ARRAY PROCESSING MACHINES

J. van Leeuwen & J. Wiedermann

RUU-CS-84-13
December 1984

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-531454
The Netherlands

ARRAY PROCESSING MACHINES

J. van Leeuwen & J. Wiedermann

Technical Report RUU-CS-84-13
December 1984

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
the Netherlands

ARRAY PROCESSING MACHINES*

J. van Leeuwen and J. Wiedermann**

Department of Computer Science, University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht, the Netherlands.

Abstract. We present a new model of parallel computation called the "array processing machine" or APM (for short). The APM was designed to closely model the architecture of existing vector- and array processors, and to provide a suitable unifying framework for the complexity theory of parallel combinatorial and numerical algorithms. After an introduction to the model and its basic programming techniques, we show that the APM can efficiently simulate a variety of extant models of parallel computation and vector processing. In particular it is shown that APMs satisfy Goldschlager's "parallel computation thesis". Several extensions to the basic model of the APM are discussed.

1. Introduction. The technical development of ever faster computers requires that we review the adequacy of the current models of (sequential) computation. The Turing machine (see e.g. [1]) and the Random Access Machine (or RAM, cf. Cook & Reckhow [5]) are the outstanding members of a class of models which have captured the fundamental notions of computability and the main architectural features and capabilities of existing early-generation computers, respectively. Current designs of (super-)computers, however, are based on a variety of architectural innovations which attempt to incorporate advanced forms of pipelined and parallel processing (cf. Hockney & Jesshope [11]). Programming languages are being designed that provide a choice of

---

*This work was carried out while the second author was visiting the Dept. of Computer Science, University of Utrecht, the Netherlands (Fall 1984).

**Address: VUSEI-AR, Dubravska 3, 842 21 Bratislava, Czechoslovakia.

high-level primitives for "parallel programming" based on the expedience of certain operations on a given line of (super-)computers, like in the early FORTRAN days (see e.g. Hwang et. al. [12]) New notions of computational complexity have emerged to measure the performance of vector- and parallel algorithms which are not immediate from the traditional class of models of computation.

In recent years a "second" class of computational models has emerged, whose members attempt to abstract and formalize the salient features of vectorized and/or parallel computation. We mention the following examples:

(i) vector machines (Pratt & Stockmeyer [14]),
(ii) MRAM's (Hartmanis & Simon [10]),
(iii) P-RAM's (Fortune & Wyllie [8]),
(iv) k-PRAM's (Savitch & Stimson [17]),
(v) alternating Turing machines (Chandra, Kozen & Stockmeyer [2]),
(vi) LPRAM's (Savitch [16]),
(vii) Concurrent Read-Exclusive Write PRAM's (CREW PRAM's, see e.g. Stockmeyer & Vishkin [19]),
(viii) SIMDAG's (Goldschlager [9]).

For definitions we refer to the open literature. See Cook [4] for an excellent account of the current research on parallelism using circuit-based models.

As pointed out by van Emde Boas [22] there are various computational correspondences between the members of the "second" machine class. One is the property of satisfying the "parallel computation thesis" (after Goldschlager [9]) which asserts that PARALLEL TIME on these machines is polynomially related to SPACE on traditional (sequential) models. (Wiedermann [25] has proposed a model of parallel computation that appears to belong to an "intermediate" class of machines.) None of the existing models in the "second" class can be held as a very realistic model of existing parallel computers, either because of the highly idealised architectural assumptions that are

made (as in SIMDAG's) or because of the extreme power of certain primitives in the instruction set (as in vector machines). On the other hand one may argue that no machine model from the "second" class can be completely realistic as e.g. the capability to activate an exponential number of processors in polynomial time (as in P-RAM's) is not in agreement with physical law. (Compare Chazelle & Monier [3], Schorr [18].) Yet an adequate model is required for the proper study of combinatorial and numerical algorithms in a vectorized or parallel computing environment.

Our aim is twofold: (a) we like to have a computational model that relates as closely to current parallel computers (in the sense of [11]) as RAM's do to early-generation computers and (b) we like to have a model that incorporates, perhaps indirectly, the essential features of all (known) members of the "second" machine class. In this paper we present the "array processing machine" or APM (for short), which we believe to be a suitable model of parallel computation that satisfies our aim. The APM can be viewed as an extension of the RAM-model with a vector-processing capability, but with primitive operations that are quite different in detail from Pratt & Stockmeyer's "vector machines". The model is described in Section 2, where we also introduce the basic complexity measures for APM computations. In Section 3 we show basic programming techniques for APM's including programs to sort in $O(\log^2 N)$ time and to solve a tridiagonal system of linear equations in $O(\log^2 N)$ time as well. (It is presently open to solve a tridiagonal system on APM's in $O(\log N)$ time.) In section 4 we show that APM's belong to the "second" machine class by proving that they satisfy the "parallel computation thesis", and show efficient simulations of some common models of parallel computation on the APM. In Section 5 we discuss the flexibilities in the APM-model and propose a variety of extensions to its basic instruction set that might be suitable for implementation in existing vector computers.

2. The basic model of the "array processing machine". We assume the reader to be familiar with the ordinary RAM-model (see e.g. Aho, Hopcroft, & Ullman [1]) and the global characteristics of current

supercomputer architectures (cf. Hockney & Jesshope [11]). The "array processing machine" or APM (for short) can be viewed as a RAM, extended with suitable primitives for vector processing. Its name derives from its capability of addressing any row of contiguous memory locations (an "array") for processing purposes. In this section we outline the basic model of the APM.

2.1. Architecture of the APM. An APM consists of the following components (see figure 1):

(a) a (one-way) read-only input file, divided into fields (locations or cells) that contain one datum of input each. We only allow scalar input. Either single data or vectors of (consecutive) data can be read from the input file.

(b) a (one-way) write-only output file, likewise divided into fields that can contain one datum of output each. We limit the model to scalar output. Either single data or vectors of data (from consecutive locations in memory) can be output to the output file. The output file is initially empty.

(c) a (random access) data memory, divided into words (locations or cells) that can be individually addressed. No limit is set to the wordsize. Memory can be viewed as a linear array with addresses ranging from 0 to N, for some N sufficiently large. Either single words or entire vectors of consecutive words, i.e., linear subarrays, can be addressed.

---

figure 1

(d) a (random access) program memory, containing the APM program that is executed. Each APM instruction is assumed to fit in one word. Memory addressed can be used as labels. APM programs are always loaded at address 0.

(e) a program counter, containing the address of the next instruction in program memory that must be executed. Jump instructions (viz. tests on zero) can alter the contents of the program counter in a straightforward manner.

(f) a (scalar) accumulator AC, which is simply a "special" word or register. Many instructions use the AC as an implicit source or destination (scalar) operand.

(g) a vector accumulator VAC, which is a potentially unbounded linear array of words or registers. Many vector instructions use the VAC as an implicit vector operand. Vectors are loaded into the VAC starting at its first word, overwriting the entire previous contents of the VAC. Thus the VAC always has a length equal to the length of the current vector stored in it. There can be no addressing of words within the VAC. Operations on the VAC can make use of "masks".

(h) buffer registers for I/O and memory accesses and arithmetic-logic circuitry, collectively referred to as the ALU. It is assumed that the ALU is engineered for executing both scalar and vector operations with great efficiency.

