Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C. Mulder, W.P. Weijland

Verification of an algorithm for log-time sorting by square comparison

Computer Science/Department of Software Technology Report CS-R8729 June

69B71, 69D24, 69F12, 69F32

Verification of an algorithm for log-time sorting by square comparison

J.C. Mulder

Centre for Mathematics and Computer Science

P.O. Box 4079,1009 AB Amsterdam, The Netherlands

W.P. Weijland

Dept. of Computer Science, University of Amsterdam,

P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Abstract: In this paper a concurrent sorting algorithm called RANKSORT is presented, able to sort an input sequence of length n in log n time, using n^2 processors. The algorithm is formally specified as a *delay-insensitive* circuit. Then, a formal correctness proof is given, using bisimulation semantics in the language ACP of BERGSTRA & KLOP [BK].
The algorithm has area·time^2 = $O(n^2 \log^4 n)$ complexity which is slightly suboptimal with respect to the lower bound of $AT^2 = \Omega(n^2 \log n)$ (see: THOMPSON [TH]).

Key words and phrases: concurrency, process algebra, sorting, correctness proof, rank sort, orthogonal tree network, asynchronous cooperation, delay-insensitive.
1985 Mathematics subject classification:
68Q35, 68Q60, 68Q10, 68Q55.
1982 CR categories: B.7.1, D.2.4, F.1.2, F.3.2.

Note: Partial support received from the European Communities under ESPRIT contract no. 432, An Integrated Formal approach to Industrial Software Development (Meteor).
This report will be submitted for publication elsewhere.

Report CS-R8729
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. INTRODUCTION

Many authors have studied the concurrency aspects of sorting, and indeed the n-time ***bubblesort*** algorithm (using n processors) is rather thoroughly analysed already (e.g. see: HENNESSY [HEN] and KOSSEN & WEIJLAND [KW]). However, *bubblesort* is not the most efficient sorting algorithm in sequential programming, since it is n^2-time and for instance *heapsort* and *mergesort* are $n\log n$-time sorting algorithms. So, the natural question arises whether it would be possible to design an algorithm using even less than n-time.

In this paper we discuss a concurrent algorithm, capable of sorting n numbers in O(log n) time. This algorithm is based on the idea of *square comparison*: putting all numbers to be sorted in a square matrix, all comparisons can be made in O(1) time, using n^2 processors (one for each cell of the matrix). Then, the algorithm only needs to evaluate the result of this operation.

The algorithm presented here, which is called RANKSORT, is not the only concurrent time-efficient sorting algorithm. Several *sub n*-time algorithms have been developed by others (see: THOMPSON [TH]). For instance algorithms were presented of time-complexity $\sqrt{n}$, $\log^3 n$, $\log^2 n$ and log n. Indeed, the square comparison algorithm presented here, appeared in [TH] as well. Its network has been given various names, like *mesh of trees* or *orthogonal tree network.*

In this paper we will show how a log n-sorter can be constructed. Moreover we will present a formal specification of the algorithm and prove it correct using bisimulation semantics with asynchronous cooperation. The formal language, used in this paper, is called ACP, i.e: *Algebra of Communicating Processes* [BK]. It turns out that in this language the construction of the sorting machine is *delay-insensitive*, which says that any temporary cut in one of the wires of the machine, may delay its computation but cannot endanger its correct behaviour.

At this place we want to thank Niek van Diepen (Centre for Mathematics and Computer Science) and Karl Meinke (University of Leeds) for their contributions to this paper. Moreover we thank Jaap Jan de Bruin for his assistance concerning the illustrations which were made on an Apple Macintosh. Finally, we thank Jos Baeten for his remarks on the early drafts of this paper.

2. THE ALGEBRA OF COMMUNICATING PROCESSES

The axiomatic framework in which we present this document is ACP_τ, the Algebra of Communicating Processes with silent steps, as described in BERGSTRA & KLOP [BK]. In this section, we give a brief review of ACP_τ.

Process algebra starts from a finite collection A of given objects, called atomic actions, atoms or steps. These actions are taken to be indivisible, usually have no duration and form the basic building blocks of our systems. The first two compositional operators we consider are $\cdot$, denoting sequential composition, and + for alternative composition. If x and y are two processes, then $x \cdot y$ is the process that starts the execution of y after the completion of x, and $x+y$ is the process that chooses either x or y and executes the chosen process. Each time a choice is made, we choose from a set of alternatives. We do not specify whether the choice is made by the process itself, or by the environment. Axioms A1-5 in table 1 below give the laws that + and $\cdot$ obey. We leave out $\cdot$ and brackets as in regular algebra, so $xy + z$ means $(x \cdot y) + z$.

$x + y = y + x$	A1	$x\tau = x$	T1
$x + (y + z) = (x + y) + z$	A2	$\tau x + x = \tau x$	T2
$x + x = x$	A3	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$(x + y)z = xz + yz$	A4		
$(xy)z = x(yz)$	A5		
$x + \delta = x$	A6		
$\delta x = \delta$	A7		
$a \mid b = b \mid a$	C1		
$(a \mid b) \mid c = a \mid (b \mid c)$	C2		
$\delta \mid a = \delta$	C3		
$x \| y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + x \mid y$	CM1		
$a \mathbin{\lfloor\!\lfloor} x = ax$	CM2	$\tau \mathbin{\lfloor\!\lfloor} x = \tau x$	TM1
$ax \mathbin{\lfloor\!\lfloor} y = a(x \| y)$	CM3	$\tau x \mathbin{\lfloor\!\lfloor} y = \tau(x \| y)$	TM2
$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$	CM4	$\tau \mid x = \delta$	TC1
$ax \mid b = (a \mid b)x$	CM5	$x \mid \tau = \delta$	TC2
$a \mid bx = (a \mid b)x$	CM6	$\tau x \mid y = x \mid y$	TC3
$ax \mid by = (a \mid b)(x \| y)$	CM7	$x \mid \tau y = x \mid y$	TC4
$(x + y) \mid z = x \mid z + y \mid z$	CM8		
$x \mid (y + z) = x \mid y + x \mid z$	CM9	$\partial_H(\tau) = \tau$	DT
		$\tau_I(\tau) = \tau$	TI1
$\partial_H(a) = a \quad \text{if } a \notin H$	D1	$\tau_I(a) = a \quad \text{if } a \notin I$	TI2
$\partial_H(a) = \delta \quad \text{if } a \in H$	D2	$\tau_I(a) = \tau \quad \text{if } a \in I$	TI3
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI4
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	D4	$\tau_I(xy) = \tau_I(x) \cdot \tau_I(y)$	TI5

table 1. ACP_τ.

On intuitive grounds $x(y + z)$ and $xy + xz$ present different mechanisms (the moment of choice is

different), and therefore, an axiom $x(y + z) = xy + xz$ is not included.
We have a special constant δ denoting deadlock, the acknowledgement of a process that it cannot do anything anymore, the absence of an alternative. Axioms A6,7 give the laws for δ.
Next, we have the parallel composition operator $\|$, called merge. The merge of processes x and y will interleave the actions of x and y, except for the communication actions. In $x\|y$, we can either do a step from x, or a step from y, or x and y both synchronously perform an action, which together make up a new action, the communication action. This trichotomy is expressed in axiom CM1. Here, we use two auxiliary operators $\lfloor\!\lfloor$ (left-merge) and $|$ (communication merge). Thus, $x \lfloor\!\lfloor y$ is $x\|y$, but with the restriction that the first step comes from x, and $x|y$ is $x\|y$ with a communication step as the first step. Axioms CM2-9 give the laws for $\lfloor\!\lfloor$ and $|$. On atomic actions, we assume the communication function given, obeying laws C1-3. Finally, we have on the left-hand side of table 1 the laws for the encapsulation operator ∂_H. Here H is a set of atoms, and ∂_H blocks actions from H, renames them into δ. The operator ∂_H can be used to encapsulate a process, i.e. to block communications with the environment.
The right-hand side of table 1 is devoted to laws for Milner's silent step τ (see MILNER [MI]). Laws T1-3 are Milner's τ-laws, and TM1,2 and TC1-4 describe the interaction of τ and merge. Finally, τ_I is the abstraction operator, that renames atoms from I into τ.
In table 1 we have $a,b,c \in A_\delta$ (i.e. $A \cup \{\delta\}$), x,y,z are arbitrary processes, and $H,I \subseteq A$.

Definition The set of **basic terms, BT**, is inductively defined as follows:

i. $\tau, \delta \in BT$
ii. if $t \in BT$ and $a \in A$, then $at \in BT$
iii. if $t \in BT$, then $\tau{\cdot}t \in BT$
iv. if $t, s \in BT$, then $t{+}s \in BT$.

elimination theorem (BERGSTRA & KLOP [BK]) Let t be a closed term over ACP_τ. Then there is a basic term s such that $ACP_\tau \vdash t{=}s$.

