices. It is shown that the execution time of all AT^2 optimal sorting algorithms is proportional to $\sqrt{kn}$ (n is the number of keys and k is their length).

In Section 4 chip-external sorting algorithms are introduced. Algorithms of this kind have been presented by several authors (Kim *et al.*, 1984; Todd, 1978; Bonuccelli *et al.*, 1984; Ja'Ja and Owens, 1985). The chip-external sorting algorithms presented here fit the linear model. They do not use a bus system and addressable memory. They use shift memory and are systolic in their nature. The data is circulated in a continuous flow several times through a VLSI sorting chip that performs a sort-split operation on blocks of data. It is shown that the throughput of the sorting chip determines the asymptotic time complexity of the chip-external sorting algorithm. Furthermore it is shown that this throughput is maximal for AT^2 or AP^2 optimal sorting algorithms.

2. The Basis of Evaluation

In this section the basic definitions needed for complexity analysis are given and the hardware model the complexity analysis is based on is defined. The hardware model reflects the restrictions of parallelism that are enforced by the technology to be used.

2.1. *Basic Definitions of Asymptotic Analysis*

In this paper the symbols i, j, k, m, n, n_0, n_S are used for nonnegative integers. The symbols n, n_0, n_S, and N represent numbers of keys and k is used for the length of a key. The symbols c, c_1, c_2, p, q, r, s, and x represent real numbers.

DEFINITION 2.1. Let $f = f(n)$, $g = g(n)$, $h = h(n)$ be functions of n.

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 : \forall n > n_0 f(n) \leq c \cdot g(n)$$
$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 : \forall n > n_0 f(n) \geq c \cdot g(n)$$
$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$
$$f(n) \in [\Omega(g(n)), O(h(n))] \Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) = O(h(n)).$$

The mappings O, Ω, and Θ introduce classes of functions of similar complexity ignoring constant factors. The term *asymptotic analysis* is used throughout this paper to denote the evaluation of complexity functions by means of the above mappings.

DEFINITION 2.2. $f(n) = \Phi(g(n)) \Leftrightarrow \forall \varepsilon > 0 : f(n) \in [\Omega(n^{-\varepsilon} \cdot g(n)), O(n^{\varepsilon} \cdot g(n))]$.

Definition 2.2 introduces a new classification of complexity functions which is characterized by the following lemmas.

LEMMA 2.1. $f(n) = \Phi(g(n)) \Leftrightarrow \lim_{n\to\infty} (\log f(n) - \log g(n))/\log n) = 0$.

LEMMA 2.2. $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Phi(g(n))$.

Since the proofs of these lemmas are straightforward they are omitted here. The inverse of Lemma 2.2 is not true, thus Θ is a true refinement of Φ. While Θ ignores only constant factors, Φ also ignores logarithmic factors. In a similar way other classifications of complexity functions can be given ignoring factors of different complexity.

In this paper we mainly use O, Ω, Θ. Φ is needed in Section 4.3.

2.2. *The VLSI Hardware Model*

There are several levels of abstraction to look at a VLSI chip. Here we deal with two of these levels.

On the upper level the VLSI chip can be viewed as a computation graph whose vertices are information processing devices and whose arcs are wires, that are electrical connections used for information transfer, power supply, or distribution of clock pulses. Throughout this paper we assume that there is a global clock synchronizing the actions of the information processing devices.

On the lower level of abstraction we look at an embedding of the computation graph in the Euclidean plane. Here information processing devices and wires are represented as areas in the Euclidean plain. Wires are represented as connected sets of rectangular areas, each of them is called a wire segment. Information processing units are called gates or nodes below. The computation graph is to be laid out in conformity with the rules dictated by technology. These rules whose justification is sometimes based on a rather delicate and subtle analysis (Bilardi, 1984) will be given below.

The hardware model introduced here has been taken from Bilardi (1984) and Thompson (1983), omitting those rules that have no effect on the asymptotic behavior of the analyzed design.

Assumption 1: *The area.*

(a) Wire segments have minimal width $\lambda > 0$ and a length $l \geq \lambda$.

(b) The length of a wire is the sum of the lengths of its wire segments.

(c) Each point of the Euclidian plain is in at most $\nu \geq 2$ wire segments. (This assumption refers to the bounded number of layers on a chip.)

(d) The area assigned to a node has $O(1)$ units of area (the unit of area λ^2). Each node area is connected to (overlaps with) $O(1)$ wire areas (bounded degree of nodes).

(e) Areas assigned to nodes do not overlap.

(f) The area of a VLSI circuit is the number of units of area included in the smallest rectangular containing the embedding of its computation graph in the Euclidean plain.

Assumption 2: *The time.*

(a) In each clock period each wire carries at most one 1-bit signal (unit bandwidth). This assumption restricts the use of wires for storage purposes.

(b) Gates have a switching time of $O(1)$ time units.

(c) A bit requires $O(1)$ units of time to propagate along a wire of arbitrary length. Under this assumption the model will be called the *constant model.*

(c′) A bit needs $\Theta(l)$ to propagate along a wire of length l. Under this assumption the model is called the *linear model.*

(d) The computation time of an algorithm is the maximal number of units of time between the beginning and the end of a computation.

(e) The period of an algorithm is the number of units of time between the beginning of the input of one problem instance and the beginning of the input of the next problem instance.

We assume that all gates are synchronized by a global clock. The clock period has $O(1)$ time units under the constant model and $\Theta(d)$ time units under the linear model, where d is the maximal wire length of the analyzed design.

The use of the linear model for asymptotical analysis is equivalent to the use of the constant model with the additional protocol assumption that only local communication is allowed. An algorithm satisfies this local communication assumption if there is a VLSI technical realization such that the length of the wires is bounded by some constant, which is independent of the problem size.

Assumption 3: *The node functions.*

(a) Each node function can be described by a finite state machine (of type Moore automata). The state of the Moore automata is represented as bit vector with $O(1)$ state bits. These state bits are newly determined at each clock pulse depending on the old current state and on the input signals. The output signals depend on the state only. According to this characterization by finite state machine we split up the ports of a gate into input and output ports and assign directions to the arcs (wires).

(b) If output from more than one gate is sent to the same input port, then these signals have to be identical at any time.

Assumption 4: *The protocol.*

(a) Each bit of the input data can be read only once (*semelective*).

(b) Input and output of data are available at prespecified (instance independent) time (*when-oblivious*).

(b′) The order of input and output of data is prespecified (instance independent). Condition (b′) is an alternative to (b) and is a weaker condition (see below).

(c) Input and output of data are available at prespecified (instance independent) ports (*where-oblivious*).

(d) All input and output ports are at the boundary of the layout region.

(e) All bits of a given input word enter the chip at the same input port (*word-local protocol*).

Whenever the protocol assumption "when-oblivious" is asked for it should be checked whether the weaker condition (b′) would be sufficient. Condition (b′) has the advantage that for example in the case of sorting algorithms it would allow one to have the sorting time dependent on the instance. Thus almost sorted data could be processed faster than random data.

Protocol condition (d) seems to be too strong for some upcoming technologies and therefore might be dropped.

Protocol condition (e) might not always be adequate but it seems reasonable especially when the data is supplied by some external storage according to this protocol assumption.

Assumption 5: *The sorting problem.*

(a) A problem instance of a VLSI sorting chip S consists of n_S keys of length k (keys are binary numbers).