The parts (e) through (h) together form the "processor" of the APM (cf. figure 1). When a vector in memory is addressed we assume that it "streams" as one unit into the VAC. Thus the VAC is the model-equivalent of the vector registers encountered in all supercomputer architectures. Masks are vectors with 0-1 valued components. No separate registers are provided for storing masks, and (hence) masks should always be developed in memory.

2.2. Instruction set of the APM. Clearly the instruction set of the APM must include all ordinary RAM-instructions (see e.g. Aho, Hopcroft, & Ullman [1]), referred to as scalar instructions. The capabilities of the APM as a parallel computer derive from its assortment of

vector instructions, which normally operate by applying a fixed scalar operation to the components of specified vector-operands in parallel. (One of the operands is normally stored in the VAC.) The instruction set is kept small and simple, to serve the purposes of the model. In section 5 we discuss various additional instructions that one might like to include.

The scalar instructions of the APM are similar to the corresponding RAM-instructions (see figure 2). Each instruction consists of an operation code and an address field. The address field denotes either an address in data memory (in which case the AC is implicit as the source or destination register) or an address in program memory (in which case it serves as an instruction label). In the former case addresses can be specified "direct" or "indirect" in one of the following ways:

a) = i, indicating the integer i itself.

b) i , indicating the contents of word i (denoted as [i]).

c) *i , indicating the contents of the word addressed by the contents of register i (error if out of bounds).

In the latter case we allow symbolic addresses or "labels" that must occur as (unique) prefixes to instructions in the APM program.

The vector instructions of the APM can be viewed as parallel analogs of the scalar instructions discussed above (see figure 3). We assume that each vector instruction is counted as "one unit". Vectors

---

figure 2

---

figure 3

and masks are adressed through scalar registers, i.e., by direct addresses, by giving both the address of the first and of the last component. A vector instruction is performed by applying the corresponding scalar instruction to the vector operand and/or the VAC component-wise "in parallel". When a mask is specified, the instruction only applies to the components in positions corresponding to the 1-values in the mask. (Thus the components in positions corresponding to 0-values are "masked" from the operation and remain unaltered.) Masks must be explicitly constructed in memory, typically by means of the VTGTZ or VTZERO instruction. In all vector instructions that involve more than one vector operand (e.g. a mask), the length of the vector operands involved must be identical. There is no implicit "compression" or "decompression", cf. Section 5.

2.3. APM-programs. In its most basic form, an APM-program is a sequence of APM-instructions. The instructions are implicitly numbered 0, 1, 2, ... corresponding to their ultimate address in the program memory. Scalar and vector instructions can be mixed in arbitrary order. Execution begins at address 0 and proceeds sequentially. Only branching instructions could cause "jumps". Execution proceeds until a HALT instruction is encountered (normal ending) or an instruction is undefined (abnormal ending), like division by zero. Only the former is counted as a valid ending of a computation. The "result" of a (valid) computation is the contents of the output file after the normal ending of the program on a given input file. The result of invalid

computations is undefined.

Definition. A function or problem f is APM-computable if and only if there is an APM-program π that realizes the exact input-output behaviour of f.

Observe that all "computation" in an APM is bound to the AC and the VAC.

We shall use various simplifications when dealing with APM-programs. These include the use of symbolic addresses and labels, and of Pascal-type control structures. The "translation" to a basic APM-program will usually be straightforward, and maintain the essential instruction count (to within a constant factor). We assume the reader to be familiar with the similar practice for RAM-computations and RAM-programs (cf. Aho, Hopcroft, & Ullman [1] or e.g. Engeler [6]).

The time complexity of APM-computations is defined in analogy to RAM-computations (cf. Aho, Hopcroft, & Ullmann [1]): each instruction is charged "one" unit of time (uniform cost criterion) or "log M" units of time (logarithmic cost criterion), where M is the maximum scalar value of a component or address involved in the instruction when it is executed. Thus in the latter case an instruction may be charged differently each time it is executed. Which cost criterion is used will depend in the sort of quantitative result we are after. Most interesting is the charging principle for vector instructions: the time charged is equal to time charged for the most expensive scalar operation on a component. In particular when the uniform cost criterion is used all vector instructions take "one" unit of time, just like all scalar instructions. The space complexity of APM-computations is defined exactly as for RAMs: it is the largest address in data memory used in the computation.

One additional complexity measure is introduced that is specific to APMs. The "degree of parallelism" of an APM-computation is defined as the size of the largest vector that is loaded in the VAC during the

computation.

Notation

T(n) = the time complexity of and APM-program, maximized over all admissible input files of size n.

S(n) = the space complexity of an APM-program, maximized over all admissible input files of size n.

L(n) = the degree of parallelism of an APM-program, maximized over all admissible input files of size n.

Note that the "parallelism" of APMs derives from the implicit parallelism of "identical" operations and not from any parallelism of instructions. Thus it is a prototypical SIMD machine (see e.g. Flynn [7]) in its basic mode of operation. In the model we assume that there is no à priori bound on the maximum length of the vectors that can be processed in one instruction, to facilitate the study of vectorized algorithms without further conceptual complication. If a maximum length L is in effect, then the real processing time of an APM-program may rise to as much as $O(\frac{L(n)}{L}T(n))$.

3. Basic APM algorithms. We contend that the APM model is well-suited for analysing the vectorisable structure of numeric and non-numeric, i.e., combinatorial, problems. In this Section we present efficient APM-programs for a number of tasks, to give an impression of the power of vector instructions. The programs also illustrate a number of programming techniques that are essential in deriving efficient parallel algorithms on APM's in general. We use the uniform cost criterion throughout.

Lemma 3.1. The sum of a vector of N scalars is APM-computable in O(log N) steps.

Proof.

We assume that N is given in address Na and that the vector is stored as an array A[0..N-1] at address Aa. The program for computing $S=\sum_{1}^{N} A[i]$ illustrates the techniques of folding a vector. The first step

is to sum A[0..M-1] and A[M..N-1] (M=N/2) and to store the result in A[0..M-1]. This is repeated for the array A[0..M-1], until eventually M=1 and the value of S is in A[0]. Care is needed when N or M is odd, and clearly all vector-adding is done into the VAC. The resulting APM-program is shown in figure 4. Clearly the computation requires about log N iterations of the basic loop, and each iteration involves 3 vector instructions and another O(1) scalar instructions. This gives a total time bound of O(log N). □

Lemma 3.2. The rank of an element X in an ordered set of N elements is APM-computable in O(log N) steps.

Proof.

We assume that N and X are given, and that the set of N elements is given as an array A[0..N-1] (not necessarily in sorted order). First compute a vector Y of length N consisting of all X's by the technique of recursive doubling, as follows. For i from 0 to log N, construct a vector Y_i of length 2^i consisting of all X's. Because Y_{i+1} can be viewed as the juxtaposition of two copies of Y_i, two VLOAD- and VSTORE-operations and O(1) scalar instructions suffice per iteration. The vector Y is assembled by juxtaposing vectors Y_i for values of i corresponding to the 1's in the binary expansion of N. To compute the rank of X one can now proceed as follows. VLOAD the vector Y, VSUB the array A, and do a VTGTZ to obtain a 0-1 vector Z. The rank of X wil be one plus the sum of the elements of Z. Using lemma 3.1 the entire

figure 4

algorithm can be performed in O(log N) steps on an APM. □

The results that follow will all make use of the technique of masking. Masking enables an APM to perform its vector instructions over selected components of the vector in the VAC only. The components that are selected correspond to the 1's in an appropriate mask. As in a real vector computer, masks must be explicitly computed and thus add "practical" overhead costs to the ideal parallel or vectorized APM-algorithms.