The elimination theorem allows us to use induction in proofs. The set of closed terms modulo derivability (the initial algebra) forms a model for ACP_τ. However, most processes encountered in practice cannot be represented by a closed term, but will be specified recursively. Therefore, most models of process algebra also contain infinite processes, that can be recursively specified. First, we develop some terminology.

Definition i) Let t be a term over ACP_τ, and x a variable in t. Suppose that the abstraction operator τ_I does not occur in t. Then we say that an occurrence of x in t is **guarded** if t has a subterm of the form $a{\cdot}s$, with $a \in A_\delta$ (so $a \neq \tau$!) and this x occurs in s. (I.e. each variable is 'preceded' by an atom.)
ii) A **recursive specification** over ACP_τ is a set of equations $\{x = t_x : x \in X\}$, with X a set

of variables, and t_x a term over ACP_τ and variables X (for each $x \in X$). No other variables may occur in t_x.
iii) A recursive specification $\{x = t_x : x \in X\}$ is **guarded** if no t_x contains an abstraction operator τ_I, and each occurrence of a variable in each t_x is guarded.

Notes: i) The constant τ cannot be a guard, since the presence of a τ does not lead to unique solutions: to give an example, the equation $x = \tau x$ has each process starting with a τ as a solution.
ii) A definition of guardedness involving τ_I is very complicated, and therefore, we do not give such a definition here. The definition above suffices for our purposes.

Definition: On ACP_τ, we can define a **projection operator** π_n, that cuts off a process after n atomic steps are executed, by the axioms in table 2 ($n \geq 1$, $a \in A_\delta$, x,y are arbitrary processes).

$\pi_n(a) = a$	$\pi_n(\tau) = \tau$
$\pi_1(ax) = a$	$\pi_n(\tau x) = \tau \cdot \pi_n(x)$
$\pi_{n+1}(ax) = a \cdot \pi_n(x)$	
$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	

table 2. Projection.

Remarks: Because of the τ-laws, we must have that executing a τ does not increase depth. A process p is **finite** if it is equal to a closed term; otherwise p is **infinite**. Note that if p is finite, there is an n such that $\pi_n(p) = p$.

projection theorem (BAETEN, BERGSTRA & KLOP [BBK2]) If the set of processes P forms a solution for a guarded recursive specification E, then $\pi_n(p)$ is equal to some closed ACP_τ-term for each $p \in P$ and $n \geq 1$, and this term does not depend on the particular solution P.

The projection theorem leads us to formulate the following two principles, which together imply that each guarded recursive specification has a unique solution (determined by its finite projections).
The **Recursive Definition Principle (RDP)** is the assumption that each guarded recursive specification has at least one solution, and the **Recursive Specification Principle (RSP)** is the assumption that each guarded recursive specification has at most one solution. In this paper, we assume RDP and RSP to be valid.

To give an example, if p is a solution of the guarded recursive specification $\{x = a \cdot x\}$, we find $\pi_n(p) = a^n$ for all $n \geq 1$, so we can put $p = a^\omega$. For more information, see [BBK1].
Abusing language, we also use the variables in a guarded recursive specification for the process that is its unique solution.

In BAETEN, BERGSTRA & KLOP [BBK1], a model is presented for ACP_τ, consisting of rooted, directed multigraphs, with edges labeled by elements of $A \cup \{\delta, \tau\}$, modulo a congruence relation called rooted $\tau\delta$-bisimulation (comparable to Milner's observational congruence, see [MI]). In this model all axioms presented in this paper hold, and also principles RDP and RSP hold.

The axioms of **Standard Concurrency** (displayed in table 4, with $a \in A \cup \{\delta\}$) will also be used in the sequel. A proof that they hold for all closed terms can be found in BERGSTRA & KLOP [BK].

$$(x \mathbin{\|\!\_} y) \mathbin{\|\!\_} z = x \mathbin{\|\!\_} (y \| z)$$
$$(x \mid ay) \mathbin{\|\!\_} z = x \mid (ay \mathbin{\|\!\_} z)$$
$$x \mid y = y \mid x$$
$$x \| y = y \| x$$
$$x \mid (y \mid z) = (x \mid y) \mid z$$
$$x \| (y \| z) = (x \| y) \| z$$

table 4. Standard concurrency.

As one can easily see, encapsulation and abstraction cannot in general be distributed over $\|$, since in a merge processes may do a communication step and thus it is of great importance which comes first, the encapsulation (or abstraction) operator or the merge. Next, *conditional axioms* will be presented to state conditions for distributing τ_I and ∂_H over $\|$.

$\alpha(\delta) = \varnothing$	AB1
$\alpha(\tau) = \varnothing$	AB2
$\alpha(ax) = \{a\} \cup \alpha(x)$	AB3
$\alpha(\tau x) = \alpha(x)$	AB4
$\alpha(x + y) = \alpha(x) \cup \alpha(y)$	AB5
$\alpha(x) = \cup_{n \geq 1} \alpha(\pi_n(x))$	AB6
$\alpha(\tau_I(x)) = \alpha(x) - I$	AB7

table 5. Alphabet.

Definition: The **alphabet** of a process is the set of atomic actions that it can perform. So an alphabet is a subset of A. In order to define the alphabet function α on processes, we have the axioms in table 5 (for $a \in A$, x,y are arbitrary processes; see BAETEN, BERGSTRA & KLOP [BBK2]).

Note that $\alpha(\delta) = \alpha(\tau) = \varnothing$ is necessary by axioms A6 and T1. The axioms AB6 and AB7 can be proved from AB1-5 for *closed* terms, but are needed here to define the alphabet on general processes.
Now we can formulate the conditional axioms as is done in table 6.

$\alpha(x) \mid (\alpha(y) \cap H) \subseteq H$	$\Rightarrow$	$\partial_H(x \| y) = \partial_H(x \| \partial_H(y))$	CA1
$\alpha(x) \mid (\alpha(y) \cap I) = \varnothing$	$\Rightarrow$	$\tau_I(x \| y) = \tau_I(x \| \tau_I(y))$	CA2
$\alpha(x) \cap H = \varnothing$	$\Rightarrow$	$\partial_H(x) = x$	CA3
$\alpha(x) \cap I = \varnothing$	$\Rightarrow$	$\tau_I(x) = x$	CA4
$H = J \cup K$	$\Rightarrow$	$\partial_H(x) = \partial_J \circ \partial_K(x)$	CA5
$I = J \cup K$	$\Rightarrow$	$\tau_I(x) = \tau_J \circ \tau_K(x)$	CA6
$H \cap I = \varnothing$	$\Rightarrow$	$\tau_I \circ \partial_H(x) = \partial_H \circ \tau_I(x)$	CA7

table 6. Conditional axioms.

In [BBK2] the axioms CA1-7 have been proved to hold for all closed ACP_τ-terms. We will assume that they hold for all processes.

3. SORTING BY SQUARE COMPARISON

Suppose we have a sequence $a_0, a_1, a_2, \ldots, a_{n-1}$ of distinct numbers, for some n>0, and consider the problem of computing a non-decreasing permutation of this sequence. Note that, in fact, we can start from an arbitrary set of symbols and any linear ordering >, defined on this finite set. Now restrict this ordering to the **n** elements that are considered, then we obtain a finite ordering, which can be represented in a matrix as pictured in figure 1 and 2.

In every cell (i,j) of the matrix in figure 1 we write **1** if $a_i > a_j$, and **0** otherwise. Note that now the matrix has only **0**'s on its diagonal. Moreover it is antisymmetric, i.e: if $i \neq j$ we have **1** in (i,j) iff we have **0** in (j,i). So in fact we only need one 'half' of the matrix.
The idea of square comparison now simply reads as follows: suppose we have a finite sequence of numbers to be sorted, then all the information relevant to the ordering problem can be computed in unit time, starting from the matrix above. Indeed, in one blow all n^2 individual cells (i,j) can do one

>=1 ≤=0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
a_0								
a_1								
a_2								
a_3								
a_4								
a_5								
a_6								
a_7								

figure 1. Defining ≥ by laying out a full matrix.

comparison (between a_i and a_j), and next all information about > is available. Note that we can set up this matrix in O(log n) time, starting from *n* processors containing the values to be sorted. Thus all ordering information can be computed in O(log n) time.

After O(log n) time we have computed a matrix which is full of 0's and 1's. Note, that on the i^{th} row, we have a 1 for every a_j which is smaller than a_i. Hence the number of 1's in the i^{th} row is precisely the number of elements a_j out of $a_0,a_1,a_2,...,a_{n-1}$, satisfying $a_j<a_i$. However, the number of elements $<a_i$ is exactly the index of a_i in the *sorted* sequence, i.e. represents the *place* of the number a_i in the sorted array.
Finally note that the number of 1's can simply be found, by computing the sum of all matrix values on the row considered. This computation can be done in O(log n) time, since we can repeatedly add pairs of numbers concurrently, until there is only one single value left. Thus we conclude that, for all input values, we can compute the 'sorted index' in O(log n) time.