(b) Let $\mathbf{X} = (X_1, X_2, \ldots, X_{n_S})$ be the input and $\mathbf{Y} = (Y_1, Y_2, \ldots, Y_{n_S})$ be the output of the sorting problem. Then $\mathbf{Y}$ is a permutation of $\mathbf{X}$ with $Y_i \leq Y_{i+1}$ $(i \in \{1, 2, \ldots, n_S - 1\})$.

Since most of the analysis on sorting algorithms that can be found in the literature has been done using the constant model, in this paper some of the results stated for the constant model are reevaluated under the linear model.

The linear model is equivalent to the constant model with the additional local communication assumption. Therefore lower bounds that are valid under the constant model are also valid under the linear model, and upper bounds presented in this paper for the linear model are also valid under the constant model.

3. Lower and Upper Bounds for the Asymptotic Behavior of VLSI Sorting Algorithms

In this section the lower bounds that are valid under the constant model are given (Bilardi, 1984). Then it is shown how these bounds change, when the linear model is applied. Then upper bounds are developed for the linear model, which show that the lower bounds cannot be improved any further.

All analysis done in this paper refers to medium or large key length (Bilardi, 1984), i.e., $k - \log n = \Omega(\log n)$. Thus it is achieved that all keys of a given sorting problem can be different.

3.1. *Lower Bounds*

Bilardi gives lower and upper bounds for sorting algorithms on the measures of complexity A, T, and AT^2:

For $k - \log n = \Omega(\log n)$

(a) $A = \Omega(n \log n)$—this lower bound is gained computing the storage space needed to store the unsorted data.

(b) $T = \Omega(\log n)$—the bounded degree assumption (1d) is used to prove this lower bound.

(c) $AT^2 = \Omega(n^2 \log^2 n)$ for $T \in [\Omega(\log n), O(\sqrt{n} \log n)]$—this lower bound is gained calculating flow of information across cuts of VLSI designs.

The measure of complexity AT^2 is justified by the lower bound (c). In Section 4.3 there will be another justification for it.

Since these lower bounds given by Bilardi are based on the constant model, they are also valid under the linear model as is explained in Section 2. It can be shown however that the lower bounds for AT^2 can be improved under the linear model.

THEOREM 3.1. *Under the linear model all realizations of sorting algorithms satisfy* $T = \Omega(\sqrt{n} \log n)$.

Proof. Since in sorting every output bit depends on every input bit (Duris *et al.*, 1985), we know that all input and output bits are within a circle of radius $O(T)$. Furthermore all information that has any influence on the output must be within a radius of $O(T)$ around the corresponding output port. Thus the whole area used for sorting is limited by $A = O(T^2)$. Using $A = \Omega(n \log n)$ and $AT^2 = \Omega(n^2 \log^2 n)$ the theorem follows immediately. ■

So under the linear model AT^2 optimal sorting algorithms can only exist for $A = \Theta(n \log n)$ and $T = \Theta(\sqrt{n} \log n)$.

For the LSSS-sorter (Lang *et al.*, 1985) there are realizations such that $A = \Theta(nk)$ and $T = \Theta(\sqrt{nk})$. In Section 3.2 it is shown that the LSSS-sorter can be modified such that it becomes AT^2 and AP^2 optimal.

So the asymptotic area and time complexity of all other sorting algorithms is greater or equal to that of LSSS sort.

3.2. AT^2 *Optimal Sorters under the Linear Model*

In Lang *et al.* (1985) the LSSS sorting algorithm is presented with $A = \Theta(nk)$ and $T = \Theta(\sqrt{nk})$. It uses local communication only. The comparator-exchange cell suggested for its realization consists of a bit serial comparator and two k bit shift registers. It executes a comparison of two k bit numbers in time $T_V = \Theta(k)$ using an area $A_V = \Theta(k)$. The total area for the realization of this sorting algorithm is $\Theta(nA_V)$ and its execution time is $\Theta(\sqrt{n}T_V)$.

Replacing in a realization of the LSSS-sorter all k bit shift registers by $\sqrt{k} \times \sqrt{k}$ matrices and all bit serial comparators by matrix comparators M with $T_{MC} = \Theta(\sqrt{k})$ and $A_{MC} = \Theta(k)$ results in a sorting algorithm with $A = \Theta(nk)$, $T = \Theta(\sqrt{nk})$, and $AT^2 = \Theta(n^2k^2)$.

For the resulting algorithm the input and output forms a square matrix each component of which is a square matrix too. Therefore it is called the *M–M*-sorter. The design of the matrix comparator needed for its realization is given below.

The matrix comparator is a systolic architecture somewhat similar to the bit serial comparator. The keys are stored as bit matrices and are pumped through the architecture's most significant columns first. While they move through the comparator, each of their bit positions is compared. As long as there are no differences the columns can be output unchanged. As soon as the most significant position where the two processed numbers differ is found, the corresponding information has to be spread to all bit positions of less significance, so that those columns can be exchanged if necessary. A floorplan for such an architecture is shown in Fig. 3.1.

Each operation executed by the cells of this design consists of two phases. In the first place of each clock period each cell puts the information available to its input ports into its registers: $B = b_1 b_2$ and $C = c_1 c_2$. C and B can carry the values 00, 11, 01, and 10. $C = e$ denotes $C = 00$ or $C = 11$. The contents of the registers B are shifted unchanged to the right neighbors, so that the structure of the matrices X and Y stays unchanged.

The second phase of each clock period serves to spread the result of the comparison. The comparator cells C execute in this phase the following operation (Let C' be the succeeding state of C),

$$c_1' c_2' = C' := \begin{cases} E & \text{if } E \neq e \qquad (***) \\ S & \text{if } E = e \wedge S \neq e \quad (*) \\ N & \text{if } E = S = C = e \quad (**) \\ C & \text{otherwise,} \end{cases}$$

where E, S, and N denote the east, south, and north neighbors of register C. The connections of the C cells to their north and south neighbors are not represented in Fig. 3.1. In the left column the contents of the C and B registers are identical right after the first place. Each of them contains a bit of X and a bit of Y with identical significance.

The cells denoted by R execute the exchange operation corresponding to the result of the comparison (R' is the succeeding state of R):

$$r_1' r_2' = R' := \begin{cases} b_2 b_1 & \text{if } c_1 c_2 = 10 \\ b_1 b_2 & \text{otherwise.} \end{cases}$$

Proof of correctness of the matrix comparator. Let $C(i, t)$ be the matrix of the C registers of the leftmost column at time t at the end of the first phase:

$$C(i, t) = x_{ir-t} y_{ir-t} \textit{ with } t = 0, 1, \ldots, w \textit{ and } w = \sqrt{k} - 1.$$

Let j, p be the index of the most significant position where $C(j, p) = a \neq e$. Let $C'(i, t)$ be the matrix of the C registers of column w at time $t + w$ at the end of phase two (see Fig. 3.2).

