IMPLEMENTATION OF
SIMULTANEOUS MEMORY ADDRESS ACCESS
IN MODELS THAT FORBID IT

by

Uzi Vishkin

Technical Report No. 210
July 1981
ABSTRACT

Models of synchronized parallel computation in which all the processors have access to a common memory are considered.

We focus on algorithms in models that allow simultaneous access to the same memory location, for both read and write instructions. Assume that such an algorithm uses \( p \) processors, \( d \) time units, and \( s \) memory space. We present a universal algorithm that implements this algorithm in models that forbid simultaneous access to the same memory location, using \( p \) processors, \( O(d \log^2 p) \) time units, and \( O(s+p) \) memory space. This implementation algorithm is shown to compare favorably with its conventional naive counterpart, as the extra memory space it requires is independent of the implemented algorithm.
1. Introduction

The subject of synchronized parallel computation got considerable attention in the last few years. See the list of references for example. Most of these works assume models of computation that would fit the following model of computation.

1.1 The Model

The model of computation has five submodels, differing in the outcome of access conflicts (see below), but with the following features common to all of them:

There are \( p \) processors, each equipped with small local memory and a processing unit capable of performing typical operations like reading, writing and arithmetic and Boolean operations. Each processor has an identification number \( i, 1 \leq i \leq p \).

All processors have access to a common, arbitrarily large memory for both reading and writing operations. There are no restrictions on simultaneous operations of processors as long as no pair of them attempts to have simultaneous access to the same memory location for read or write operations. The outcome of such access conflicts is the crux of the difference among the five submodels. It is discussed later.

The program is located in the common memory and written so that every processor knows what it should do in any time unit.

There is a universal clock to the program that ticks every time unit and each processor can perform one and only one elementary operation between two ticks. A starting time will be assigned to some of the instructions. The execution of such an instruction must start exactly in the starting time assigned to it. This
enables us to achieve synchronization whenever necessary. An instruction that lacks a starting time should be executed as soon as its predecessor is finished. The event of a processor reaching an instruction after its starting time will be considered illegal. The starting time is an integral part of the instruction and will usually be expressed as a function of the input parameters and the processor's serial number. The specified starting times will be called synchronization points.

Example: Let $n$ be the input length, $i$ - the processor's number, $E$ - a vector of length $p$. A typical instruction may look like:

$$\begin{align*}
[2ni] & \text{ if } i \leq n \\
& \text{ then } E(i) \leftarrow 5 \\
& \text{ else } E(i) \leftarrow 6
\end{align*}$$

Processor $i$ starts to perform this instruction in time $2ni$ and inserts 5 or 6 to $E(i)$ according to the $i$'s value. For example, if $i = 4$ and $n = 7$, the processor 4 starts performing the instruction in time $t = 56$ and inserts 5 to $E(4)$.

The depth of an algorithm is the time elapsed between the starting of the first processor and the termination of the last one. All the overheads including assignment of processors to their jobs are taken into account in the depth evaluation.

The five submodels are characterized as follows in ascending order of strength:

1. In Submodel 1, simultaneous access to the same memory location is not allowed. If it happened, it is considered illegal. Thus it is the task of the correctness proof to show that it never happens.

2. Submodel 2 permits simultaneous reading of several processors from the same location, but not simultaneous writing (illegal!).
3. Submodel 3 comprises Submodel 2 and permits simultaneous writing of several processors in the same memory location as long as all of them try to write the same value. If several processors attempt to write simultaneously different things in the same memory location, it is considered illegal.

4. Submodel 4 comprises Submodel 2 and permits simultaneous writing of several processors in the same memory location (they may attempt to write different things). In the case of several processors trying to write in the same memory location, one of them succeeds but we do not know in advance which one.

5. Submodel 5 comprises Submodel 2 and permits simultaneous writing of several processors in the same memory location. In that case the processor with the smallest identification number succeeds.

Remark: The variant of simultaneous access to the same memory location for a mixed objective (e.g. reading and writing) was not described since every such time unit can be broken into two halves: in the first half the reading is performed, and in the second half the writing is taken care of. Thus, there is a way to avoid this problem without changing the depth of an algorithm by an order of magnitude.