In fact we have computed a *permutation* of the index values <0,1,2,...,n-1>. From this permutation one can compute the sorted array in O(1) time, since all cells consider the computed index value, as an adress to send the value to, they actually contain. Having enough wires to interconnect all cells, this can be done in one single computation step (note: by putting the processors in a tree configuration once again, we can do this in O(log *n*) time, with many wires less).
So, indeed, we can sort a sequence of numbers in log *n* time using n^2 processors.

An example of this square comparison method is presented in the following figure.

> ≥	2	7	1	-5	11	2	3	8		
2	0	0	1	1	0	0	0	0	+ $\longrightarrow$	2
7	1	0	1	1	0	1	1	0	+ $\longrightarrow$	5
1	0	0	0	1	0	0	0	0	+ $\longrightarrow$	1
-5	0	0	0	0	0	0	0	0	+ $\longrightarrow$	0
11	1	1	1	1	0	1	1	1	+ $\longrightarrow$	7
2	1	0	1	1	0	0	0	0	+ $\longrightarrow$	3
3	1	0	1	1	0	1	0	0	+ $\longrightarrow$	4
8	1	1	1	1	0	1	1	0	+ $\longrightarrow$	6

figure 2. An example of the square comparison method on the sequence 2,7,1,-5,11,2,3,8.

Here we have a small problem: suppose two numbers in the array are equal (the numbers are no longer distinct), then the matrix values, computed in figure 1, would be equal for both numbers. Thus the problem is that the computed array of index values no longer is a permutation of <0,1,2,...,n-1>, since some of the computed indices might be equal.
In figure 2, this problem is solved by slightly changing the former procedure. Now, the 'lower' cells, i.e. the cells under and on the main diagonal of the matrix, do not compare two values via '>' but via '≥'. Now it turns out that the computed indices indeed are a permutation of <0,1,2,...,n-1> and that the 'original order' of equal numbers is preserved in the sorted array.

In figure 2 the sequence 2,7,1,-5,11,2,3,8 is considered. Note that here, the computed index values <2,5,1,0,7,3,4,6> indeed form a permutation of <0,1,2,3,4,5,6,7>. To be specific: note that the number 2 has two different computed indices (namely 2 and 3); without the adaptation mentioned above, both occurrences of the value 2 would yield the index value 3.

The sorting machine considered in this paper is pictured in figure 3, for **n**=4.
Note that on the upper side we have **n** trees, one for every input value. Each input value is broadcast to **n** leaves in a row of the matrix, which is in the middle part of the machine. Then, the cells on the main diagonal will send the value received from the upper tree downwards to the

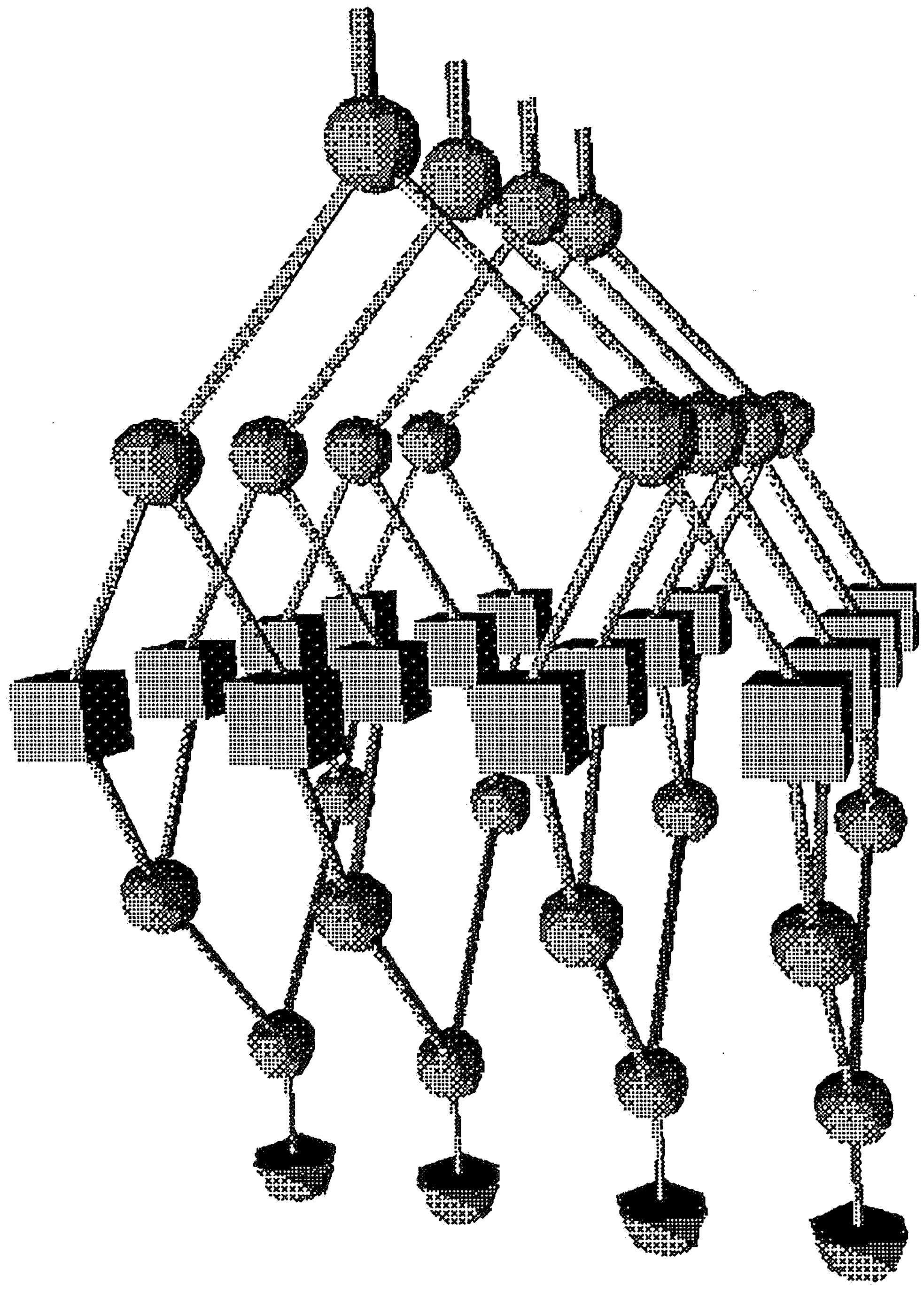

figure 3. A 'mesh of trees'; the circuit configuration of RANKSORT.

bottom of the connected lower tree; this value is broadcast upwards again to **n** matrix cells, belonging to a column of the matrix. So, every matrix cell now contains two values, precisely in the way as in figure 1. Then the $\mathbf{n}^2$ comparisons are made and each cell sends a **1** or a **0** to its upper tree. In every node the addition of two input values is computed, and the result is sent upwards again. Finally, the computed index permutation can be read from the roots of the upper trees.

4. A FORMAL SPECIFICATION OF THE SORTING MACHINE

In this section we will present a formal specification of RANKSORT, using the language ACP as described in section 2. First, we have to name the channels of the machine (figures 3-5) in order to be able to give a precise definition of the behaviour of the individual cells.
For reasons of simplicity, in the following we will assume $\mathbf{n} = 2^k$ for some given $k>0$, **n** being the length of the array to be sorted.

In figure 4 we present the names of the processes, corresponding to the vertices in the trees and the cells of the matrix. The upper trees are called U_i ($0 \leq i < n$) and the cells in these trees are numbered U_{ij} ($0<j<n$). Likewise, the lower trees are called L_j, with cells L_{ij} ($0<i<n$), and the matrix cells are called M_{ij} ($0 \leq i,j < n$). The bottom cells will be called B_j ($0 \leq j < n$).
Note that for all i, U_i has depth ${}^2\log \mathbf{n} = k$ and has $2^k-1 = \mathbf{n}-1$ cells. Further, the cells and channels in the trees are numbered 'left first / breadth first', as one can see in the figures 4 and 5.

Now, let us present a more detailed description of the behaviour of the individual processes.

- A cell $U_{i,j}$ will receive a value from its upper neighbour. Next, it will send this value to both of its lower neighbours, and from both of them it will receive another value in return. Since both lower neighbours are independent processes, these send and receive actions are fully interleaved. Finally, having received two values from below, $U_{i,j}$ will send its sum up again.