The value a has to be distributed at least to all C cells that belong to less significant positions of the binary numbers X and Y than $C(j, p)$. Row (*) guarantees that the value a is distributed to all cells on top of it in its

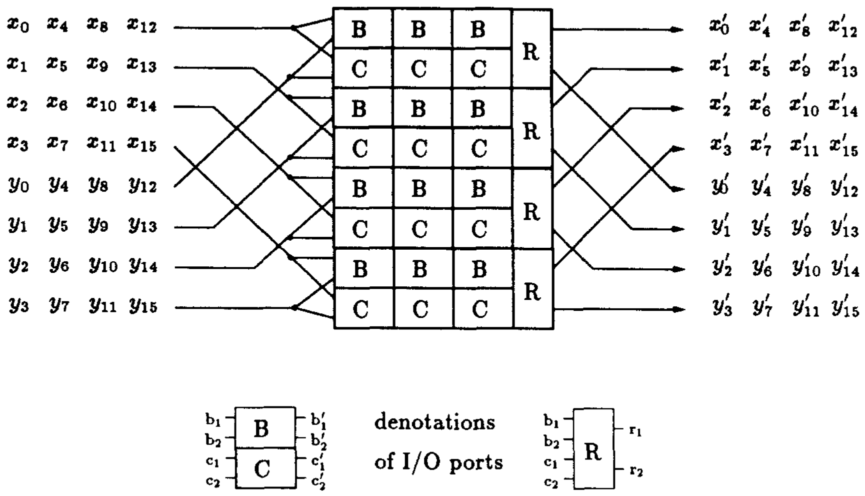

x_0 x_4 x_8 x_{12}
x_1 x_5 x_9 x_{13}
x_2 x_6 x_{10} x_{14}
x_3 x_7 x_{11} x_{15}
y_0 y_4 y_8 y_{12}
y_1 y_5 y_9 y_{13}
y_2 y_6 y_{10} y_{14}
y_3 y_7 y_{11} y_{15}
B
C
R
x'_0 x'_4 x'_8 x'_{12}
x'_1 x'_5 x'_9 x'_{13}
x'_2 x'_6 x'_{10} x'_{14}
x'_3 x'_7 x'_{11} x'_{15}
y'_0 y'_4 y'_8 y'_{12}
y'_1 y'_5 y'_9 y'_{13}
y'_2 y'_6 y'_{10} y'_{14}
y'_3 y'_7 y'_{11} y'_{15}
b_1 b_2 c_1 c_2
b'_1 b'_2 c'_1 c'_2
denotations of I/O ports
r_1 r_2

FIG. 3.1. A matrix comparator.

column. Row $(**)$ achieves that a is sent to all positions below in its column. So at most $\sqrt{k} - 1$ clock pulses after $C(j, p)$ is generated, so in time before $C(j, p)$ reaches an R cell F have $C(i, p) = a$ for $i = 0, \ldots, w$. Row $(***)$ guarantees that in the following time units all columns with a smaller column index than p receive the value a. So the R cells output the matrices X and Y according to a. ■

The VLSI technical realization of the architecture presented here consumes a larger than necessary area. Easy thoughts show what we can do

$$\begin{array}{cccccc}
 & & & p & & \\
 & & & \downarrow & & \\
 & x & x & x & e & e \\
j \rightarrow & x & x & a & e & e \\
 & x & x & e & e & e \\
 & x & x & e & e & e \\
 & x & x & e & e & e \\
 & & & C(i, t) & &
\end{array}
\qquad
\begin{array}{cccccc}
 & & & p & & \\
 & & & \downarrow & & \\
 & a & a & a & e & e \\
j \rightarrow & a & a & a & e & e \\
 & a & a & a & e & e \\
 & a & a & a & e & e \\
 & a & a & a & e & e \\
 & & & C'(i, t) & &
\end{array}$$

FIGURE 3.2

with $k/2$ comparators (*CP*) (see Fig. 3.3). All the cells named *B*, *SC*, and *SR* in Fig. 3.3 are just shift registers. The *R* cells are identical to the *R* cells of Fig. 3.1.

In the leftmost $\sqrt{k}/2$ columns of this design pairs of bits of identical significance are moved to the middle of the columns through the registers *SC* to the comparators *CP*.

The comparator cell has a two bit state (referred to as c, initialized as $c = c_1c_2 = 00$). The successive state depends on the actual state of its three pairs of bits that *CP* receives as input. The top pair and bottom pair contain bits of identical significance of the numbers to be compared (they are denoted as t and b). The middle pair (denoted by m) is the state of its left neighbor.

For the leftmost comparator the middle input is set to $m = 00$ (not shown in Fig. 3.3). These four bit pairs have different priorities reflecting the significance of the bit positions they represent: $c > b > m > t$. If none of these bit pairs is equal to e then the successive state of the cell will become 00. Otherwise it receives the value of the bit pair with highest priority among those which are unequal to e. Thus

$$c_1'c_2' = C' := \begin{cases} c & \text{if } c \neq e \\ b & \text{if } c = e \wedge b \neq e \\ m & \text{if } c = b = e \wedge m \neq e \\ t & \text{otherwise.} \end{cases}$$

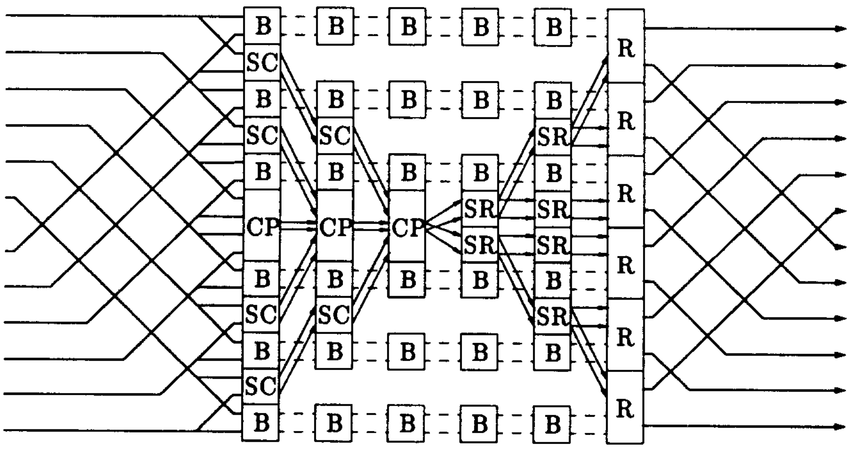


FIG. 3.3. A variation of the matrix comparator.

In the $\sqrt{k}/2$ right columns the result of the comparison is handed through the registers SR to the R cells. The contents of the B registers is just moved to the right so that it can be output by the R cells according to the result of the comparison.

Though this variant of the matrix comparator has the same asymptotical area complexity as the first design, its real area consumption is significantly less since the comparator cells are much less area consumptive than one bit shift registers. (Here we assume that the size and speed of the comparator cells of the two different designs are similar, otherwise our statement would hold only for large k.) Such observations are of no significance as long as like here we just want to use the design to prove an upper bound. They are important though in case VLSI technical realizations of such comparators are planned.

The matrix comparator above is designed for square matrices. This design can be generalized to arbitrary $r \times s$ matrices: Their area requirement is $A_{\mathrm{MC}} = O(r^2)$, their execution time is $T_{\mathrm{MC}} = O(r + s)$, and their period is $P_{\mathrm{MC}} = O(s)$. Here the storage area for the data matrices is not included in the area requirements.

In many cases, for example, if the matrix comparator is used in the M–M-sorter and $s > r$, two storage areas of size $r \times (s - r)$ are needed in front of each comparator. This would add up to an area requirement of $\Theta(rs)$ for a comparator cell with two registers.

The area–time complexity for an M–M sorter using $r \times s$ matrices to store the numbers is

$$AT^2 = O(n \cdot A_{\mathrm{MC}} \cdot n \cdot T^2_{\mathrm{MC}}) \textit{ with } A_{\mathrm{MC}} = \Theta(r^2 + \max\{r(s - r), 0\}) \textit{ and}$$

$$T_{\mathrm{MC}} = \Theta(r + s). \textit{ So for } s \geq r : AT^2 = O(n^2rs^3) \textit{ and for}$$

$$s < r : AT^2 = O(n^2r^4).$$

$$\textit{With } rs = k \textit{ and } r = \Theta(\sqrt{k}) \textit{ we have } AT^2 = O(n^2k^2).$$

If r and s differ from $\Theta(\sqrt{k})$ then the resulting M–M-sorter is not AT^2 optimal anymore. There are AP^2 optimal EA-parallel sorting algorithms using the $r \times s$ comparator for any choice of r and s. Their realizations do not need the storage matrices in front of the comparator cells. These are presented in Section 3.3.