Lemma 3.3. The maximum of an ordered set of N elements is APM-computable in O(log N) steps.

Proof.

We assume that N is given, and that the set of elements is given as an array A[0..N-1]. Once again we use the idea of folding. Let M=N/2. By VLOADing A[0..M-1] and vector-comparing it to A[M..M-1], one can compute a mask α that has 1's precisely in the positions i($0 \le i \le M-1$) such that A[i] is less than A[M+i]. Now VLOAD A[0..M-1], and do a masked VLOAD of A[M..N-1] on top of it using mask α. VSTORE the result in A[0..M-1], and it should be clear that the maximum of the original array is confined to this vector. (If N is odd, a similar modification as in the proof of lemma 3.2. will do.) Repeat the procedure until M=1, and the desired maximum is available in A[0]. The APM-program (in more informal notation this time) is shown in figure 5. As there are log N iterations that take O(1) instructions each, the algorithm takes

figure 5

O(log N) time. □

Lemma 3.4. The merge of two sorted vectors of N elements ($N=2^k$, some $k\geq 0$) is APM-computable in O(log N) steps.

Proof.

We assume that $N=2^k$ is given, and that the sorted N-vectors are stored in consecutive order in an array A[0..2N-1] with the first vector in increasing order and the second in decreasing order. The program implements Batcher's "bitonic merge", see e.g. Stone [20] or Knuth [13]. The first step compares the elements A[i] and A[N+i] and swaps the smaller to the left and the larger to the right, for i from 0 to N-1. The procedure is repeated on the two halves of A and proceeds recursively, until blocks of length 1 are reached. After j steps ($j\geq 1$) the array is divided into 2^j blocks $A_0,\ldots,A_{2^j-1}$ of length $N/2^j$ each, such that for each $1\leq l<2^j-1$ the elements of block A_l are all less than or equal to the elements of the succeeding blocks $A_{l+1},\ldots$. The program will make sure that at the beginning of the next step a mask α is available with $\alpha[0..2N-1]$ equal to $(0^{N/2^j}1^{N/2^j})^{2^j}$. Denote the two halves of A_l by A_{l0} and A_{l1} ($0\leq l\leq 2^j-1$). Let LOW correspond to $A_{00}A_{01}A_{10}A_{11}\cdots A_{2^j-10}$ and HIGH to $A_{01}A_{10}A_{11}\cdots A_{2^j-10}A_{2^j-11}$. A Batcher step is implemented "in parallel" by comparing the vectors LOW and HIGH, constructing the appropriate mask, and exchanging elements within the blocks again by suitable operations on the vectors LOW and HIGH (see figure 6). As there are

figure 6

only O(1) instructions involved in executing a Batcher step and there are log N steps in all, the algorithm takes O(log N) time on an APM. □

By executing the technique of Lemma 3.4. one can implement Batcher's sorting algorithm (cf. Stone [20]) and prove that an N-vector can be sorted on an APM in $O(\log^2 N)$ steps.

For the next result, let $\{x_i\}_{0\le i\le N}$ be defined by a three-term recurrence $x_i=a_i x_{i-1}+b_i x_{i-2}$ $(2\le i\le N)$ with initial values $x_0=1$ and $x_1=a_1$.

Lemma 3.5. The terms $\{x_i\}_{0\le i\le N}$ of a three-term recurrence ($N=2^k$, some $k\ge 0$) are APM-computable in $O(\log^2 N)$ steps.

Proof.

We assume that N is given and that the coefficients $a_1, a_2, \ldots, a_N$ and $o, b_2, \ldots b_N$ are available in N-vectors A and B respectively. By vector-comparing e.g. A to itself, one can compute on N-vector C of all zeros in O(1) time. Interpret the vertical slice of 4 elements in position i as a 2x2 matrix Y_i with

$$Y_1 \quad \begin{bmatrix} A[i] & B[i] \\ C[i] & D[i] \end{bmatrix}$$

and define the "2-vectors" X_i by $X_i=(x_i\ x_{i-1})^T$ $(1\le i\le N)$. Define $X_0=(1\ 0)^T$. It follows that $X_i=Y_i \circ X_{i-1}$ and hence $X_i=(\prod_1^i Y_j)\circ X_0$. Our task is completed once we have computed the N products $\prod_1^i Y_j$. We shall accumulate the results in the very vectors A through D.

At the i^{th} step, beginning with i=0, the positions will be divided into $N/2^i$ blocks of length 2^i. The slices in a block represent the 2^i subproducts Y_{j+1}, $Y_{j+1}\ Y_{j+2}$,..., $\prod_{j+1}^{j+2^i} Y_1$ for the corresponding value of j. Also a mask M will have been computed of the form

take the last matrix in every block, e.g. $\prod_{j-2^i+1}^{j} Y_1$, and multiply it with all matrices in the succeeding block. It should be clear that we obtain $N/2^{i+1}$ blocks of size 2^{i+1} with the desired "doubled length" chain of subproducts, and we can proceed with the $(i+1)^{st}$ step. After $\log N$ steps the N products $\prod_{1}^{i} Y_j$ are available in one "block".

The i^{th} step can be implemented on the APM as follows. By doing a vector-compare between M and its left-shift over 1, one can compute a mask $M'=(0^{2^i-1}1)^{\frac{N}{2^i}}1\,0^{2^i}$. Using M' we can select the multiplicants from the blocks and store the elements in the corresponding positions of four N-vectors E, F, G, and H. (Note that E, F, G, and H contain zeros in all remaining positions.) By uniform recursive doubling one can create 2^i copies of each multiplicant in E, F, G, and H and align them with the 2^i matrices as they are represented in A, B, C, and D. The matrix multiply in every position (slice) can be uniformly caried out in O(1) vector-multiply's and vector-add's. The mask for the next step is easily computed from M. As the i^{th} step thus required O(i) instructions, due largely to the recursive doubling of the multiplicants, the algorithm requires in the order of $\sum_{0}^{\log N} O(i) = O(\log^2 N)$ time on the APM. □

Lemma 3.5. shows an interesting distinction between traditional (idealized) models of parallelism and APM's. Whereas the N terms of a three-term recurrence are evaluated in O(log N) parallel time when some kind of unbounded parallelism is assumed (see e.g. van Leeuwen [23]), the implementation on an APM apparently cannot avoid the greater expense of every "parallel" step leading to an $O(\log^2 N)$ total time bound. It is presently open to compute the N terms of a three-term recurrence in O(log N) time on an APM.

Lemma 3.6. The solution of an NxN tridiagonal system of linear equations Ax=b ($N=2^k$, some $k\geq 0$) is APM-computable in $O(\log^2 N)$ steps.

Proof.

(We assume that A is non-singular and admits an LU-decomposition.) It was shown by Stone [21] (see also [23]) that the LU-decomposition of a tridiagonal system can be computed by evaluating the terms of a suitable three-term recurrence. The solutions of Ly=b and Ux=y are computed by evaluating the corresponding two-term recurrences. By lemma 3.5. all phases can be computed in $O(\log^2 N)$ time on an APM. □

It is presently open to solve NxN tridiagonal systems in O(log N) time on an APM.