- A matrix cell $M_{i,j}$ in the middle of the sorter will first receive a value from the upper neighbour. Then, if it is a *diagonal* cell, it will send this value downwards to its lower neighbour. For sake of simplicity, we will make *non-diagonal* cells send a value **nil** downwards as well. Next, the cell will receive a new value from below, and send up a **0** or a **1**, depending on its position (see figure 2) and the two input values.

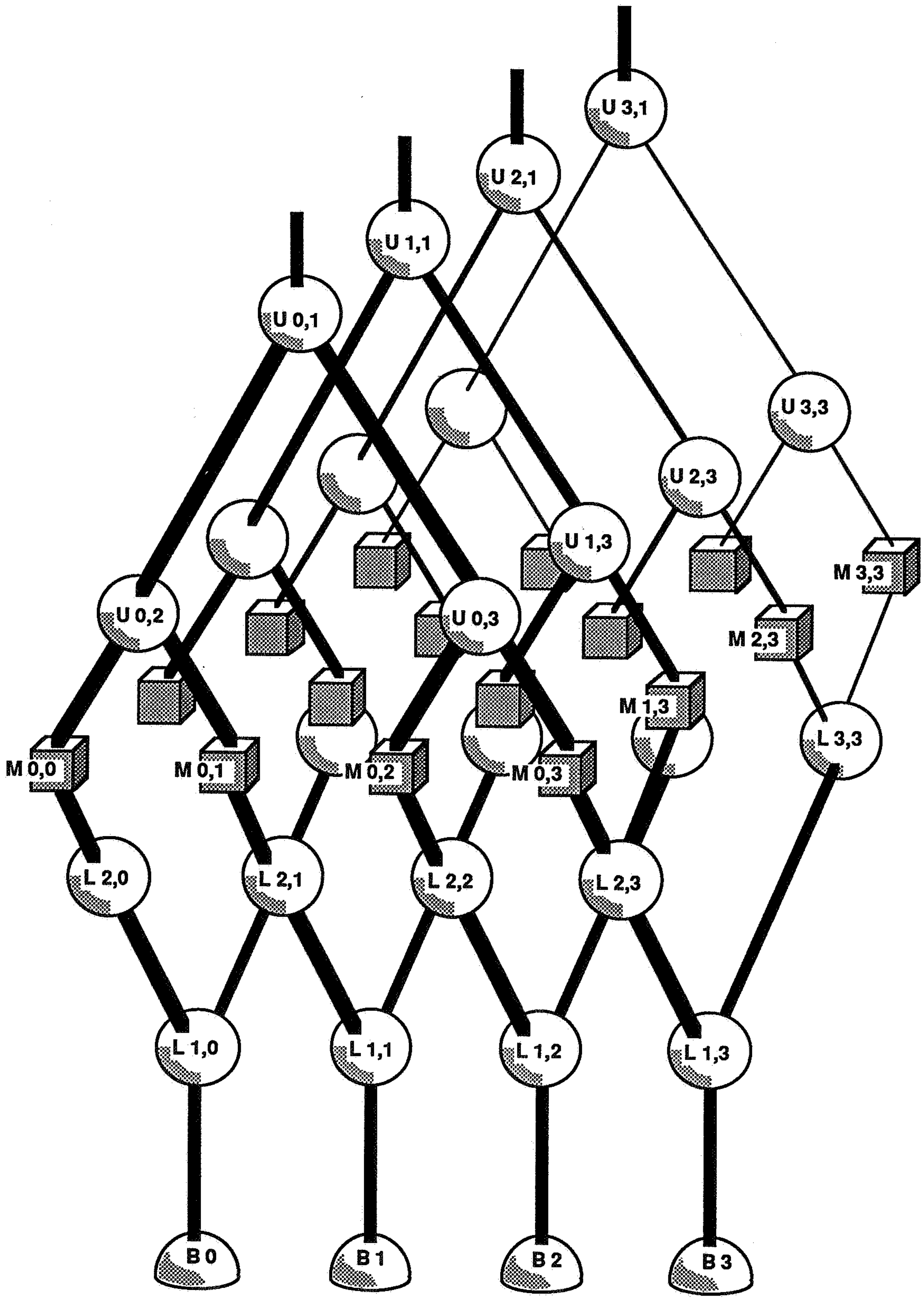

figure 4. The names of the individual cells in the sorter.

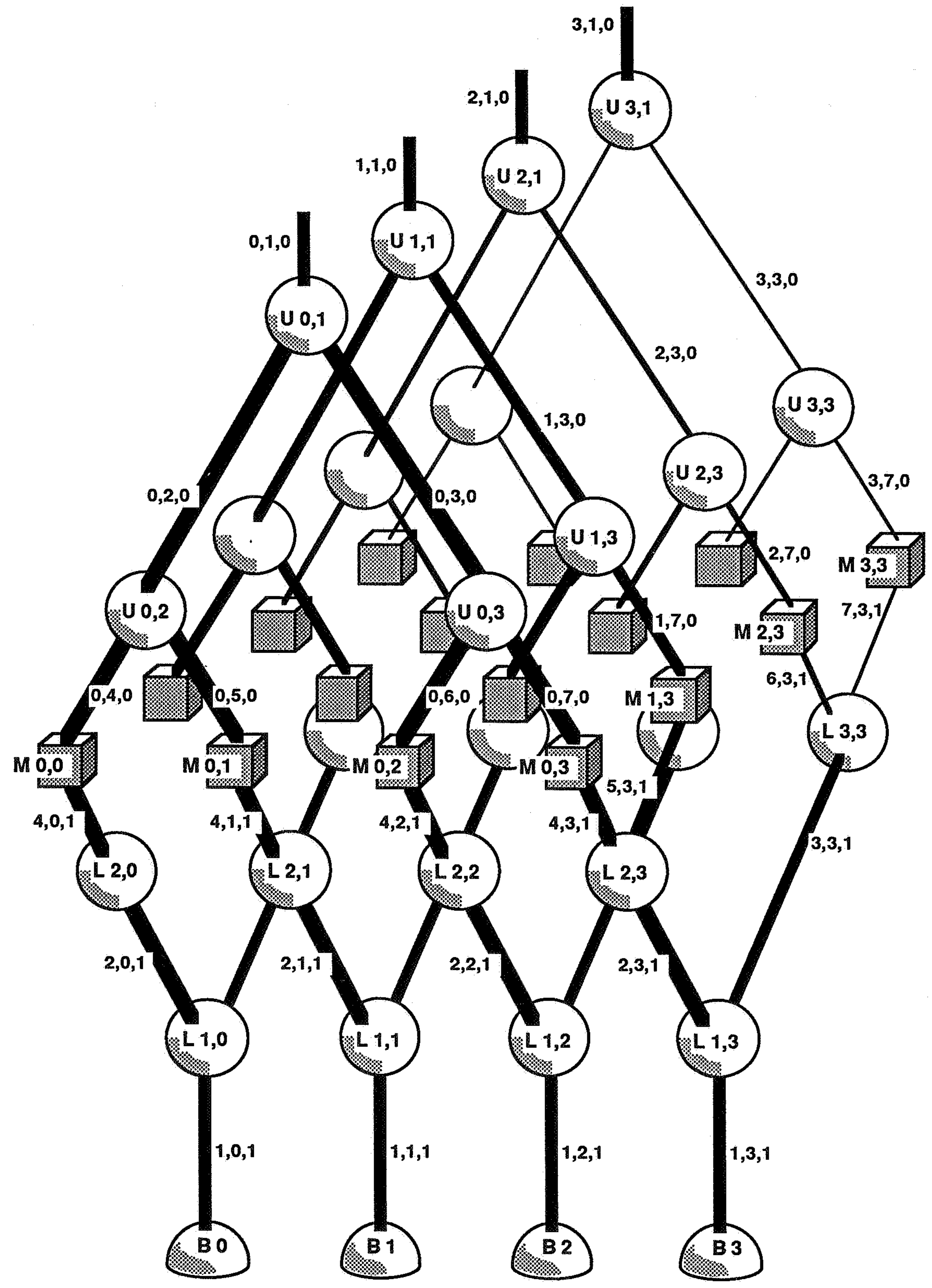

figure 5. The channel numbers are in 'left first / breadth first' order.

- A cell $L_{i,j}$ from one of the lower trees, will first receive two values from above (in any order). Note that in any lower tree only one leaf, the one in the diagonal of the matrix, will send down a number. The others will only send down **nil**. Now, if one of the values received from above is not **nil**, $L_{i,j}$ will send this value to its lower neighbour. Otherwise it will send down just **nil**. Next a value is received from below and 'broadcast' upwards, just like in $U_{i,j}$, by sending it to its upper neighbours.