Rearranging the input. Bilardi (1984) gives the protocol assumption, that all bits of a key are input to the same input node (word-local protocol, see Section 2). The M–M-sorter violates this protocol assumption, since it needs the keys as $\sqrt{k} \times \sqrt{k}$ matrices. Algorithm SEMA (*se*rial $\rightarrow$ *ma*trix) does the transformation from bit serial to $\sqrt{k}$ bit parallel in k steps. It is described below.

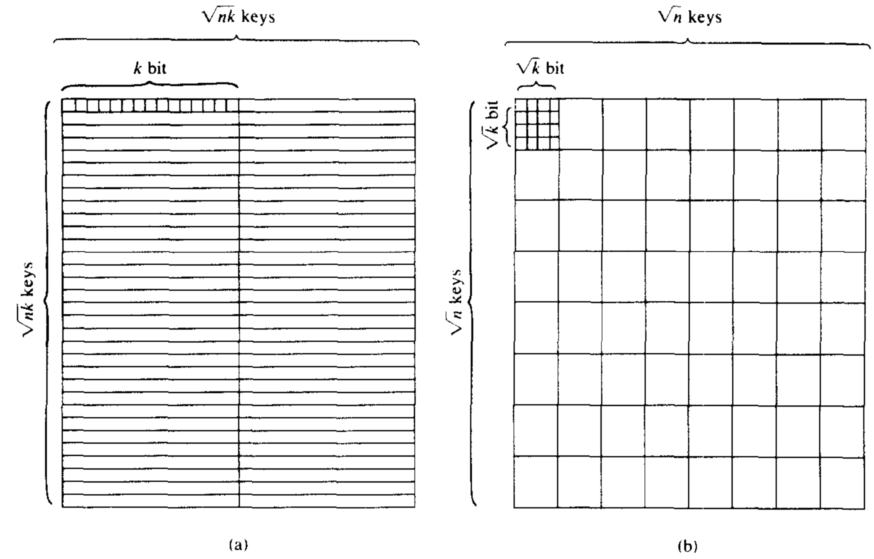


FIG. 3.4. Input (a) and output (b) of algorithm SEMA.

For didactical reasons algorithm SEMA is described as a static rather than as a systolic version. As shown in Fig. 3.4a it reads n k bit numbers in a bit serial fashion into a $\sqrt{nk} \times \sqrt{nk}$ matrix of processors and generates a $\sqrt{n} \times \sqrt{n}$ matrix each component of which is a $\sqrt{k} \times \sqrt{k}$ matrix and represents a key (see Fig. 3.4b).

Algorithm SEMA consists of the iterative application of algorithm TOSI (two matrices positioned on *to*p of one another are transformed into two matrices positioned *si*de by side). TOSI transforms in an $2r \times s$ array of processors two $r \times s$ keys into two $2r \times s/2$ keys ($rs = k$).

Algorithm TOSI(r, s). Each of the $2rs$ processors has two one bit registers A and B (see Fig. 3.5).

After input of the data all bits of the numbers $X = x_0 \cdots x_{rs-1}$ and $Y = y_0 \cdots y_{rs-1}$ are stored in the A registers (see Fig. 3.6a).

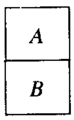


FIG. 3.5. A processing element with two registers.

Step 1. The rs processors in the left half of the $2r \times s$ processor array copy the contents of the A registers into the B registers (Fig. 3.6b).

Step 2. In the r upper rows of processors the contents of the B registers are shifted from the left half to the right half. In the r rows below the contents of the A registers are shifted from the right half to the left half (Fig. 3.6c).

Step 3. In the left half the bits are shifted upward and in the right half they are shifted downward until all bits are located in the A registers (Fig. 3.6d).

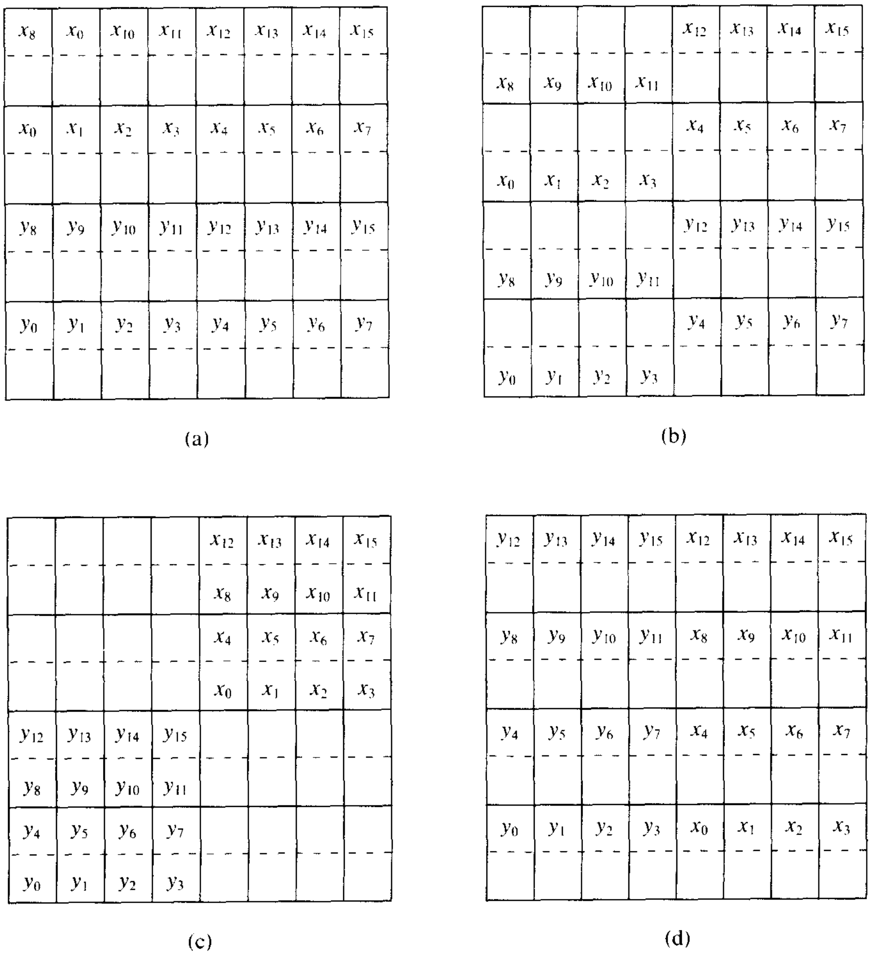


FIG. 3.6. Transforming two 2 × 8 bit numbers into two 4 × 4 bit numbers.

The time complexity of algorithm SEMA. Algorithm TOSI(r, s) needs $1 + 2r + s/2$ units of time. So Algorithm SEMA needs

$$T_{\text{SEMA}} = \frac{1}{2}\log k + 2 \cdot \sum_{i=0}^{1/2\log(k-1)} 2^i + \frac{1}{2} \cdot \sum_{i=1/2\log(k+1)}^{\log k} 2^i$$

$$= \frac{1}{2}\log k + \sum_{i=1}^{1/2\log k} 2^i + \sum_{i=1/2\log k}^{\log(k-1)} 2^i = \frac{1}{2}\log k + \sqrt{k} + k - 2 = O(k)$$

units of time.