4. The computational power of APMs. There are two ways to establish the computational power of machine model: (i) by studying the hierarchy of its complexity classes, and (ii) by comparing it to other prominent machine models. In this Section we will follow the latter approach. The larger part of this Section will be devoted to showing that APM's are polynomially related to SIMDAG's, and thus belong to the "second machine class" (cf. Section 1). Alternatively the result may be seen as an argument for the practicality of the "second machine class", as APM's capture the salient features of existing vector computers. Note that SIMDAG's are more or less identical to CRCW-PRAM's ("concurrent read- concurrent write parallel random access machines") and the results of Stockmeyer & Vishkin [19] can be brought to bear on APM's. We will briefly discuss the relation between APM's and circuits later in this section. In all results the logarithmic cost criterion will be assumed.

A SIMDAG (Goldschlager [9]) consists of a CPU, a set of parallel processing units (PPU_0, PPU_1, ...) with local memory (y-locations), and a global random-access memory (x-locations). The PPU's carry their own index in a "signature" register. The SIMDAG is an "SIMD" machine (Flynn [7]). The CPU executes a SIMDAG-program serially, and either applies an instruction directly to global memory (indirect addressing allowed) or broadcasts it to an initial segment of the PPU's for parallel execution in the local memories or on global memory (with indirect addressing only via y-locations). For theoretical

reasons there is no explicit multiplication or division instruction in the instruction set (cf. Hartmanis & Simon [10]). PPU's are allowed to simultaneously read the same location in global memory, but when PPU's attempt to simultaneously write to the same location only the PPU of lowest index succeeds. SIMDAG's satisfy the "parallel computation thesis" (cf. Section 1).

In the simulation of a SIMDAG by an APM, the local and global memories are represented as vectors. To simulate a fetch by the PPU's from global memory, the following technique is useful. Suppose there are P processors, and let global memory have size S. Represent the global memory by a vector GLOB, and let the corresponding addresses be stored in a vector ADDR. Let the requests (= addresses in global memory) be stored in a vector REQ, with the indices 0 through P-1 stored in vector PROC. Use an auxiliary vector DEPOS to store the desired values, and vectors MARKP and MARKM to "mark" locations corresponding to processor requests and memory. See figure 7 for the conceptual relationships between the vectors. Think of PROC//GLOB, REQ//ADDR, and MARKP//MARKM as single vectors (//denotes concatenation).

Lemma 4.1. A "parallel" fetch of the PPU's of a SIMDAG can be simulated on an APM in $O(\log^2(P+S))$ steps.

figure 7

Proof.

(Note that the steps cost log max {P+S, L} each, where L is the largest value stored in global memory.) Sort the vector REQ//ADDR using Batcher's algorithm (cf. Section 3), and carry out the same sorting steps to the vectors PROC//GLOB and MARKP//MARKM. The ordering should be such that requests appear immediately to the right of the corresponding addresses. In case there are several requests for a single address, the requests should appear in order of increasing processor index. Although the construction of the proper masks in each step will be slightly more tedious, this part will require $O(\log^2(P+S))$ steps. In O(1) further vector steps, beginning with the construction of a mask that shows the first requests lined up for every memory address (if any), one can copy the corresponding values from GLOB into DEPOS and thus satisfy the fetches of the contending processors with lowest index. Now sort "back" by first sorting MARKP//MARKM into order and next sorting PROC and ADDR seperately, using Batcher's algorithm. The sorting steps are carried out identically on the corresponding "parallel" vectors, this time including DEPOS.

Next consider the vectors REQ, PROC, and DEPOS' ≡ DEPOS[0..P-1]. Sort REQ and apply the same sorting steps to PROC and DEPOS'. The ordering should be such that requests for the same address appear in order of increasing processor index. The result is that REQ is divided into blocks of identical requests of which the first in every block has been satisfied (with the desired value appearing in the corresponding location of DEPOS'). Now repeat the following for i=0, 1, ..., log P: load DEPOS' into the VAC, construct a mask M of the locations that have not been previously written into, add 0^{2^i} //DEPOS'[0..P-1-2^i] into the VAC, using M, and store the result back into DEPOS'. Note that this method copies ("broadcasts") the first value of a block into all further locations of a block for all blocks simultaneously, with the proper masking to make sure that a value is not "broadcas" beyond the limits of its block. Finally, sort REQ and PROC and DEPOS' back into the order of increasing processor index. The number of steps required is $O(\log^2 P)$, $O(\log P)$, and again $O(\log^2 P)$,

hence $O(\log^2 P)$ total. The fetched value for each processor can now be read directly from the corresponding location of DEPOS. □

It should be clear that parallel fetches from local memory and parallel store's to local or global memory can be simulated in "$\log^2$-" steps by the same technique of lemma 4.1.

Theorem 4.2. Let T(n), P(n), Q(n), and S(n) be APM-countable. Any SIMDAG that uses T(N) time, P(N) processors with Q(N)-bounded local memory, and S(N) global memory can be simulated by an APM in $O(T^2(N) \log^2 \max\{P(N)+S(N), P(N)Q(N)\})$ time.

Proof.

(For simplicity we use T, P, Q, an S to denote the various time, processor, and memory bounds.) In addition to the vectors PROC, REQ, GLOB, ADDR and alike (cf. figure 7) we use the vectors LOC-M and LOC-ADDR of size P.Q to simulate the local memory locations and addresses. In a vector LOC-INDEX of size P.Q we store the processor index for every location, and it is helpful to think of the locations as organized into P blocks of size Q and equal processor index. It is clear that every (global) step of a SIMDAG's CPU can be directly simulated by an APM.

The execution of a "parallel" instruction by a SIMDAG can be divided into the following phases:

I (Read phase) The PPU's of index ≤ 1 fetch a value from global memory, possibly with indirect addressing through a local register.

II (Compute phase) The PPU's of index ≤ 1 fetch values from local memory and execute a "compute-"step, and store the result in local memory.

III (Write phase) The PPU's of index ≤ 1 write a value to global memory, possibly with indirect addressing through a local register and resolving contention write-conflicts by giving priority to the processors with lowest index.

An APM can simulate a "parallel" instruction in the following manner. In phase I the APM first broadcasts the address of a local

register to the first 1 locations of the REQ vector and fills the remaining locations with ∞. ("∞" is a fictitious address that serves to void the actions with the locations that contain it.) By the technique of lemma 4.1. the corresponding requests are satisfied from LOC-M, and the results are stored in a new REQ vector. The resulting requests are then satisfied from GLOB. The values fetched from global memory are now stored into LOC-M, by following the same technique. Phases II and III are handled in exactly the same way. Note in phase II that the SIMDAG is an SIMD machine, and thus the "compute-" step is the same for all PPU's and can be simulated in O(1) vector instructions by the APM.

Let L(N) be the largest value accumulated during the SIMDAG computation on an input of size N. Each parallel instruction will require $O(\log^2 \max\{P+S, PQ\})$ steps in the APM and each step takes $O(\log \max\{P, Q, S, L\})$ time. For a T-time bounded SIM DAG one has $L=O(2^T)$ (see Goldschlager [9]). Thus each parallel instruction takes up to $O(T \log^2 \max\{P+S, PQ\})$ time on an APM. It follows that the complete simulation is carried out in $O(T^2 \log^2 \max\{P+S, PQ\})$ time. □

Corollary 4.3. Let T(n) be APM-countable. A T(N)-time bounded SIMDAG can be simulated on an APM in $O(T^4(N))$ time.

Proof.

The largest integer value that can be accumulated during a T(N)-time bounded SIMDAG computation has size $O(2^{T(N)})$ (cf. Goldschlager [9]). Thus P(N), Q(N), S(N) are bounded by $O(2^{T(N)})$. Now apply the bound from theorem 4.2. □

Conversely, time-bounded APMs can be simulated efficiently by SIMDAG's within similar bounds.