- Finally, a cell B_j from the bottom of the machine, acts as a reflector: it will receive a value from it upper neighbour, and simply return it. Note that B_j will actually receive the number (≠**nil**) which is sent down by $M_{j,j}$.

Now we will translate these informal descriptions into the algebraical specification language ACP. To do this we need the definitions of the following functions.

Definition We need a function **diag** to specify the value that will actually be sent down by $M_{i,j}$ after having received *d*:

diag(i,i,*d*) = *d*

diag(i,j,*d*) = **nil** (i≠j).

Definition We also need a function **comp** to express what boolean value, **0** or **1**, will be sent up by $M_{i,j}$ again, after having received *d* and *e*. So in **comp** we actually use the square comparison method (see figure 2):

```
comp(i,j,d,e) = if i≤j then if d≥e then 1 else 0 fi
                            else if d>e then 1 else 0 fi
                fi;
```

Definition Finally we need a kind of *exclusive or* on strings of symbols, to express what value is sent down bij $L_{i,j}$ after having received two values:

xor(d,**nil**) = **xor**(**nil**,d) = d

xor(d,e) = **xor**(**nil**,**nil**) = **nil** (d,e∈D).

Inductively, we will define **xor** on arbitrary strings of length $n=2^k$:

$$\mathbf{xor}(d_1,d_2,\ldots,d_{2^k}) = \mathbf{xor}(\mathbf{xor}(d_1,\ldots,d_{2^{k-1}}),\mathbf{xor}(d_{2^{k-1}+1},\ldots,d_{2^k})) \qquad (d_i \in D \cup \{\mathbf{nil}\}).$$

Note, that if exactly one value out of $\{d_1,\ldots,d_n\}$, d_i say, is not equal to **nil**, then **xor**$(d_1,\ldots,d_n) = d_i$. So **xor** 'picks' out the unique value ≠**nil**, assuming this unique value exists. This more general definition will be needed later, to describe the specific behaviour of the lower trees, since all of its leaves will send down **nil** except for the leaf on the diagonal of the matrix.

Now we will turn to the formal specification of the cells. In this specification we have atomic actions $r_{i,j,m}(d)$ and $s_{i,j,m}(d)$ for *receiving* and *sending* a datum d to and from the channel [i,j,m]. Note that receive and send actions do not have a fixed 'direction' in the channel.
We assume D to be a (finite) set of numbers. All (bound) variables are written in italics.

A formal specification of the cells in the sorter

$$U_{i,j} = \sum_{d\in D} r_{i,j,0}(d) \cdot [\{s_{i,2j,0}(d) \cdot \sum_{n\in N} r_{i,2j,0}(n)\} \parallel \{s_{i,2j+1,0}(d) \cdot \sum_{m\in N} r_{i,2j+1,0}(m)\}] \cdot s_{i,j,0}(n+m)$$

$$M_{i,j} = \sum_{d\in D} r_{i,j+n,0}(d) \cdot s_{i+n,j,1}(\mathrm{diag}(i,j,d)) \cdot \sum_{e\in D} r_{i+n,j,1}(e) \cdot s_{i,j+n,0}(\mathrm{comp}(i,j,d,e))$$

$$L_{i,j} = [\sum_{d\in D\cup\{nil\}} r_{2i,j,1}(d) \parallel \sum_{e\in D\cup\{nil\}} r_{2i+1,j,1}(e)] \cdot s_{i,j,1}(\mathrm{xor}(d,e)) \cdot \\ \cdot \sum_{f\in D} r_{i,j,1}(f) \cdot [s_{2i,j,1}(f) \parallel s_{2i+1,j,1}(f)]$$

$$B_j = \sum_{d\in D} r_{1,j,1}(d) \cdot s_{1,j,1}(d)$$

As a shorthand, the scope rules of $\sum$ are violated in the first equation. Writing out $\parallel$ using the axioms CM1-4 of table 1, $U_{i,j}$ can easily be specified correctly (see also [KW] and [WE]).
It takes some effort to check all the indices, corresponding to the names of the channels. However, making use of the regular configuration of the circuit, and comparing the specification with the figures 3 and 4, one can find out that they are presented correctly here. Furthermore, in the next section we will concentrate on a formal proof of correctness of the sorter, and from any such proof it follows immediately that the channel numbers in the specification above are correct.

Now we present the final specification of the sorting machine as a whole by simply interconnecting all cells.

A specification of RANKSORT

$$\mathrm{RANKSORT}(n) = \mathop{\parallel}_{i,j<n} \{ U_{i,j} \parallel M_{i,j} \parallel L_{i,j} \} \parallel \mathop{\parallel}_{i<n} \{ B_i \}$$

So this is the specification, in detail, of RANKSORT. Indeed, it is not clear at all, why such a complex machine would be a sorting machine. In the next section we will hide almost all of the internal actions of the machine (only actions via channels [i,1,0] are of interest to the user). Then we will prove the result to be a sorting machine, and hence prove RANKSORT correct.

5. FORMULATING A CORRECTNESS THEOREM

In this section we will present a formal theorem of correctness for RANKSORT, i.e: abstracting from internal actions, we will state that RANKSORT indeed behaves like a sorting machine. To do this, we first have to specify what actually *is* a sorting machine.

Definition In the following we define the *sorted indices* of a given sequence of numbers. Suppose $a = a_0,a_1,a_2,\dots,a_{n-1}$ is such a sequence of numbers, then we have:

(i) $<p_0(a),\dots,p_{n-1}(a)> \in \mathrm{PERM}(<0,\dots,n-1>)$,

(ii) $a_{pi-1(a)} \leq a_{pi(a)}$ $\quad (0<i<n)$,

(iii) $a_{pi-1(a)} = a_{pi(a)} \Rightarrow p_{i-1}(a) < p_i(a)$ $\quad (0<i<n)$.

Because of part (iii) of the definition the permutation $p_i(a)_{0\leq i<n}$ satisfying all three conditions, is uniquely determined.

Note that from the sorted indices $p_i(a)_{0\leq i<n}$ we can immediately compute the sorted sequence itself: assume we have n processors $P_0,\dots,P_{n-1}$, containing the values $p_0(a),\dots, p_{n-1}(a)$ and $a_0,\dots,a_{n-1}$ respectively, and suppose all processors are interconnected by channels (wires) then in one step every process P_i can send the number a_i to the 'adress' given by $p_i(a)$, i.e: to $P_{pi(a)}$.

Next we will formulate a crucial proposition, stating a criterion for correctness of the square comparison method. A proof of this proposition is omitted.

proposition

For all sequences $a = a_0,\dots,a_{n-1}$ and all $0\leq i<n$ we have:

$$\sum_{j=0}^{n-1} \mathrm{comp}(i,j,a_i,a_j) = p_i(a).$$

Clearly, the proposition states that the square comparison method provides us with the sorted indices of the input sequence. Using this proposition we will be able to prove RANKSORT correct, in the sense that RANKSORT turns out to calculate precisely $\sum_{0\leq j<n} \mathrm{comp}(i,j,a_i,a_j)$ for all sequences $a_0,\dots,a_{n-1}$.

Definition Suppose a process SORT(n) satisfies the equation

$$\text{SORT(n)} = \Big(\mathop{\|}_{0\le i<n} [\, \Sigma_{xi}\, r_{i,1,0}(x_i)\,] \Big) \cdot \Big(\mathop{\|}_{0\le i<n} s_{i,1,0}(p_i(x)) \Big) ,$$

and $x = x_0,...x_{n-1}$, then SORT(n) is called a *sorting machine* of size n.

So, by definition, we agree that any machine, that receives a sequence of **n** numbers, and consequently outputs all sorted indices of this input sequence, may be called a sorting machine. Now we will return to **RANKSORT** again.
Let D be a (finite) set of numbers. Suppose $n=2^k$, $k\geq0$.

The *communication function* | is defined by

$(r_{i,j,m}(d) \,|\, s_{i,j,m}(d)) = (s_{i,j,m}(d) \,|\, r_{i,j,m}(d)) = c_{i,j,m}(d)$ for all i,j,m,
all other communication actions result in deadlock, δ.

The *encapsulation sets* J_n, K_n, H_n and E_n are defined by

$M_n = \{\, s_{i,j+n,0}(d),\, r_{i,j+n,0}(d) : d\in D\cup\{\mathbf{nil}\},\, i,j<n \,\} \cup$
$\{\, s_{i+n,j,1}(d),\, r_{i+n,j,1}(d) : d\in D\cup\{\mathbf{nil}\},\, i,j<n \,\}$
corresponding to all channels connected with the matrix cells $M_{i,j}$,
$B_n = \{\, s_{1,j,1}(d),\, r_{1,j,1}(d) : d\in D\cup\{\mathbf{nil}\},\, j<n \,\}$
corresponding to the channels connected with the bottom cells B_j ,
$H_n = \{\, s_{i,j,m}(d),\, r_{i,j,m}(d) : d\in D\cup\{\mathbf{nil}\};$ for all i,j,m , such that:
$(j,m)\neq(1,0)$ and $(i,m)\neq(1,1)$ and $i,j<n \,\}$
which is the set of all communicating actions, except for actions from M_n or B_n or the ones corresponding to the input/output channels [i,1,0] (i<n),
$E_n = H_n \cup M_n \cup B_n$.