The time complexity of the M–M-sorter is $\Theta(\sqrt{nk})$. Thus for $k = O(n)$ the transformation executed by SEMA can be incorporated into it, without changing its asymptotic area or time complexity. Thus we gain the following result:

THEOREM 3.3. *There are sorters which, evaluated under the linear model, have area complexity* $A = \Theta(nk)$ *and time complexity* $T = \Theta(\sqrt{nk})$.

Thus we have shown that under the protocol assumption of local communication the lower bound $AT^2 = \Omega(n^2k^2)$ for sorting medium sized keys cannot be improved.

There are also systolic realizations for the sorting algorithms presented by Thompson and Kung (1977) and Kumar and Hirschberg (1983) and Nassimi and Sahni (1979). These could also make use of the matrix comparator achieving the same asymptotic area and time complexity as the M–M-sorter. Their real area and time complexity differ from those of the M–M-sorter only by small constant factors.

3.3. AP^2 *Optimal Sorters*

For many applications it is more important to have sorting algorithms with a small period than sorting algorithms with short execution time. Under the constant model $AP^2 = \Omega(n^2k^2)$ holds for medium sized keys. This can be proved in evaluating information flow across bisections in the same way the AT^2 lower bound is proved (Bilardi, 1984), taking into account that the flow of information necessary for the solution of a sorting problem is identical to the flow of information per period summed up over all problems that are under progress in a period.

To show that this lower bound cannot be improved, variations of realizations of odd–even transposition sort and of the M–M-sorter are used.

The odd–even transposition sort as presented by Thompson (1983) is AP^2 optimal for $P = k$. Replacing in its realization the bit-serial comparator by an $r \times s$ matrix comparator results in an AP^2 optimal sorter with $P = s$. Since s can be picked arbitrarily out of $[1, k]$ $(s|k)$ we get Lemma 3.4.

LEMMA 3.4. *Under the linear model there are AP^2 optimal sorters for all $P \in [\Omega(1), O(k)]$.*

The M–M-sorter for $\sqrt{n} \times \sqrt{n}$ matrices can be used to sort $r \times \sqrt{n}$ matrices $(r|\sqrt{n})$ with period $P = \Theta(r\sqrt{k})$. The area is $A = \Theta(nk)$ and $AP^2 = \Theta(r^2nk^2) = \Theta(N^2k^2)$ with $N = r\sqrt{n}$.

Since $r \in [1, \sqrt{n}]$ can be chosen arbitrarily with $r|\sqrt{n}$ we get Lemma 3.5.

LEMMA 3.5. *Under the linear model there are AP^2 optimal sorting algorithms for all $P \in [\Omega(\sqrt{k}), O(\sqrt{nk})]$.*

Putting these two lemmas together we get Theorem 3.4.

THEOREM 3.4. *Under the linear model there are AP^2 optimal sorters for all $P \in [\Omega(1), O(\sqrt{nk})]$.*

While under the linear model there are AT^2 optimal sorters only for $T = \Theta(\sqrt{nk})$ there are AP^2 optimal sorters for a large range for P. All upper bounds presented here can also be used under the constant model.

4. CHIP-EXTERNAL SORTING

The complexity measure AT^2 represents a tradeoff for sorting algorithms. This tradeoff carries through to their realizations; i.e., there are sorting chips allowing for large problem sizes that are relatively slow, and there are sorting chips with small problem sizes that are relatively fast. The characteristic data of a sorting chip S that we are going to take into account are its maximal problem size it can handle n_S, its sorting time to sort n_S keys t_S, and its period p_S.

Of course it depends on the set of sorting problems that have to be solved which of these parameters is most important. If the throughput of sorting problems is most important, we would want a small period. We also might need a small sorting time, or the size of the sorting problems might force us to use one of the relatively slow sorting chips, since they have a higher capacity.

When the problem size exceeds the chip capacity the sorting chip can be used in a chip-external sorting algorithm. There are several external sorting algorithms presented in the literature, having similarities to the designs presented here.

W. Kim *et al.* (1984) have presented an external sorting algorithm based on the two-way merge algorithm, which is similar to one of the designs presented in this paper, using the same external sorting scheme and achieving the same performance. The internal sorting algorithm they use is tailored toward its use in a query processor and thus not optimal in general.

S. Todd (1978) uses log n processors to sort n numbers in time $O(n)$ using $O(\log n)$ queues of variable length. This makes their design more complex than the designs presented here. Furthermore is the number of processors used in their design dependent on n, while we can use any constant number of processors.

M. A. Bonuccelli *et al.* (1984) use a mesh of trees to sort blocks of data and a RAM to store the data. Thus they need long distance communication. Evaluated under the linear model such designs would be ruled out, since signal propagation time then leads asymptotically to unacceptable low performance.

J. Ja'Ja' and R. M. Owens (1985) use p processors to sort n numbers in time $O(n/\sqrt{p} + n^2/p^2)$. They do stress the advantage of using serial memory. Their designs are outperformed by the algorithms of this paper under the assumption that only a constant number of processors is used.

Here we present two such external sorting algorithms that can be realized on a single board and use a sorting chip S to perform a sort split operation on blocks of data.

Both algorithms are based on a similar hardware structure (see Fig. 4.1). *CI* is the input control unit and *CO* is the output control unit. All the other building blocks (*M*00–*M*11) in Fig. 4.1 consist of data chips. These data chips are huge shift registers. The main idea behind this structure is that it is systolic and can be realized using local communication only. So the evaluation of these algorithms will also be valid under the linear model. There is no bus system and no addressable memory. The data is

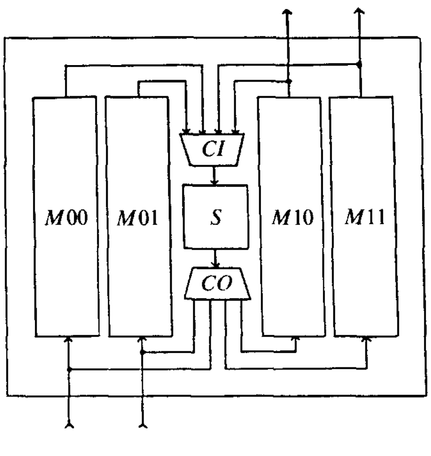


FIG. 4.1. A one board chip-external sorter.

continuously pumped through the data chips and the sorting chip directed by the control units.

The input control unit *CI* determines the shift memory from which to read. The output control unit determines where the sorted blocks of data go. The sorting chip S sorts two blocks of data of size $n_S/2$ into one sorted block of size n_S.

Since the data chips consist of shift registers only and shift registers are usually part of the sorting chips, it can be assumed that the data chips can be driven at the same clock frequency as the sorting chip. Therefore the time complexity of such a design is totally determined by the parameters n_S, t_S, and p_S of S and the problem size N.

The number of pins needed for the data chips is less than or equal to the number of pins of the sorting chip. So the whole construction seems technically feasible whenever the sorting chip is technically feasible.

If addressable memory and bus systems are used instead, as suggested by Miranker *et al.* (1983), the access time to the memory could determine the time complexity of the whole construction.