Theorem 4.4. Any T(N)-time and S(N)-memory bounded APM (with no multiplication or division instruction) can be simulated by a SIMDAG that uses $O(T^2(N)\log^2 S(N))$ time, S(N) processors with O(1)-bounded local memory, and O(S(N)) global memory.

Proof.

(For simplicity we use T and S to denote the relevant time and space bounds.) The random access memory of the APM is simulated directly by the SIMDAG's global memory. Scalar instructions of the APM are executed by the SIMDAG's CPU, which also contains the simulated accumulator (AC). The vector accumulator (VAC) of the APM is represented indirectly, by the row of (ordinary) accumulator registers of the PPUs. To access locations in global memory, we assume that all PPUs have reserved register 0 as "memory address register" (MAR) and register 1 as "memory buffer register" (MBR).

In the simulation of vector instructions we distinguish between unmasked and masked instructions. First consider unmasked vector instructions, e.g. VLOAD i,j. Assume that the CPU first "reads" i and j to resolve any indirect addressing and computes $l=[j]-[i]+1$ (the length of the vector). As a SIMDAG can only "instruct" initial segments of the PPUs, we must arrange that the vector is loaded into PPUs 0 through $l-1$ (which subsequently act as the VAC). To this end the CPU first broadcasts the value of [i] to all PPUs of index $\leq l-1$, and then broadcasts the instruction "add SIG(nature) and store the result in y_0" to the very same processors. Now every PPU of index i with $0 \leq i \leq l-1$ contains in its MAR the address of the i^{th} element of the vector in global memory. The parallel fetch is accomplished by letting the CPU broadcast the instruction "$y_1 \leftarrow x_{y_0}$" to all PPUs of index $\leq l-1$. When the (unmasked) vector instruction is a VADD, a VSUB, or another vector operation, the "argument" is fetched from global memory in the same way and the CPU subsequently broadcasts the corresponding "ADD", "SUB", or other instruction to the row of PPUs. (The accumulator values are assumed to have moved to, say, register 2 so as to clear the MBR in the PPUs.) When the (inmasked) vector instruction is a VSTORE a very similar procedure is followed, with the CPU broadcasting an instruction "$x_{y_0} \leftarrow y_1$" to all PPUs of index $\leq l-1$.

Next consider the simulation of a masked vector instruction, e.g. VADD i,j,p,q. Let the CPU compute l as above. Now the difficulty is to

prevent the SIMDAG from applying the vector instruction to the PPUs that are in "masked" positions. As the SIMDAG has no mechanism for conditional execution, we have to use an expensive intermediate simulation. First arrange that PPU_0 through PPU_{l-1} read both the argument vector and the mask, very much like before. Without loss of generality we may assume that the PPUs are fully described by four-field record of the type [SIG, maskbit, ac-value, argument]. Arrange that PPU_0 through PPU_{l-1} dump their record of values in consecutive order in a free area of global memory. (Note that $4l \leq 4S(N)$ locations suffice.) Now use a SIMDAG routine based on e.g. Batcher's algorithm to sort the records such that the records with maskbit 1 appear up front. In logarithmic time the CPU can determine the length l' of this front part, using binary search over the sorted records. Now arrange that PPU_0 through $PPU_{l'-1}$ read the records with maskbit 1, and let the CPU broadcast the desired instruction to these processors. Upon completion, let PPU_0 through $PPU_{l'-1}$ write the updated record values back into the stored records in global memory and sort the records back into the original order by SIG-value. Next let PPU_0 through PPU_{l-1} read their records back into local memory. (When desired a similar routine can be used to void the the argument-field in the records with maskbit 0 before all values are stored back.)

An unmasked vector instruction may cost a SIMDAG up to $O(\log S(N))$ extra time but the costs stay within the same order of magnitude as for the APM. A masked vector instruction will cost the SIMDAG $O(\log^2 S(N))$ extra steps of cost $O(\log \max\{S(N), L(N)\})$, where $L(N)$ is the largest value accumulated during the APM computation. As $L(N) = O(2^{T(N)})$ the bound of $O(T(N)\log^2 S(N))$ for simulating a masked vector instruction follows. Hence the total time bound of the simulation is $O(T^2(N) \log^2 S(N))$. □

Corollary 4.5. A $T(N)$-time bounded APM (without multiplication or division) can be simulated by a SIMDAG in $O(T^4(N))$ time.

From corollaries 4.3. and 4.5. one immediately concludes that APMs (without multiplication and division instructions) and SIMDAGs

are polynomially related.

Theorem 4.6. APMs (without multiplication and division instructions) belong to the "second" machine class.

In particular it follows that time-bounded APMs are polynomially related to space-bounded Turing machines and RAMs (the "parallel computation thesis"), just like SIMDAGs are.

Stockmeyer & Vishkin [19] discuss the relation between resource-bounded CRCW PRAMs (i.e., SIMDAGs) and various other models of parallel computation. In particular they prove a result attributed to Ruzzo & Tompa that states that for well-behaved functions $T(n)$ and $S(n)$ with $\log T(n) \leq S(n) \leq T(n)$ and $S(n) \geq \log n$, the class of languages accepted by CRCW PRAMs that use $T(n)$ time and $2^{O(S(n))}$ processors. The results carry over to resource bounded APMs, using theorems 4.2. and 4.4. As an example (cf. [19], corollary 2) it follows that context-free languages can be accepted in $O(\log^4 n)$ time on an APM. It is likely that the bound can be improved by a more direct algorithm.

In recent studies of parallelism considerable attention is given to the uniform circuit model (see Cook [4]). In the remainder of this Section we show that APMs can efficiently simulate circuits, provided a suitable representation of the circuits is given on input, A circuit is a directed acyclic graph with, say, N input nodes and M output nodes. Internal nodes must be reachable from input, and are assumed to have in-degree 1 or 2. We assume that every node computes one of a fixed number of primitive operations $f_1,\ldots,f_r$ (e.g. the boolean operations) that are simulated in $O(1)$ scalar instructions on an APM. The fan-out can be arbitrary.

Theorem 4.7. A (uniform) circuit of size $C(N)$ and depth $D(N)$ can be simulated by an APM in $O(D(N)\log^4 C(N))$ time and $O(C^2(N))$ memory.

Proof.

For technical reasons we assume that the given circuit is modified such that in a parallel (i.e., "level-by-level") elaboration of the circuit no node of degree 2 receives its arguments simultaneously. (This is achieved by inserting dummy nodes of degree 1 where needed, at the expense of increasing C(N) and D(N) by a factor of at most 2.) Represent the circuit by a C(N)xC(N) adjacency matrix, stored row-wise as a vector ADJ of size $C^2(N)$ in the APM memory. We assume the nodes ordered such that the input nodes are listed first, and the output nodes last. In addition we assume that r masks $F_1,\ldots,F_r$ of size C(N) are provided that identify the nodes in the listing which compute $f_1,\ldots,f_r$ respectively. The representation of the circuit is stored in O(1) steps of cost O(log C(N), using the VREAD instruction.