Finally, the *abstraction set* I is defined by

$I = \{\, c_{i,j,m}(d) : d\in D\cup\{\mathbf{nil}\};$ for all i,j,m $\}$.

The definition of the communication function says, that receive and send actions only result in a communication $c_{i,j,m}(d)$ if they correspond to the same channel [i,j,m] and the same datum d. If not, a deadlock occurs; e.g: $(r_{2,7,0}(d) \,|\, s_{5,2,1}(d)) = (r_{i,j,m}(d_1) \,|\, s_{i,j,m}(d_2)) = (r_{i,j,m}(d) \,|\, r_{i,j,m}(d)) = \delta$.
The choice of the encapsulation sets M_n, B_n and H_n is quite standard: we want no single receive or send actions to happen without direct communication with their 'partner', since otherwise data would be sent to a channel but are never read from it. Except for the receive and send actions on the channels [i,1,0] ($0\leq i<n$): they are the input and output channels of the machine, and are ready for communication with the outside world. The encapsulation sets, M_n and B_n, are defined separately

from H_n, to simplify the proofs that will be presented later. At the end of the proof, however, we will encapsulate all actions from $E_n = H_n \cup M_n \cup B_n$.
The abstraction set I has no index **n** since it contains *all* communication actions $c_{i,j,m}(d)$. By renaming all actions from I into τ we can *hide* internal communication actions from the outside world.

Note that any user of **RANKSORT** will indeed not be interested in the internal communications of the machine, and only will observe the outside behaviour, i.e: $\tau_I \partial_{En}($ **RANKSORT(n)** $)$. Now a correctness theorem can easily be formulated as follows:

Theorem (correctness of RANKSORT)

For all $k \geq 0$ and $n=2^k$, we have

$$ACP_\tau \vdash \tau_I \partial_{En}(\textbf{RANKSORT(n)}) = \textbf{SORT(n)}$$

where **SORT(n)** is specified earlier.

This theorem states that $\tau_I \partial_{En}($ **RANKSORT(n)** $)$ indeed is a sorting machine in the sense of the definition of **SORT(n)**. The proof will be presented in the next section.

6. A FORMAL PROOF OF CORRECTNESS

In this section we will present the final proof of the correctness theorem. First we will simplify the problem by stating and proving two lemmas. Combining both of them we can easily find the proof we are looking for.
First we will formulate what we expect the ith upper tree $U_{i,1} \| \ldots \| U_{i,n-1}$ to behave like. This is done in lemma 1 below.

Lemma 1

Assume $n=2^k$, for some given $k>0$. Then in the theory ACP_τ we have

$$\tau_I \partial_{H_n}(U_{i,1} \| \ldots \| U_{i,n-1}) =$$

$$= \sum_{x_i} r_{i,1,0}(x_i) \cdot \mathop{\|}_{0 \leq j < n} [\, s_{i,j+n,0}(x_i) \cdot \sum_{y_{ij}} r_{i,j+n,0}(y_{ij}) \,] \cdot s_{i,1,0}(\sum_{j=0}^{n-1} y_{ij})$$

Proof

By induction on k.

k=1: Now n=2, so $\tau_I \partial_{H_n}(U_{i,1} \| \ldots \| U_{i,n-1}) = \tau_I \partial_{H2}(U_{i,1}) = U_{i,1}$, and the lemma directly follows from the definition of $U_{i,1}$.

k+1: Suppose the lemma holds for $n=2^k$. Now we prove it to hold for $2n=2^{k+1}$ as well.

$$\tau_I \partial_{H2n}(U_{i,1}\|\ldots\|U_{i,2n-1}) = \tau_I \partial_{H2n}(\tau_I \partial_{Hn}(U_{i,1}\|\ldots\|U_{i,n-1})\|U_{i,n}\|\ldots\|U_{i,2n-1})$$

$$= \tau_I \partial_{H2n}(\{\Sigma_{xi}\, r_{i,1,0}(x_i) \cdot \mathop{\|}_{0\leq j<n} [s_{i,j+n,0}(x_i) \cdot \Sigma_{yij}\, r_{i,j+n,0}(y_{ij})] \cdot s_{i,1,0}(\sum_{j=0}^{n-1} y_{ij})\ \} \ \|\ \mathop{\|}_{0\leq j<n} U_{i,j+n})$$

Note, that we needed the conditinal axioms of table 6, to prove the first step. Using the definition of $U_{i,j+n}$ we immediately find

$$= \tau_I \partial_{H2n}(\{\Sigma_{xi}\, r_{i,1,0}(x_i) \cdot \mathop{\|}_{0\leq j<n}[\, s_{i,j+n,0}(x_i) \cdot \Sigma_{yij}\, r_{i,j+n,0}(y_{ij})] \cdot s_{i,1,0}(\sum_{j=0}^{n-1} y_{ij})\ \} \ \|$$

$$\mathop{\|}_{0\leq j<n} \Sigma_{dj\in D}\, r_{i,j+n,0}(d_j) \cdot$$

$$[\{s_{i,2(j+n),0}(d_j) \cdot \Sigma_{nij\in N}\, r_{i,2(j+n),0}\,(n_{ij})\} \ \|\ \{s_{i,2(j+n)+1,0}(d_j) \cdot \Sigma_{mij\in N}\, r_{i,2(j+n)+1,0}(m_{ij})\}] \cdot$$

$$\cdot\, s_{i,j+n,0}(n_{ij}+m_{ij})\ )$$

Note that for every $0\leq j<n$ we have two communications: the first one binding the variable d_j and the value x_i, and the second one binding y_{ij} and $n_{ij}+m_{ij}$. So we find:

$$= \tau_I \partial_{H2n}(\ \Sigma_{xi}\, r_{i,1,0}(x_i) \cdot \mathop{\|}_{0\leq j<n} \{c_{i,j+n,0}(x_i) \cdot$$

$$[\{s_{i,2(j+n),0}(x_i) \cdot \Sigma_{nij\in N}\, r_{i,2(j+n),0}(n_{ij})\} \| \{s_{i,2(j+n)+1,0}(x_i) \cdot \Sigma_{mij\in N}\, r_{i,2(j+n)+1,0}(m_{ij})\}] \cdot$$

$$\cdot\, c_{i,j+n,0}(n_{ij}+m_{ij})\ \} \cdot\ s_{i,1,0}(\sum_{j=0}^{n-1} n_{ij}+m_{ij})\ )$$

$$= \Sigma_{xi}\, r_{i,1,0}(x_i) \cdot$$

$$\mathop{\|}_{0\leq j<n} [s_{i,2(j+n),0}(x_i) \cdot \Sigma_{nij\in N}\, r_{i,2(j+n),0}(n_{ij})\} \ \|\ \{s_{i,2(j+n)+1,0}(x_i) \cdot \Sigma_{mij\in N}\, r_{i,2(j+n)+1,0}(m_{ij})\}]$$

$$\cdot\, s_{i,1,0}(\sum_{j=0}^{n-1} n_{ij}+m_{ij})$$

using the equation $(\tau x\|y) = \tau(x\|y)$, which can be derived directly from the axioms of ACP_τ. Thus we have

$$= \sum_{xi} r_{i,1,0}(x_i) \cdot \mathop{\|}_{0 \le j < 2n} [\, s_{i,j+2n,0}(x_i) \cdot \sum_{y_{ij} \in N} r_{i,j+2n,0}(y_{ij})\}] \cdot s_{i,1,0}(\sum_{j=0}^{n-1} y_{ij})$$

renaming the *n*'s and *m*'s into *y*'s again.

end proof.

So indeed, the ith upper tree first will receive a number x_i from channel [i,1,0], i.e: from its own root. Next, after some time, we will see all of its leaves send this value downward to the cells in the matrix, getting some other value in return. All processes in the leaves of the tree are interleaved, precisely as we expected. Finally, after some time, we will find the sum of all values being sent up from the leaves, appear at the root channel [i,1,0] again.
In the same way we can describe what the jth lower tree acts like, as is done in lemma 2.