In [SV-81c] we establish some lower bounds for performance of algorithms in submodels 1, 2. It was shown that Submodel 2 is "stronger" than Submodel 1, and that Submodel 3 is stronger than Submodel 2. Below, we present circumstantial evidence that submodel i+1 is stronger than submodel i for i = 3,4 too. Namely, a problem will be defined such that it is solved within constant depth in submodel i+1, while we do not see how to solve this problem within constant depth in submodel i, using the same number of processors (i = 3,4).
Problem 1

Input. A vector \( A \) of length \( p \) such that \( A(j) = 0 \) or \( A(j) = 1 \) for all \( j, 1 \leq j \leq p \).

Put in cell \( \alpha, k \) such that \( A(k) = 1 \) (or zero if there is no such \( k \)).

Solution in Submodel 4:

Assume that \( \alpha = 0 \) initially.

The following instruction is executed simultaneously by all the \( p \) processors.

\[
\text{Processor } i \ (1 \leq i \leq p): \text{if } A(i) = i \text{ then } \alpha + i .
\]

Problem 2

Input. The same as in Problem 1.

Put in cell \( \alpha \), the smallest \( k \) such that \( A(k) = 1 \) (or zero if there is no such \( k \)).

Solution in Submodel 5:

Assume that \( \alpha = 0 \) initially.

\[
\text{Processor } i \ (1 \leq i \leq p): \text{if } A(i) = 1 \text{ then } \alpha + i .
\]

1.2 Results

Models that allow simultaneous access to the same memory location were shown to be (see most of the works in the list of references) natural and smooth frameworks for thinking. Therefore, the need for an inexpensive translation of these models into models that are more acceptable from the engineering point of view, arises.
The main theme of this paper is to show two implementation theorems. The first (second) provides a universal implementation algorithm that translates every algorithm that is formulated in Submodel 5 (and hence in submodels 2, 3, 4 too) into an equivalent algorithm in Submodel 1 by multiplying the depth by a factor of $O(\log p)$ ($O(\log^2 p)$) and multiplying (adding) the required memory space by a factor of $O(p)$ (a size of $O(p)$). This is done by the same number of processors as in the original algorithm. This theorem is given in Section 2(3).

The first implementation algorithm was first observed in [Ec-79]. It is naive and straightforward, but not satisfactory; if Submodel 1 is the "right" model, then the algorithm provides for utilization of $O(1/p)$ of the memory space when the algorithm is designed in terms of submodels 2, 3, 4 or 5 to be translated into Submodel 1.

The main contribution of this paper is the second implementation algorithm.
2. First Implementation Theorem

Theorem 2.1 Every algorithm that uses \( p \) processors in Submodel 5 can be implemented in Submodel 1, using the same number of processors, while the depth of the algorithm is multiplied by a factor of \( O(\log p) \) and the memory space required is multiplied by a factor of \( O(p) \).

As the proof is "natural" and attainable through simple manipulation of basic data-structures, we omit it here in favor of an illustration; the interested reader is referred to [Ec-79].

Let us take any algorithm in Submodel 5 and assume \( p = 7 \). To every memory location that is required for the algorithm, we attach a complete binary tree with \( 2^{\lfloor \log_2 p \rfloor} \) leaves, called an ACCESS-TREE and designated by the memory location (for instance \( c \)). Such a tree with its nodes is shown in Fig. 2.1.

![Figure 2.1](image-url)
To distinguish between time units of the algorithm and those of the implementation, the former are referred to as PULSES.

By a remark that appears in the description of the model we assume that in every pulse the processors may read or write, but it is impossible that some of them read and some others write in the same pulse. In other words, the pulses are classified into two sets: READING PULSES and WRITING PULSES.

We will show how every pulse of the algorithm is implemented within $O(\log p)$ time units.

Let us first demonstrate the implementation of a writing pulse. For example, the processors wish to write in the following memory addresses:

<table>
<thead>
<tr>
<th>Processor</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>idle</td>
</tr>
</tbody>
</table>

(Note that Processor 7 does not wish to write during that pulse.)

The idea of the implementation is very simple: Processor $i$, that wishes to write in address $c$, starts from node (leaf) $[c,1,i]$ of the first level of the access tree. In every level it decides either to continue climbing to the father of its current node or to quit. If its current node is the left son of its father, it "signals" (it is explained later how) that it exists there and continues to the next level. If its current node is a right son, it finds out whether there is any signal in the left brother of its node; if yes it quits, otherwise it continues to its node's father.
Address 10. The implementation of a reading pulse consists of two phases, the first of which is identical with Table 2.1 except that the "loser" processors wait at their final nodes instead of quitting. In the second phase, after the "winner" processor has read from the desired address, it copies its value into all nodes of the tree along the path from the root to its leaf, while the "losers" do the same, in due course, with the values of the fathers of their respective "waiting node". This second phase is shown in detail in Table 2.2.