A significant problem using VLSI sorting chips might be the data rates that the chip has to be supplied with. In case the data rates needed by the sorting chip are higher than the environment can supply, the construction suggested here can balance the chip data rates with those achievable by the environment. The data can be read using the low data rate and can be sorted at a high data rate. In order to do this it is suitable to introduce a third pair of shift registers, that can be used to handle input and output while the other two pairs are involved in sorting.

Both sorting algorithms presented here start in a situation where 2^m blocks of unsorted data of $n_S/2$ keys each are located in the shift memories $M00$ and $M01$. The shift memories $M10$ and $M11$ are empty.

4.1. *A Systolic Realization of Batcher's Bitonic Sorter on Blocks of Data*

Batcher's bitonic sorting algorithm sorts 2^m keys in $(m - 1)/2$ stages. In each of these stages each of the 2^m keys is involved in a comparison exchange operation. The algorithm does iterative merging of bitonic sequences and each stage is realized by an algorithm of the "descend" class (Preparata and Vuillemin, 1981). *B*atchers *b*itonic sort on *b*locks of data (BBB(S)) is a systolic realization of Batchers bitonic sorting algorithm. Each of its stages is realized shifting the whole set of data through the sorting chip S. The sorting chip S sorts all pairs of blocks that are in corresponding positions in the two shift registers used as input memory.

Executing part B of algorithm BBB(S) once does the sort-split operation on all 2^{m-1} pairs of neighboring data blocks and is the realization of one stage of delay of the bitonic sorting net (Knuth, 1973). It needs 2^{m-1} ·

$p_S + t_S - p_S$ units of time. Afterward $Mq0$ and $Mq1$ and S are empty and all the data are contained in $M\bar{q}0$ and $M\bar{q}1$. So after input and output memory has been swapped ($q := \bar{q}$) part B can be executed again. Part B is executed $(m^2 + m)/2$ times. Thus the time complexity of algorithm BBB(S) is

$$T_{\mathrm{BBB(S)}}(N) = (2^{m-1} \cdot p_S + t_S - p_S) \cdot (m^2 + m)/2.$$

$\mathrm{bit}_p(i)$ determines where the sorted blocks go and changes every 2^p steps while the counter i runs from 0 to $2^{m-1} - 1$, so that each execution of part B involves the "perfect shuffle operation" (PS in Fig. 4.2) on groups of 2^p blocks.

ALGORITHM BBB(S)

```
begin
$p := 0$; $q := 0$
for $j := 0$ to $m - 1$ do
A   for $k := 0$ to $j$ do
    begin
        if $k = j$ then $p := j$
B       for $i := 0$ to $2^{m-1}$ do
        begin
            Read one block each
              from $Mq0$ and $Mq1$.
            Sort the $n_S$ keys
              in ascending order if $\mathrm{bit}_j(i) = 0$,
              in descending order otherwise.
            Shift the two resulting blocks
              to $SP\bar{q}r$ with $r = \mathrm{bit}_p(i)$.
        end
        $q := \bar{q}$
    end
end
```

The sequences of blocks denoted by L_i', R_i' in Fig. 4.2 are the results of applying the "sort-split operation" to the sequences of blocks L_i, R_i.

The positions of the blocks in the shift memory are denoted as shown in Fig. 4.3.

The perfect shuffle operation moves a block from position Z to position Z', where the binary number Z' results from the binary number Z rotating the rightmost $j + 1$ bits of Z by one position to the right (Knuth, 1973).

For $k = 0$ the indices of neighboring blocks differ only at bit j. So their difference is 2^j. Each execution of part B decreases this difference by the

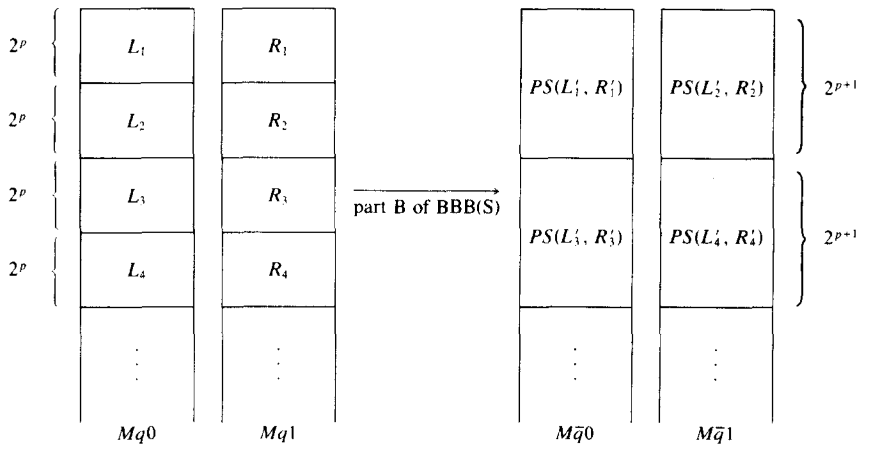


FIG. 4.2. The execution of one merge step.

factor 2. For $k = j$ it becomes 1. Thus part A of algorithm BBB(S) is an algorithm of the "descent class" (Preparata and Vuillemin, 1981) and BBB(S) is a realization of the bitonic sorting algorithm. In Fig. 4.4 the block positions and the values of the variables j, k, i, p, q are given for all points of time immediately before and after each execution of part B of algorithm BBB(S) for $m = 3$. Under the assumption $2^m > t_S/p_S$ the time complexity of algorithm BBB(S) is

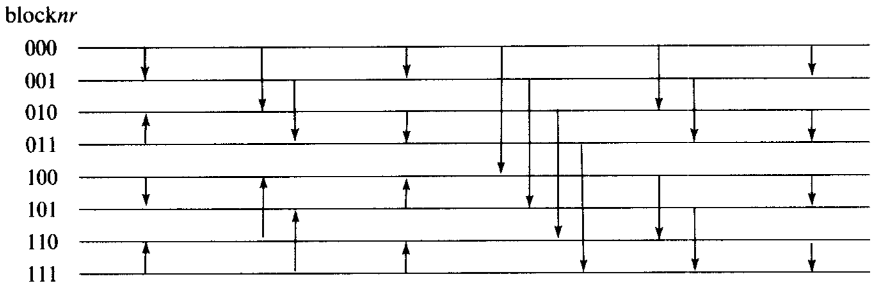
$$T_{\mathrm{BBB(S)}} = \Theta(m^2 \cdot 2^m \cdot p_S) = \Theta\left(\frac{N}{n_S} \cdot p_S \cdot \log^2 \frac{N}{n_S}\right).$$

For $N = \Omega((n_S)^a)$ with $a > 1$ we get

$$T_{\mathrm{BBB(S)}} = \Theta\left(\frac{p_S}{n_S} \cdot N \cdot \log^2 \mathrm{N}\right).$$

000	001
010	011
100	101
110	111
$Mq0$	$Mq1$

FIG. 4.3. Indices of block positions.



i	$j = 0$, $k = 0$, $p = 0$		$j = 1$, $k = 0$, $p = 0$		$j = 1$, $k = 1$, $p = 1$		$j = 2$, $k = 0$, $p = 1$		$j = 2$, $k = 1$, $p = 1$		$j = 2$, $k = 2$, $p = 2$	
000	000	001	000	010	000	001	000	100	000	010	000	001
001	010	011	001	011	010	011	001	101	100	110	010	011
010	100	101	100	110	100	101	010	110	001	011	100	101
011	110	111	101	111	110	111	011	111	101	111	110	111
↑ $\text{bit}_0(i)$											$Mq0$	$Mq1$

FIG. 4.4. The values of the variables during execution of BBB(S).