In order to simulate a computation by the circuit several vectors and masks are needed. Initialize vectors ARG1 and ARG2 of size C(N) to hold the first and second argument, respectively, of the j^{th} node in location j ($0 \leq j \leq C(N)-1$). Also use a vector VAL of size C(N) to hold the computed values at the nodes. The computation begins by vector-reading N input values into the first N locations of VAL. Let V be the set of nodes of the circuit. Define the following subsets of V:

V_0 = the set of input nodes,

$V_i = \{x \in V \mid$ the predecessors of x are $\in \bigcup_{0}^{i-1} V_j$ but x isn't$\}$ $(i>0)$,

$W_i = \{x \in V \mid \exists_{y \in V_i}$ there is an edge from y to x$\}$ $(i \geq 0)$.

The sets V_i are the "levels" of the circuit. W_i is the set of nodes that receive an argument from the computation in the i^{th} level. By assumption no node receives more than one argument at a time. The sets V_i and W_i $(i=0,1,\ldots)$ will be computed as masks of size C(N). In an additional mask we keep track of the nodes of indegree 2 that have received one of their two arguments. The i^{th} step of the simulation $(i \geq 0)$ now consists of the following phases:

I Compute W_i. If $W_i = \emptyset$ then the computation (hence: the simulation) halts, and the results are stored with the output nodes.

II Broadcast the values computed at level i to the nodes of W_i.

III Compute V_{i+1}.

IV "Evaluate" the nodes in level V_{i+1}.

We assume that the masks $X_i = \bigcup_{0}^{i} V_j$ and V_i are available from the preceeding step.

To implement phase I, observe that W_i consists of all nodes that are reached in one "step" from V_i. The information must be retrieved from ADJ by a suitable sequence of vector instructions. First build a vector AUX1 consisting of the mask W_i repeated C(N) times, using the technique of recursive doubling (cf. Lemma 3.2.). View AUX1 as a vector of size $C^2(N)$ that represents a C(N)xX(N) matrix stored row-wise. Apply an APM-routine to compute a vector AUX2, which represents the transpose of AUX1. Assuming wlog that C(N) is chosen to be a power of two, a nice technique of Stone [20] can be used based on log C(N) iterations of the perfect shuffle. One iteration consists of a selection of the even- and odd-indexed components respectively, followed by a Batcher sort to "move" out the intermediate zeros and a juxtaposition of the resulting, collapsed, even and odd vectors. Thus one iteration takes $O(\log^2 C(N)$ steps, of a cost of O(log C(N)) per step. The vector AUX2 consists of C(N) blocks of size C(N), with the j^{th} block consisting of all 1's or all 0's depending on whether the j^{th} node is or is not in W_i. Compute AUX3 ≡ AUX2 ∧ ADJ and "fold" AUX3 back into one block of size C(N) by or-ing all blocks using the technique of lemma 3.1. The resulting block is precisely W_i (as a mask). Phase I is seen to require $O(\log^4 C(N))$ time, using the logarithmic cost criterion.

In phase II the values computed at the nodes in V_i must be passed on. The values are assumed to be stored in the corresponding locations of the vector VAL. Use the method described above to compute a vector AUX2' consisting of C(N) blocks of size C(N), in which the j^{th} block consists of the value of the j^{th} node repeated C(N) times of the node belongs to V_i and 0's otherwise. By a masked VLOAD using ADJ as a mask, we obtain a vector AUX3' which contains in every "occupied" position the value that must be stored as argument with the

corresponding node. Fold AUX3' back into block AUX4 of size C(N) using the technique of Lemma 3.1. Because of the technical assumption at the beginning of the proof, there is no clash of values during the folding. Now store the values of AUX4 into the corresponding locations of ARG1 or ARG2. (Use W_i and the mask that kept track of the preceding store's into nodes of degree 2 to decide which values to store in ARG1 and which in ARG2. Update the latter mask for use in the next step.) Phase II requires $O(\log^4 C(N))$ time as well.

Phase III is easy, after having computed W_i. Clearly V_{i+1} is computed as $V_{i+1} = W_i \wedge (\neg X_i)$, which takes the APM O(1) steps of cost O(log C(N). For later use we also compute $X_{i+1} = X_i \wedge V_{i+1}$. Phase IV is somewhat cumbersome because the nodes in V_{i+1} can compute different functions. By using the masks F_1 through F_r one can divide the nodes of V_{i+1} into $r = O(1)$ groups and for each group store the ARG1 values into the VAC and execute the corresponding (simulated) vector instructyion with the ARG2 values in memory. Using the same mask the result values are transfered into the corresponding locations of VAL. Phase IV also costs O(1) steps of cost O(log C(N)) assuming, as we may, that the accumulated values in the coordinates remain bounded.

It follows that the complete simulation of the circuit on an APM requires $O(\log^4 C(N))$ time per level, hence $O(D(N)\log^4 C(N))$ time total. □

It is likely that the time-bound in theorem 4.7. can be improved, but the result leads to a number of useful conclusions as it is. We also note that theorem 4.7. assumes that a "coding" of the circuit is presented on the input. In a further analysis one might want the circuit to be "APM-constructable".

The study of the uniform circuit model (cf. Cook [4]) has emphasized Boolean circuits of polynomial size and poly-logarithmic depth.

Definition. NC^k is the class of all problems that can be solved by

(uniform) circuits with $C(N) = N^{O(1)}$ and $D(N) = O(\log^k N)$.

Ruzzo [15] has shown that NC^k is precisely the class of problems solved by alternating Turing machines in $O(\log^k N)$ time and $O(\log N)$ space. From theorem 4.7. we conclude:

Corollary 4.8. The problems in NC^k can be computed by an APM in $O(\log^{k+4} N)$ time, using a polynomial-size bounded memory.

(Note that the logarithmic cost criterion is used in deriving the timebound.) The classes NC^1 and NC^2 contain many familiar numeric and non-numeric problems, such as matrix inversion and minimum spanning tree construction in undirected connected graphs (see Cook [4]).

5. Extensions of the basic APM model. The APM-model as presented in Section 2 was based on the simplest primitives as can be observed in present-day supercomputers. We have emphasized the processing capabilities of a vector machine, to facilitate the study of "vectorized" algorithms. Essential are the fast transfer of vectors to and from specified areas in memory, and the fast execution of vector instructions. In practice the assumptions of the model are approximated by the use of "parallel memories" for vector-storage and -retrieval (see van Leeuwen & Wijshoff [24]) and highly pipelined processors and vector buffers.

Compared to existing vector machines (see Hockney & Jesshope [11], Hwang et. al. [12]) several, more complex vector instructions were omitted from the APM instruction set that can be very attractive from the programmer's point of view. We list the following, typical instructions as examples:

(a) testing whether all components of a vector are zero,
(b) loading a vector-constant of specified length into the vector accumulator,
(c) summing a vector (VSUM),
(d) computing the maximum component of a vector (VMAX),
(e) sorting a vector (VSORT),

(f) compressing a vector according to a mask (VCOMPRESS), i.e., the construction of an new vector by simply ommiting the masked components,

(g) "masked merging" of two vectors (VMMERGE), i.e., the merge of two vectors in which the components of the first are written into the 0-position of the mask and the components of the second into the 1-positions of the mask.

The reason why instructions of this sort are often found in the repertoire of current vector computers is simply the efficiency and ease of implementation on the underlying hardware. The choice of instructions is further determined by the presumed needs of the programmar of vectorized (numerical) software, for increased flexibility and productivity.