Lemma 2

Assume $n=2^k$, for some given $k>0$. Then we have

$$\tau_I \partial_{H_n}(L_{1,j} \| \dots \| L_{n-1,j}) =$$

$$= \mathop{\|}_{0 \le i < n} [\, \sum_{z_{ij} \in D \cup \{nil\}} r_{i+n,j,1}(z_{ij}) \,] \cdot s_{1,j,1}(\mathrm{xor}(z_{0j},\dots,z_{n-1j})) \cdot$$
$$\sum_{u_j \in D} r_{1,j,1}(u_j) \cdot \mathop{\|}_{0 \le i < n} s_{i+n,j,1}(u_j)$$

Proof
By induction on k.
k=1: Now n=2, so the result directly follows from the definition of $L_{1,j}$.
k+1: $\tau_I \partial_{H_{2n}}(\tau_I \partial_{H_n}(L_{1,j} \| \dots \| L_{n-1,j}) \| L_{n,j} \| \dots \| L_{2n-1,j}) =$

$$= \tau_I \partial_{H_{2n}}(\mathop{\|}_{0 \le i < n} [\, \sum_{z_{ij} \in D \cup \{nil\}} r_{i+n,j,1}(z_{ij}) \,] \cdot s_{1,j,1}(\mathrm{xor}(z_{0j},\dots,z_{n-1j})) \cdot$$
$$\sum_{u_j \in D} r_{1,j,1}(u_j) \cdot \mathop{\|}_{0 \le i < n} s_{i+n,j,1}(u_j) \;\|$$
$$\mathop{\|}_{0 \le i < n} [\, \sum_{d_{ij} \in D \cup \{nil\}} r_{2i+n,j,1}(d_{ij}) \,\|\, \sum_{e_{ij} \in D \cup \{nil\}} r_{2i+1+n,j,1}(e_{ij}) \,] \cdot$$
$$s_{i+n,j,1}(\mathrm{xor}(d_{ij},e_{ij})) \cdot \sum_{f_{ij} \in D} r_{i+n,j,1}(f_{ij}) \cdot [\, s_{2i+n,j,1}(f_{ij}) \,\|\, s_{2i+1+n,j,1}(f_{ij}) \,] \;)$$

using the definition of $L_{i,j}$ and the lemma for $n=2^k$

$$= \tau_I \partial_{H_{2n}}(\underset{0\le i<n}{\|} [\Sigma_{d_{ij}\in D\cup\{\mathbf{nil}\}} r_{2i+n,j,1}(d_{ij}) \| \Sigma_{e_{ij}\in D\cup\{\mathbf{nil}\}} r_{2i+1+n,j,1}(e_{ij})] \cdot$$
$$c_{i+n,j,1}(\mathsf{xor}(d_{ij},e_{ij})) \cdot s_{1,j,1}(\mathsf{xor}(\,\mathsf{xor}(d_{0j},e_{0j}), \ldots, \mathsf{xor}(d_{n-1j},e_{n-1j})\,)) \cdot$$
$$\Sigma_{u_j\in D}\, r_{1,j,1}(u_j) \cdot \underset{0\le i<n}{\|} \{c_{i+n,j,1}(u_j) \cdot [\, s_{2i+n,j,1}(u_j) \| s_{2i+1+n,j,1}(u_j)\,]\,\}\,)$$

binding $\mathsf{xor}(d_{ij},e_{ij})$ and z_{ij} ; moreover the variables u_j and f_{ij} are identified, for all i,j.
Note that $\mathsf{xor}(\,\mathsf{xor}(d_{0j},e_{0j}), \ldots, \mathsf{xor}(d_{n-1j},e_{n-1j})\,) = \mathsf{xor}(d_{0j},e_{0j},\ldots,d_{n-1j},e_{n-1j})$; renaming d_{ij} and e_{ij} into $z_{2i\,j}$ and $z_{2i+1\,j}$ respectively, we find

$$= \underset{0\le i<2n}{\|} [\, \Sigma_{z_{ij}\in D\cup\{\mathbf{nil}\}} r_{i+n,j,1}(z_{ij})\,] \cdot s_{1,j,1}(\mathsf{xor}(z_{0j},\ldots,z_{2n-1\,j})) \cdot \Sigma_{u_j\in D}\, r_{1,j,1}(u_j) \cdot \underset{0\le i<2n}{\|} s_{i+n,j,1}(u_j)$$

end proof.

From lemma 2 we read that the jth lower tree first will receive **n** values (probably with some **nil**'s) from its leaves, say $z_{0j},\ldots,z_{n-1j}$. Then it will send $\mathsf{xor}(z_{0j},\ldots,z_{n-1j})$ to the bottom. Next it waits until it gets a value u_j from the bottom in return, and it will broadcast this value up to the leaves again, i.e: after some time all leaves, in any order, will send up u_j.
Using both lemmas we can now easily find the final proof of the correctness theorem.

Proof of the correctness theorem
Let $n=2^k$ for some $k\ge 0$.
Using the conditional axioms of table 6, one easily verifies

$$\tau_I \partial_{Hn\cup Mn}(U_{i,1} \| \ldots \| U_{i,n-1} \| M_{i,0} \| \ldots \| M_{i,n-1}) =$$
$$= \tau_I \partial_{Hn\cup Mn}(\tau_I \partial_{Hn\cup Mn}(U_{i,1} \| \ldots \| U_{i,n-1}) \| M_{i,0} \| \ldots \| M_{i,n-1}).$$

Then, using lemma 1 and the definition of $M_{i,j}$ we find

$$\tau_I \partial_{Hn\cup Mn}(\tau_I \partial_{Hn\cup Mn}(U_{i,1} \| \ldots \| U_{i,n-1}) \| M_{i,0} \| \ldots \| M_{i,n-1}) =$$
$$\Sigma_{x_i}\, r_{i,1,0}(x_i) \cdot \underset{0\le j<n}{\|} [\, s_{i+n,j,1}(\mathsf{diag}(i,j,x_i)) \cdot \Sigma_{w_{ij}}\, r_{i+n,j,1}(w_{ij})\,] \cdot s_{i,1,0}(\sum_{j=0}^{n-1} \mathsf{comp}(i,j,x_i,w_{ij}))$$

Using the conditional axioms once again we have

$$\tau_I \partial_{Hn\cup Bn}(L_{1,j} \| \ldots \| L_{n-1,j} \| B_j) = \tau_I \partial_{Hn\cup Bn}(\tau_I \partial_{Hn\cup Bn}(L_{1,j} \| \ldots \| L_{n-1,j}) \| B_j).$$

From the definition of B_j and lemma 2 we find directly

$$\tau_I \partial_{Hn\cup Bn}(\tau_I \partial_{Hn\cup Bn}(L_{1,j} \| \ldots \| L_{n-1,j}) \| B_j) =$$

$$\mathop{\|}_{0\le i<n} [\, \Sigma_{zij}\, r_{i+n,j,1}(z_{ij})\,] \cdot \mathop{\|}_{0\le i<n} s_{i+n,j,1}(\mathrm{xor}(z_{0j},\ldots,z_{n-1j}))$$

so we have,

$\tau_I \partial_{En}(\,\mathbf{RANKSORT(n)}\,) =$

$$\tau_I \partial_{En}\Big(\mathop{\|}_{0\le i<n} [\, \tau_I \partial_{Hn\cup Mn}(\tau_I \partial_{Hn\cup Mn}(U_{i,1}\|\ldots\|U_{i,n-1})\| M_{i,1}\|\ldots\|M_{i,n-1})\,]$$

$$\| \mathop{\|}_{0\le j<n} [\, \tau_I \partial_{Hn\cup Bn}(\tau_I \partial_{Hn\cup Bn}(L_{1,j}\|\ldots\|L_{n-1,j}) \| B_j)\,] \Big) =$$

$$= \tau_I \partial_{En}\Big(\mathop{\|}_{0\le i<n} \Big\{ \Sigma_{xi}\, r_{i,1,0}(x_i) \cdot \mathop{\|}_{0\le j<n} c_{i+n,j,1}(\mathrm{diag}(i,j,x_i)) \Big\} \cdot$$

$$\cdot \mathop{\|}_{0\le i<n} \Big\{ \mathop{\|}_{0\le j<n} [\, c_{i+n,j,1}(\mathrm{xor}(\mathrm{diag}(0,j,x_0),\ldots,\mathrm{diag}(n-1,j,x_{n-1}))\,] \cdot$$

$$\cdot \quad s_{i,1,0}\Big(\sum_{j=0}^{n-1} \mathrm{comp}(i,j,x_i,\mathrm{xor}(\mathrm{diag}(0,j,x_0),\ldots,\mathrm{diag}(n-1,j,x_{n-1}))\,)\Big) \Big\} \Big)$$

$$= \tau_I \partial_{En}\Big(\mathop{\|}_{0\le i<n} \Big\{ \Sigma_{xi}\, r_{i,1,0}(x_i) \cdot \mathop{\|}_{0\le j<n} c_{i+n,j,1}(\mathrm{diag}(i,j,x_i)) \Big\} \cdot$$

$$\cdot \mathop{\|}_{0\le i<n} c_{i+n,j,1}(x_j) \cdot s_{i,1,0}\Big(\sum_{j=0}^{n-1} \mathrm{comp}(i,j,x_i,x_j)\Big) \Big)$$

$$= \quad \mathop{\|}_{0\le i<n} [\, \Sigma_{xi}\, r_{i,1,0}(x_i)\,] \cdot \mathop{\|}_{0\le i<n} s_{i,1,0}\Big(\sum_{j=0}^{n-1} \mathrm{comp}(i,j,x_i,x_j)\Big)$$

$$= \quad \mathop{\|}_{0\le i<n} [\, \Sigma_{xi}\, r_{i,1,0}(x_i)\,] \cdot \mathop{\|}_{0\le i<n} s_{i,1,0}(\, p_i(x))$$

$=$ **SORT(n)** using the proposition of section 5.

end proof.