So the time complexity of BBB(S) is inversely proportional to the throughput $d_S := n_S/p_S$ of the sorting chip used.

So d_S becomes the evaluation criterion for sorting chips if they are used in this chip-external sorting scheme.

For two reasons BBB(S) is especially attractive for a VLSI technical implementation. On the one hand, it takes advantage of the short period of systolic sorting algorithms S. On the other hand the technical realization of its control units is very simple. The input control unit just consists of "or" gates and the output control unit does not need more than three counters for its realization.

In spite of that there is a significant disadvantage of this algorithm: The whole set of data has to be pumped $\Omega(\log^2 N)$ times through the sorting chip, which is the number of stages of delay of the corresponding sorting net (Knuth, 1973). As shown in Section 4.2 there are chip-external sorting algorithms that only need to pump the set of keys $O(\log N)$ times through S.

4.2. *A Systolic Realization of Two-Way Merge on Blocks of Data*

In comparison to BBB(S) *t*wo-*w*ay merge on *b*locks of data (TWB(S)) presented below seems less elegant. Its control structure is more complex and is data dependent. Its time complexity is determined by t_S and not by p_S, so that it does not take advantage of short periods of systolic sorting algorithms S.

The main advantage of TWB(S) is that the whole set of data has to be shifted through S only $O(\log N)$ times. Mehlhorn (1984) shows that the number of comparison exchange operations of two-way merge is minimal. Therefore the time complexity of TWB(S) is asymptotically optimal.

The hardware structure for TWB(S) differs from that of BBB(S) only slightly. The control units CI and CO perform more complex operations and after each sort-split operation one block of the sorted result is kept in

ALGORITHM TWB(S)

```
begin
  $q := 1$ ; $p := 0$;
  for $j := 1$ to $m$ do
  begin
  { repeat $2^{m-j}$ times
  | begin
  |   D { $r := 0$; $n_0 := 0$; $n_1 := 1$;
  |     { Shift one block $B$ from $Mp1$ to $S$;
  |     { max := largest key of $B$;
  |     repeat $2^j - 1$ times
  |     begin
  |       { shift one block $B'$ from $Mpr$ to $S$;
  |       | $n_r := n_r + 1$;
  |       | if maximal key of $B' >$ max
  |       | $\wedge\ n_{\bar r} \neq 2^{j-1} \vee n_r = 2^{j-1}$ then
B |       | begin
  |     C |   $max :=$ maximal key of $B'$;
  |       |   $r := \bar r$
  |       | end;
  |       | sort in $S$;
  |       | shift the block containing
  |       {   the smaller keys to $M\bar{p}q$
  |     end;
  |   E { shift the last block
  |     {   out of $S$ to $M\bar{p}q$;
  |     { $q := \bar q$
  { end;
  F { $p := \bar p$
  end
end.
```

the sorting chip S and is involved in the subsequent sort-split operation. If the sorting is done in ascending order, then it is the block containing the larger keys.

Depending on the block of data read last the control unit CI determines from which input memory the next block of data is obtained. The control unit CO alternates between the two output memories to store the sorted sequences of blocks.

The algorithm starts with $2^m = 2N/n_S$ blocks of unsorted data in $M00$ and $M01$.

Whenever part D is executed S is empty and the whole set of data is located in $Mp0$ and $Mp1$. It consists of sequences of blocks of length 2^{j-1} (Only for $j = 1$ the data inside the blocks is unsorted). Part D initializes the merge of two neighboring block sequences, shifting the top most block from $Mp1$ into S and resetting the counters n_0 and n_1 which count for both block sequences the number of blocks that have been processed. The input memory to be read from next is denoted by r. This is opposite to the source of the maximal key that has been read so far.

In each of the $2^i - 1$ repetitions of part C one block of data is read, max and r are determined, S sorts two blocks, and the block containing the smaller keys is shifted to the output memory. Each execution of part E completes the merge of two sequences of blocks of length 2^{j-1} to one sequence of length 2^j. The next sorted sequence will be shifted into the other output memory.

Each execution of part B merges 2^{m-j+1} sorted sequences of blocks of length 2^{j-1} into 2^{m-j} sequences of length 2^j. Then part F switches input and output memory so that the next step of the iterative merge algorithm can start with doubled length of the block sequences.

The time complexity of TWB(S) is

$$T_{\mathrm{TWB(S)}} = ((m - 1) \cdot 2^m + 1) \cdot t_S = \Theta\left(\log \frac{N}{n_S} \cdot \frac{N}{n_S}\right) \cdot t_S .$$

Under the assumption $N > (n_S)^a$ for an $a > 1$ we get

$$T_{\mathrm{TWB(S)}} = \Theta\left(N \cdot \log N \cdot \frac{t_S}{n_S}\right).$$

$g_S := n_S/t_S$ refers to the throughput of S as it is achieved when used in TWB. This need not be the maximal possible throughput for S if it is used in a different environment. In the following g_S will be called the *speed of S*.

4.3. *Speedup of Chip-External Sorting Algorithms*

In this section we analyze the functional dependency between chip capacity and speed or throughput.

To simplify the following analysis we neglect logarithmic factors in the complexity of the sorting algorithms classifying the complexities by Φ (see Section 2.1).

Let S be a VLSI sorting algorithm with area requirement $A_S(n) = \Phi(n^r)$ and time complexity $T_S(n) = \Phi(n^p)$. Thus $A_S T_S^2 = \Phi(n^{r+2p})$ with $r + 2p \geq 2$ and $r \geq 1$.

The chip capacity C (the number of units of area that are available for the realization of the sorting algorithm) is treated as a variable. This reflects the possibility of spreading the realization of a sorting algorithm over several chips on a single board, or of using only part of the available chip area. It also reflects the development of the VLSI technology, which leads to a dramatic increase in chip capacity. Thus $C = A_S(n_S) = \Phi((n_S)^r)$, i.e., $C = C(n_S)$ is the chip capacity needed to realize S with problem size n_S.

THEOREM 4.1. *Let g_S be the speed of the sorting algorithm S with $A_S = \Phi(n^r)$ and $T_S = \Phi(n^p)$. Then*

$$g_S = \Phi(C^{(1-p)/r}).$$

Proof.

$$C = \Phi((n_S)^r) \text{ with } r \geq 1 \Rightarrow n_S = \Phi(C^{1/r}).$$

$$g_S = \frac{n_S}{T_S(n_S)} = \frac{\Phi(C^{1/r})}{\Phi((n_S)^p)} = \frac{\Phi(C^{1/r})}{\Phi((\Phi(C^{1/r}))^p)} = \Phi(C^{(1-p)/r}).$$

For AT^2 optimal sorting algorithms $r + 2p = 2 \Rightarrow (1 - p)/r = \frac{1}{2} \Rightarrow g_S = \Phi(C^{1/2})$.

Assume S is a suboptimal sorting algorithm: $A_S T_S^2 = \Phi(n^{2+x})$ with $x \geq 0$, thus $r + 2p = 2 + x \Rightarrow g_S = \Phi(C^{(r-x)/2r})$. The function $(r - x)/2r$ reaches its maximum for AT^2 optimal algorithms ($x = 0$). For $x = r$, $g_S = \Phi(n^0)$ (see Thompson, 1983; Shin *et al.*, 1983; Bilardi, 1984, for examples). For $x > r$ the speed decreases with increasing chip capacity. Thus such algorithms are not suitable for any VLSI technical realization.