With the emphasis on the needs of large-scale scientific computing (as exemplified in e.g. computational physics), it is clear that vector computers do not provide instructions that are primarily aimed at non-numeric computations. Yet a variety of vector instructions can be formulated which would facilitate the programming of e.g. combinatorial or graph-theoretic (parallel) algorithms. We list a few examples:

(h) computing the rank of a specified element in a vector (when viewed as a set),

(i) determining whether a given scalar occurs in a vector,

(j) computing the position of a given scalar in a vector (take the first occurence when the scalar occurs more than once),

(k) determining the element of a specified rank in a vector.

Every set of reasonable vector primitives gives a new starting point for studying the complexity of "vectorized" algorithms to perform certain computational tasks. The "instructions" (h) through (k) would enable an implementation of priority queues dictionaries and other dynamic structures (cf. Aho, Hopcroft, and Ullman [1]) in O(1) "time".

One easily verifies that the instructions (a) through (k) can be implemented on the basic APM in $O(\log N)$ or $O(\log^2 N)$ steps. In a computation of time T(N) using the logarithmic cost criterion, the

instructions would only lead to a "polynomial" increase in the simulated computing time and the extended APM is still "second class" (cf. Section 2, Section 4).

6. Conclusion. We presented a new model of parallel computation called the "array processing machine" (APM), which is intended to be the RAM-equivalent of present-day supercomputer designs. The model facilitates a complexity analysis of vectorized numeric and non-numeric algorithms. It was shown that APMs (without multiplication and division instructions) belong to the "second" machine class (cf. van Emde Boas [22]), and thus time-bounded APMs are polynomally related to space-bounded sequential computers. Feasible parallel computability is thus translated into poly-logarithmic time computability on an APM using polynomial-sized memory.

References

[1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] Chandra, A.K., D.C. Kozen, and L.J. Stockmeyer, Alternation, J. ACM 28(1981) 114-133.

[3] Chazelle, B., and L. Monier, Unbounded hardware is equivalent to deterministic Turing machines, Theor. Comput. Sci. 24(1983) 123-130.

[4] Cook, S.A., the classification of problems which have fast parallel algorithms, Proc. FCT'83, Springer Lecture Notes in Comput. Sci. 158(1983) 78-93.

[5] Cook, S.A., and R.A. Reckhow, Time-bounded random access machines, J. Comput. Syst. Sci. 7(1973) 354-375.

[6] Engeler, E., Introduction to the theory of computation, Acad. Press, New York, NY, 1973.

[7] Flynn, M.J., Some computer organisations and their effectiveness, IEEE Trans. Comput. C-21(1972) 948-960.

[8] Fortune, S., and J. Wyllie, Parallelism in random access machines, Proc. 10th ACM Sympos. Theory of Comput., San Diego, 1978, pp. 89-94.

[9] Goldschlager, L.M., A universal interconnection pattern for parallel computers, J. ACM 29(1982) 1073-1086.

[10] Hartmanis, J., and J. Simon, On power of multiplication in random access machines, Proc. 15th Ann. IEEE Sympos. Switching and Automata Theor., New Orleans, 1974, pp. 13-23.

[11] Hockney, R.W., and C.R. Jesshope, Parallel computers, Hilger, Bristol, 1981.

[12] Hwang, K., S.P. Su, and L.M. Ni, Vector computer architecture and processing techniques, in: M.C. Yovits (ed.), Advances in Computers, vol 20, Acad. Press, New York, NY, 1981, pp. 115-197.

[13] Knuth, D.E., The art of computer programming, vol3: sorting and searching, Addison-Wesley Publ. Comp., Reading, Mass., 1975.

[14] Pratt, V.R., and L.J. Stockmeyer, A characterization of the power of vector machines, J. Comput. Syst. Sci. 12(1976) 198-221.

[15] Ruzzo, W.L., On uniform circuit complexity, J. Comput. Syst. Sci. 22(1981) 365-383.

[16] Savitch, W.J., Parallel random access machines with powerful instruction sets, Math. Syst. Theor. 15(1982) 191-2210.

[17] Savitch, W.J., and M.J. Stimson, Time-bounded random access machines with parallel processing. J. ACM 26(1979) 103-118.

[18] Schorr, A., Physical parallel devices are not much faster than sequential ones, Inf. Proc. Lett. 17(1983) 103-106.

[19] Stockmeyer, L.J., and U. Vishkin, Simulation of parallel random access machines by circuits, SIAM J. Comput. 13(1984) 409-422.

[20] Stone, H.S. Parallel processing with the perfect shuffle, IEEE Trans. Comput. C-20(1971) 153-161.

[21] Stone, H.S., An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, J. ACM 20(1973) 27-38.

[22] van Emde Boas, P., The second machine class: models of parallelism, in: J. van Leeuwen, and J.K. Lenstra, (eds.), Parallel computers and computations, CWI Syll., Centre for Mathematics

and Computer Science, Amsterdam, 1985, pp - (to appear).

[23] van Leeuwen, J., Parallel computers and algorithms, Techn. Rep. RUU-CS-83-13, Dept. of Computer Science, University of Utrecht, Utrecht, 1983.

[24] van Leeuwen, J., and H.A.G. Wijshoff, Data mappings in large parallel computers, in: I Kupka (ed.), GI-13 Jahrestagung, Informatik Fb 73, Springer Verlag, Berlin, 1983, pp. 8-20.

[25] Wiedermann, J., Parallel Turing machines, Techn. Rep. RUU-CS-84-11, Dept. of Computer Science, University of Utrecht, Utrecht, 1984.