<table>
<thead>
<tr>
<th>Time</th>
<th>Processor 1</th>
<th>Processor 3</th>
<th>Processor 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>copy 10 into [3,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>write in [1,1]</td>
<td>write in [1,3]</td>
<td>write in [2,3]</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>write in [1,6]</td>
</tr>
</tbody>
</table>

Table 2.2

Remarks: (1) Once a processor has copied a value, there is no need to re-read it. (2) The time requirement of a copying operation is the same as of writing. If it is not, the necessary changes in synchronization are straightforward.

The Depth. It is readily verifiable that the depth of implementation of a writing or a reading pulse is proportional to the height of the access trees, namely $O(\log p)$.

The Space. The access tree attached to each memory location in the original algorithm has $2^{\lceil \log p \rceil} - 1$ nodes. Thus, the required memory is the original space multiplied by $O(p)$. 
Question: How does a processor signal its presence in a certain node?

Answer: Add one to the value of the node; a processor interested in knowing whether another is present in a node, would check the node prior to the possible addition operation. The "suspected" node is then re-checked. (Synchronization is crucial for this signaling process!)

Table 2.1 shows the detailed implementation for a representative set of processors 1, 3, 6, all of whom seek access to Address 10. (The subscript 10 is omitted in the notation of the nodes.) In keeping with the assumptions of Sub-model 5, Processor 1 gains access while the other two quit.

<table>
<thead>
<tr>
<th>Level</th>
<th>Time</th>
<th>Processor 1</th>
<th>Processor 3</th>
<th>Processor 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>keep [1,5]</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>add to [1,1]</td>
<td>add to [1,3]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td>check [1,5]</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td>keep [2,1]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>add to [2,1]</td>
<td></td>
<td>add to [2,3]</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
<td>check [2,1] and quit</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>keep [3,1]</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>add to [3,1]</td>
<td></td>
<td>check [3,1] and quit</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>write in 10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1

Implementation of a reading pulse

To make things easier, we choose an example in which the processors wish to read from the addresses of Figure 2.2. We focus again on the access tree of
3. Second Implementation Theorem

Theorem 3.1: Every algorithm that makes use of \( p \) processors in Submodel 5 can be implemented in Submodel 1, using the same number of processors, while its depth is multiplied by a factor of \( O(\log^2 p) \), and the additional space requirement being \( O(p) \).

Proof: As in the previous section:

(a) We call the time units of the original algorithm pulses.

(b) We show first how to implement writing pulses and later reading pulses.

Implementation of a writing pulse

Let us look at a given writing pulse. The address at which processor \( i \) wishes to write is denoted by \( a_i \) (\( 1 \leq i \leq p \)). A processor which does not wish to write during the pulse is assigned fictitiously a non-existent address.

Batcher's [B-68] sorting algorithm serves as a main tool in our implementation algorithm. It can be shown that Batcher's algorithm may be implemented in Submodel 1. It requires a depth of \( O(\log^2 n) \) for \( n \) processors and \( O(n) \) memory space, for sorting \( n \) elements.

The writing pulse implementation includes two steps:

Step 1: Apply the Batcher algorithm in order to sort the pairs \((a_1, 1), (a_2, 2), \ldots, (a_p, p)\) in the lexicographic order. Namely, \((a_i, i) < (a_j, j)\) iff \( a_i < a_j \) or \( a_i = a_j \) and \( i < j \). Denote the output vector by \((a_{j_1}, j_{1}), (a_{j_2}, j_{2}), \ldots, (a_{j_p}, j_p)\). The processors that wish to write in the same address are represented in this vector by consecutive serial numbers. The smallest of which is attached to the lowest numbered processor in question.
Step 2: Processor 1: Write in $a_{j_1}$ the value that Processor $j_1$ intended to write there.

Processor $i$ $(2 < i < p)$: if $a_{j_{i-1}} \neq a_{j_1}$

then write in $a_{j_1}$ the value that processor $j_1$ intended to write there.

Obviously, Step 2 completes the implementation of a writing pulse.

Implementation of a reading pulse

Denote by $a_i$ the address from which processor $i$ $(1 \leq i \leq p)$ wishes to read in a given pulse.

The reading pulse implementation includes three steps.