THEOREM 4.2. *Let S be a sorting algorithm with $A = \Phi(n^r)$ and $T = \phi(n^p)$ and $r + 2p = 2 + x$ with $x \geq 0$. Then*

$$g_S = \Phi(C^{(1/2-x/2r)}).$$

Proof. $r + 2p = 2 + x \Rightarrow 1 - p = (r - x)/2 \Rightarrow (1 - p)/r = \frac{1}{2} - x/2r$. ∎

Replacing T by P and g_S by d_S in Theorems 4.1 and 4.2 results in

$$d_S = \Phi(C^{(1-p)/r}) = \Phi(C^{(1/2-x/2r)}).$$

So speed (throughput) is maximal for AT^2 (AP^2) optimal algorithms. An increase in the chip capacity by a factor of b results in the case of AT^2 (AP^2) optimal algorithms in a speedup $SO = \Phi(\sqrt{b})$. Suboptimal algorithms get a speedup $SS = SO/(\Phi(b^{x/2r}))$.

Thus, Theorem 4.2 is another justification for the measure of complexity AT^2 for VLSI sorting algorithms.

5. Summary

There are several different models of computation used on which to base evaluations of VLSI sorting algorithms and there are different measures of complexity. This paper revises complexity results under the linear model that have been gained under the constant model. This approach is due to expected technological development (see Mangir, 1983; Thompson and Raghavan, 1984; Vitanyi, 1984a, 1984b).

For the constant model we know that for medium sized keys there are AT^2 and AP^2 optimal sorting algorithms with T ranging from $\Omega(\log n)$ to $O(\sqrt{nk})$ and P ranging from $\Omega(1)$ to $O(\sqrt{nk})$ (Bilardi, 1984). The main results of asymptotic analysis of sorting algorithms under the linear model are that the lower bounds allow AT^2 optimal sorting algorithms only for $T = \Theta(\sqrt{nk})$ but allow AP^2 algorithms in the same range as under the constant model. Furthermore the sorting algorithms presented in this paper meet these lower bounds. This proves that these bounds cannot be improved for $k = \Theta(\log n)$. The building block for the realization of these sorting algorithms is a comparison exchange module that compares $r \times s$ bit matrices in time $T_C = \Theta(r + s)$ on an area $A_C = \Theta(r^2)$ (not including the storage area for the keys).

For problem sizes that exceed realistic chip capacities, chip-external sorting algorithms can be used. In this paper two different chip-external sorting algorithms (BBB(S) and TWB(S)) are presented. They are designed to be implemented on a single board. They use a sorting chip S to perform the sort-split operation on blocks of data BBB(S) and TWB(S) are systolic algorithms using local communication only so that their evaluation does not depend on whether the constant or the linear model is used. Furthermore it seems obvious that their design is technically feasible whenever the sorting chip S is technically feasible.

TWB has optimal asymptotic time complexity, so its existence proves that under the linear model external sorting can be done asymptotically as fast as under the constant model. The time complexity of TWB(S) is linearly dependent on the speed $g_S = n_S/t_S$. It is shown that the speed if looked at as a function of the chip capacity C is asymptotically maximal for AT^2 optimal sorting algorithms. Thus S should be a sorting algorithm similar to the M–M-sorter presented in this paper. A major disadvantage of TWB(S) is that it cannot exploit the maximal throughput $d_S = n_S/p_S$ of a systolic sorting algorithm S.

Therefore algorithm BBB(S) is introduced. The time complexity of BBB(S) is linearly dependent on d_S. It is shown that the throughput is maximal for AP^2 optimal algorithms. There is a wide range of such sorting algorithms including algorithms that can be realized in a way that is independent of the length of the keys. For example, BBB(S) with S being a highly parallel version of odd–even transposition sort has this kind of flexibility. A disadvantage of BBB(S) is that it is asymptotically slower than TWB(S).

Acknowledgment

The author thanks the referees whose suggestions have led to significant improvement of this paper.

References

Bilardi, G. (1984), "The Area–Time Complexity of Sorting," Ph.D. thesis, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Bonuccelli, M. A., Lodi, E., and Pagli, L. (1984), External sorting in VLSI, *IEEE Trans. Comput.* **C-33,** No. 10, 931–934.

Chazelle, B. M., and Monier, L. M. (1981), A model of computation for VLSI with related complexity results, *in* "Proceedings, 13th Symposium on Theory of Computing," pp. 318–325.

Duris, P., Thompson, C. D., Sykora, O., and Vrto, I. (1985), Tight chip area bounds for sorting, *Comput. Artificial Intelligence* (Czechoslovakia) **4,** No. 6, 535–544.

Ja'Ja, J., and Owens, R. M. (1985), Parallel sorting with serial memories, *IEEE Trans. Comput.* **C-34,** No. 4, 379–383.

Kim, W., Gajsky, D., and Kuck, D. J. (1984), A parallel pipelined relational query processor, *ACM Trans. Database Sys.* **9,** No. 2, 214–242.

Knuth, D. E. (1973), "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, MA.

Kumar, M., and Hirschberg, D. S. (1983), An efficient implementation of Batcher's odd–even merge algorithm and its application in parallel sorting schemes, *IEEE Trans. Comput.* **C-32,** 254–264.

Kung, H. T. "Why systolic architectures," *Computer Magazine* **15,** pp. 37–46, 1982.

Lang, H. W., Schimmler, M., Schmeck, H., and Schröder, H. (1985), Systolic sorting on a mesh-connected network, *IEEE Trans. Comput.* **C-34,** 652–658.

MANGIR, T. E. (1983), Impact and limitations of interconnect technology . . . , *in* "IEEE International Conference on Computer Design: VLSI in Computers," pp. 735–739.

MEHLHORN, K. (1984), "Datastructures and Algorithms. 1. Sorting and Searching," *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin.

MIRANKER, G. S., TANG, L., AND WONG, C. K. (1983), A zero time VLSI sorter, *IBM J. Res. Develop.*, 140–148.

NASSIMI, D., AND SAHNI, S. (1979), Bitonic sort on a mesh-connected parallel computer, *IEEE Trans. Comput.* **C-28,** 2–7.

PREPARATA, F. P., AND VUILLEMIN, J. (1981), The cube-connected-cycles: A versatile network for parallel computation, *CACM*, **24,** No. 5, 300–309.

SHIN, H., WELCH, A. J., AND MALEK, M. (1983), I/O overlapped sorting schemes for VLSI, *in* "IEEE International Conference on Computer Design: VLSI in Computers."

THOMPSON, C. D., AND KUNG, H. T. (1977), Sorting on a mesh-connected parallel computer, *CACM* **20,** 264–271.

THOMPSON, C. D. (1983), The VLSI complexity of sorting, *IEEE Trans. Comput.* **C-32,** 1171–1184.

THOMPSON, C. D., AND RAGHAVAN, P. (1984), On estimating the performance of VLSI circuits, *in* "Proceedings, Conference on Advanced Research in VLSI," MIT.

TODD, S. (1978), Algorithm and hardware for a merge sort using multiple processors, *IBM J. Res. Develop.* **22,** No. 5, 509–517.

VITANYI, P. M. B. (1984a), "Signal Propagation Delay, Wire Length Distribution and Efficiency of VLSI Circuits," Tech. Rep. CS-R 8412, Centre for Mathematics and Computer Science, Amsterdam.

VITANYI, P. M. B. (1984b), "Signal Propagation Delay and the Efficiency of VLSI Circuits," Tech. Rep. CS-R 8414, Centre for Mathematics and Computer Science, Amsterdam.