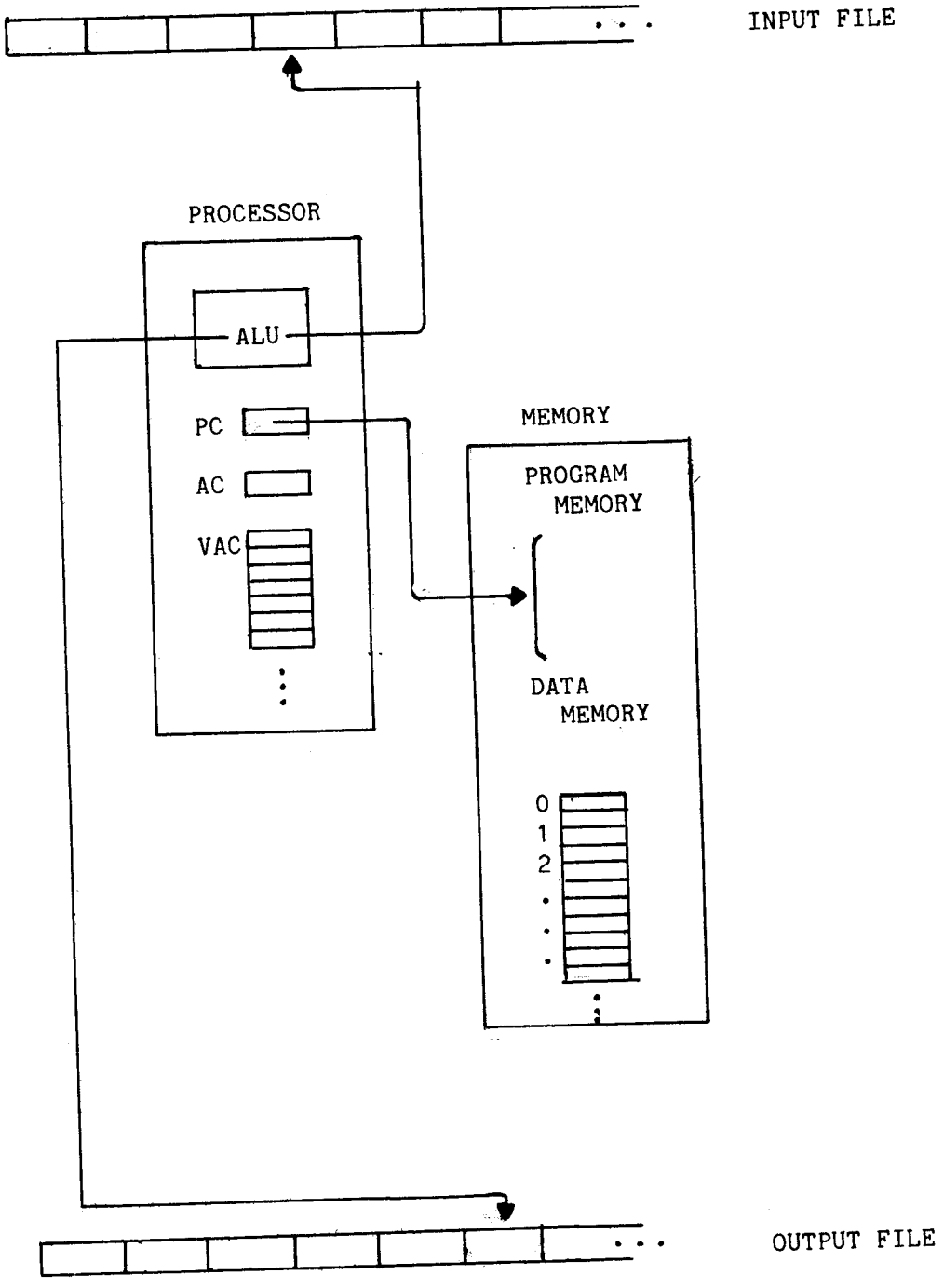

Figure 1. The APM model.

type	op-code	address	effect
I/O	READ	operand	read a (next) value from input into the specified address
	WRITE	operand	write the specified operand to output
transfer	LOAD	operand	copy the operand into the AC
	STORE	operand	copy the contents of the AC into the specified address
arithmetic	ADD	operand	add operand to the AC
	SUB	operand	subtract the operand from the AC
	MULT	operand	multiply the AC by the operand
	DIV	operand	integer-divide the AC by the operand
branching	JUMP	label	goto label
	JGTZ	label	if the AC contains a value > 0, then goto label
	JZERO	label	if the AC contains 0, then goto label
	HALT		end the program

Figure 2. Scalar instructions of the APM.

```
      {Aa, Na, Ma, and MIDa are reserved addresses}
      LOAD Na
      {if [AC] = 1 then jump to exit}
      SUB = 1
      JZERO exit
      ADD = 1
loop: {if [AC] is odd, we first add A[N-1] to A[0]..}
      DIV = 2
      MUL = 2
      SUB Na
      JZERO even
      LOAD Aa
      ADD Na
      SUB = 1
      STORE Ma
      LOAD *Aa
      ADD *Ma
      STORE *Aa
      {.. and effectively reduce the vector length by 1}
      LOAD Na
      SUB = 1
      STORE Na
even: DIV = 2
      {store A[0..M-1] into the VAC..}
      STORE Ma
      SUB = 1
      ADD Aa
      STORE MIDa
      VLOAD *Aa, *MIDa
      {.. and add A[M..N-1] to it}
      ADD = 1
      STORE MIDa
      ADD Ma
      VADD *MIDa, *[AC]
      LOAD MIDa
      SUB = 1
      STORE MIDa
      VSTORE *Aa, *MIDa
      {now set N to M and repeat}
      LOAD Ma
      STORE Na
      JUMP loop
exit: {write Aa to output or desired register}
      HALT
```

Figure 4. Summing a vector

```
n := N;
while n>1 do
   begin
      if n is odd then (A[0] := max {A[0], A[n-1]}; n := n-1);
      m := n div 2;
      α := mask of A[0..m-1] < A[m..n-1];
      A[0..m-1] := A[m..n-1] masked by α;
      n := m
   end;
max := A[0];
```

Figure 5. Computing the maximum element

```
zeros[0..2N-1] := mask of A[0..2N-1] * A[0..2N-1];
α[0..2N-1] := mask of A[0..2N-1] - A[0..2N-1];
n := N;
while n ≥ 1 do
   begin
      β[0..2N-n-1] := α[n..2N-1];
      β[2N-n..2N-1] := zeros [0..n-1];
      {thus β equals the left-shift of α over n positions followed by 0^n}
      α[0..2N-1] := mask of α[0..2N-1] * β[0..2N-1];
      {α is the mask (0^n 1^n)^N/n}
      LOW[0..2N-n-1] := A[0..2N-n-1];
      HIGH[0..2N-n-1] := A[n..2N-1];
      γ[0..2N-n-1] := mask of LOW[0..2N-n-1] > HIGH[0..2N-n-1];
      AUX[0..2N-n-1] := LOW[0..2N-n-1];
      LOW[0..2N-n-1] := HIGH[0..2N-n-1] masked by γ;
      HIGH[0..2N-n-1] := AUX[0..2N-n-1] masked by γ;
      A[0..2N-n-1] := LOW[0..2N-n-1];
      A[n..2N-1] := HIGH[0..2N-n-1] masked by α[n..2N-1];
      n := n div 2
   end;
{A in sorted order}
```

Figure 6. Merging two sorted N-vectors.

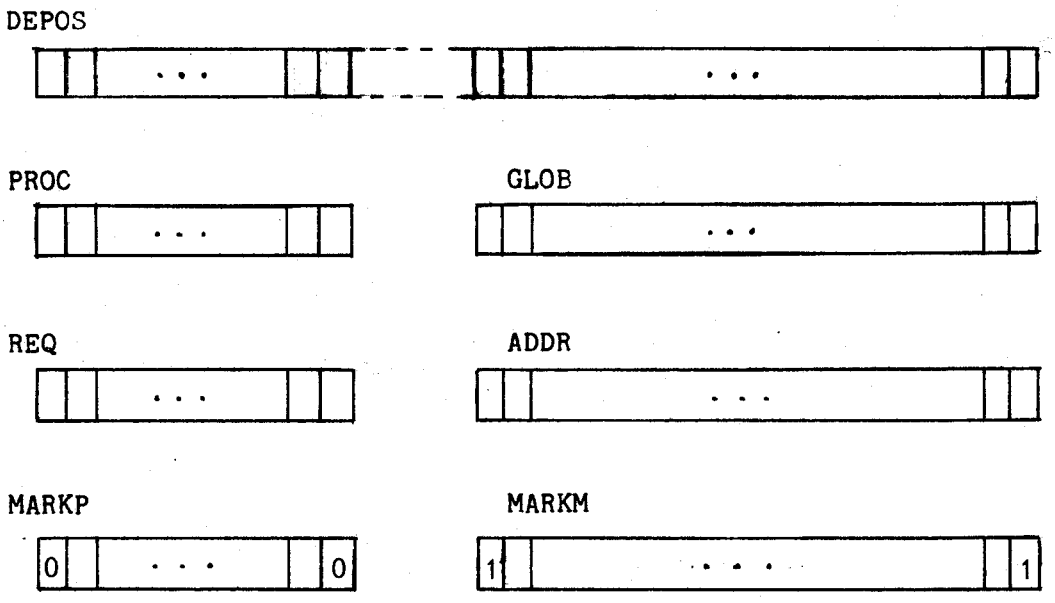

Figure 7. Simulating global memory.