Step 1: Identical with its counterpart in the implementation of a writing pulse.

For the next step we introduce an auxiliary $p \times 3$ array denoted by $A$.

$\tilde{A}(i,1)$ - is supposed to contain the value of the address $a_{j_1}$ at the end of the pulse implementation

YES if the value of $a_{j_1}$ is already

For $1 \leq i \leq p$, $A(i,2)$ - contains it is set to NO before each reading pulse implementation.

$A(i,3)$ - is supposed to contain the value of address $a_i$ at the end of the pulse implementation.
Step 2: Processor 1: Copy the value of \( a_{j_1} \) into \( A(1,1) \) and \( A(1,2) \sim YES \)

Processor \( i \) (\( 2 \leq i \leq p \)): if \( a_{j_{i-1}} \neq a_{j_i} \)

then copy the value of \( a_{j_i} \) into \( A(i,1) \)

\( A(i,2) \sim YES \)

The situation is now as follows: for every \( j_i \) (\( i = 1, \ldots, p \)), such that, processor \( j_i \) has the smallest serial number among the processors that wish to read from address \( a_{j_i} \), the correct value is written in \( A(j_i,1) \). Our current task is to disseminate this value among the other \( A(j_i,1) \)'s. It is executed in cycles in Step 3 using the following idea: as soon as processor \( i \) (\( 1 \leq i \leq p \)) knows that \( A(i,1) \) was updated (\( A(i,2) \) set to YES) it copies the value of \( A(i,1) \) into \( A(k,1) \) for an appropriate \( k \) (\( k > i \)) such that \( a_{j_k} = a_{j_i} \). In the first cycle \( k = i+1 \), and in the \( m \)-th cycle \( k = i+2^{m-1} \). (Note that whenever a processor joins the cyclic process, it does so with a \( k \) which depends on the current cycle.)

Step 3:

Processor \( i \) (\( 1 \leq i \leq p \)): \( k_i \sim 0 \)

wait until \( A(i,2) \) is turned to YES

while \( i+2^k \leq p \) and \( A(i+2^k,2) = NO \) do

\( A(i+2^k,2) \sim YES \)

\( A(i+2^k,1) \sim A(i,1) \)

\( k_i \sim k_i + 1 \)

od

\( A(j_i,3) \sim A(i,1) \)

Read \( A(i,3) \).
The auxiliary numbers $k_i$ ($i = 1, \ldots, p$) are introduced since in Submodel 1 different counters are needed for different processors.

**Depth.** Step 1 in both the writing and reading pulses implementations is of depth $O(\log^2 p)$. Step 2 in both is of constant depth. The loop of Step 3 (which is executed synchronously) contains no more than $\log_2 p$ cycles, each of constant depth. Therefore the depth of Step 3 is $O(\log p)$. It is readily seen that the space required for both pulse implementations is proportional to $p$.

### Remarks

1. No known algorithm for sorting $n$ elements using $n$ processors achieves better depth than $O(\log^2 n)$. (For example, the algorithm of [SV-81a], that is formulated in Submodel 2, achieves this depth for only $n/\log n$ processors but does not improve it for $n$ processors.) Thus, there is no advantage by an order of magnitude in translating an algorithm from Submodel 5 to Submodel 2 over direct translation into Submodel 1.

2. Note that for both pulse implementations the only obstacle to depth reduction is Step 1, namely the sorting subroutine. Whenever Submodel 1 will be capable of implementation (hopefully in the near future) and once there is greater preference among algorithm designers for the stronger submodels, the sorting subroutine may be profitably realized in hardware on the basis of Batcher's network, thereby achieving a much closer approach to the upper limit set by the other steps of the pulses implementations, namely $O(\log p)$. It might be interesting in case of an efficient such hardware implementation to conceive an additional submodel in which simultaneous writing in the same memory location is allowed but simultaneous such reading is forbidden. In that case even the barrier of $O(\log p)$ that is caused only by Step 3 of the reading pulse implementation would not exist any more.
4. Conclusion

Submodel 1 is considered to be more acceptable than the others from the practical point of view [Ec-79], [P-78]; still, considerable difficulties were encountered in attempts to derive efficient algorithms within its assumptions. The importance of the results in this paper is that they provide motivation to design and analyze algorithms in the stronger submodels, for which relatively cheap and automatic translation procedures are available.

Acknowledgement

I would like to thank Dr. Y. Shiloach for reading the manuscript and for his helpful remarks.
References