6. Some remarks about the complexity of RANKSORT

It is beyond the subject of this paper to study the complexity of the machine described in the former sections. Still, some obvious remarks can be made to indicate that RANKSORT in fact is only slightly suboptimal with respect to other well-known algorithms. All of these remarks are from [TH], in which a review over thirteen VLSI sorting algorithms is presented.

As it turns out, RANKSORT works with n^2 processors and in log n time. So one could say, comparing this complexity behaviour with for instance the $n$log n time sequential mergesort algorithm, a factor O(n) time can be 'won' by exchanging it for a large amount of space. In some well-known models of VLSI complexity this notion of 'space' is worked out in more detail (see: BILARDI & PREPARATA [BP] and THOMPSON [TH]).
A convenient unit of *area* of a VLSI chip is the square of the minimum separation between parallel wires. Every square unit on the chip surface may contain a *wire* element, or a piece of a *gate*, i.e: a localised set of transistors or other switching elements, which perform a simple logical function. Starting from a square tessellation of the chip surface, some restrictions on the design of the chip are made. For instance, no pieces of gates may overlap (i.e: any square unit only contains a part of at most one gate) and only two (or perhaps three, depending on the model) wires can pass over the same point (any square unit can represent the crossing of at most two wires).
The unit of time can be taken to be the time of one clock pulse,so the time behaviour of the chip can be expressed in a number of pulses. Note, that the specification of RANKSORT, as given in section 4, can be implemented in an *unclocked* network, since we have asynchronous cooperation between individual processes. A *clocked* network, however, is a special case of the general network in which no restrictions on timing are made, so a clock can do no 'harm' to the correct behaviour of the machine.
Of course, the list of restrictions mentioned here is not complete. In [BP] all restrictions are formulated in detail, as rules on the underlying graphs representing the VLSI networks.

In [BP] and [TH], VLSI models are used to find lower and upper bounds for the complexity behaviour of sorting algorithms. Assume a VLSI chip has area A and needs time T to do its task, then a useful complexity measure turns out to be $A \cdot T^2$ (although AT and AT/log A can be used as well). In [TH] a lower bound for the complexity of any sorting algorithm is put at $AT^2=\Omega(n^2 \log n)$. Moreover about thirteen VLSI sorting algoritms are examined, ranging from $O(n^2 \log^2 n)$ to $O(n^2 \log^5 n)$, and hence all are only slightly suboptimal in AT^2 behaviour.

Although we have $O(n^2)$ wires in the network, we need some more wire unit elements to implement the orthogonal tree network on a VLSI chip. The RANKSORT algorithm turns out to be $A=O(n^2 \log^2 n)$, and thus $AT^2=O(n^2 \log^4 n)$, which can be understood by making the following observation.
As we can see, the orthogonal tree network consists of $O(n^2)$ processors, interconnected by a number of wires. Note, that every wire is built up from a number of unit elements of O(1) area.

Now consider the projection of the orthogonal tree network on a plane, as pictured in figure 6, we see we have to leave at least log n units of space between two rows or columns of matrix cells, since this is the minimum area needed to construct a tree in between these cells. So, we may conclude that the width of the square circuit is O($n$log n), since the area between two matrix

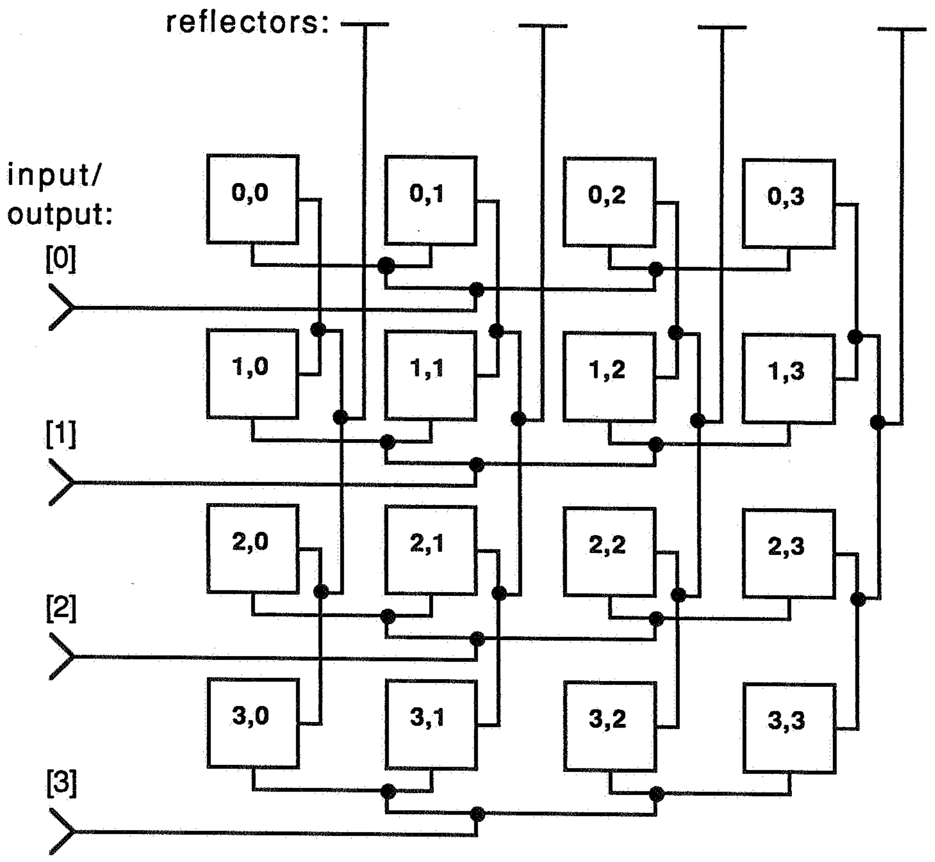

figure 6. A two dimensional projection of the orthogonal tree network with n=4.

processors is O(log n), and any processor is of O(1) area. So we find directly that the total area of the orthogonal tree network is at most O($n^2\log^2 n$), which is the surface area of a square of width O($n$log n). Since the sorting task can be done in O(log n) time, we have $AT^2 = O(n^2\log^4 n)$.

Indeed, RANKSORT can be said to be slightly suboptimal with respect to the lower bound of $AT^2=\Omega(n^2\log n)$. Clearly, however, the strong time performance of the algorithm takes a large amount of area, so we may not expect the circuit to be of much interest until chip area is cheap enough.

REFERENCES

[BBK1] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *On the consistency of Koomen's Fair Abstraction Rule*, report CS-R8511, Centre of Mathematics and Computer Science, Amsterdam 1985, to appear in TCS 51 (1/2).

[BBK2] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Conditional axioms and α/β-calculus in process algebra*, report FVI 86-17, Computer Science Department, University of Amsterdam 1986, to appear in: Proc. IFIP Conf. on Formal Description of Progr. Concepts, (M. Wirsing, ed.), Gl. Avernæs 1986, North Holland.

[BP] G. Bilardi & F.P. Preparata, *Area-Time Lower-Bound Techniques with Applications to Sorting*, Algorithmica (1986) 1: 65-91.

[BK] J.A. Bergstra & J.W, Klop, *Algebra of communicating processes with abstraction*, Theor. Comp. Sci. 37, pp. 77-121, 1985.

[KW] L. Kossen & W.P. Weijland, *Correctness proofs for systolic algorithms: palindromes and sorting*, report FVI 87-04, Computer Science Department, University of Amsterdam 1987.

[MI] R. Milner, *A calculus of communicating systems*, Springer LNCS 92,1980.

[TH] Clark D. Thompson, *The VLSI Complexity of Sorting*, IEEE transactions on computers: vol. c-32, no. 12, december 1983.

[WE] W.P. Weijland, *A systolic algorithm for matrix-vector multiplication*, report FVI 87-08, Computer Science Department, University of Amsterdam 1987.