Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.T. Tromp, P. van Emde Boas

Associative storage modification machines

Computer Science/Department of Algorithmics & Architecture Report CS-R9014 May

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Associative Storage Modification Machines

John Tromp
Centrum voor Wiskunde en Informatica
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: tromp@cwi.nl

Peter van Emde Boas*
Departments of Mathematics and Computer Science
University of Amsterdam
Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands
Email: peter@fwi.uva.nl

May 1, 1990

Abstract

We present a parallel version of the storage modification machine. This model, called the Associative Storage Modification Machine (*ASMM*), has the property that it can recognize in polynomial time exactly what Turing machines can recognize in polynomial space. The model therefore belongs to the Second Machine Class, consisting of those parallel machine models that satisfy the parallel computation thesis. The Associative Storage Modification Machine obtains its computational power from following pointers in the reverse direction.

1985 AMS(MOS) Subject Classification: 68Q05, 68Q10, 68Q15.
CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.
Keywords and Phrases: Machine Model, Storage Modification Machine, Pointer Machine, parallel, complexity classes, PSPACE, Second Machine Class, simulation.
Note: This paper will be submitted for publication elsewhere.

1 Introduction

The Storage Modification Machine (*SMM*) is a machine model introduced by Schönhage in 1977 [16]. The model has its predecessor in the Kolmogorov-Uspenskii machine (*KUM*) [10]. Schönhage advocates his model as *a model of extreme flexibility*.

The model resembles the Random Access Machine (*RAM*) [1] as far as it has a stored program and a potentially infinite memory structure where it stores its data. Whereas the *RAM* uses an infinite sequence of storage registers, each capable of storing an arbitrarily large integer, the *SMM* operates on a directed graph by creating nodes and (re)directing pointers. The main difference between the *SMM* and the *KUM* is that the *KUM* operates on undirected instead of directed graphs.

We can approximately model an *SMM* by a Pascal program, where the directed graph is described using a data type representing nodes which is defined as a *record* of pointers to nodes[1]:

*also CWI-AP6, Amsterdam

[1]In Pascal ^T means 'pointer to T'; a value of this type is the address of an object of type T. Indirection through a pointer is written as p^, which refers to the object at which p points.

```
type pointer = ^node;
     node    = record a,b: pointer end;
var  head    : pointer;
```

The tuple of nodes pointed at may be viewed as the *value* of a node. In contrast with Pascal, pointers are not allowed to be *nil* or undefined; they must always point to some node. The (finite) set of pointer names, in the example $\{a, b\}$, is called the *alphabet* of *directions*, denoted Δ. The pointers in the graph are labeled with the elements of Δ such that each node in the digraph has, for each direction $\delta \in \Delta$, exactly one outgoing δ-pointer. Hence the graph has regular outdegree $|\Delta|$. In the Pascal equivalent of an *SMM*, all data must be addressed from the single and only variable head, with expressions like `head^.b^.a^.b^.b^.a`. Similarly, the *SMM* addresses its storage with words (strings) over Δ, like *babba*. The *SMM* model doesn't distinguish a special 'head' pointer, but rather the node at which the conceptual head points. This node, which can change dynamically, is called the *center*, and is the one addressed by the empty word, ϵ. Other nodes are addressed by following pointers starting from the center.

It has been established that from the perspective of computational complexity theory the *SMM* (if equipped with the correct space measure [12, 21]) is computationally equivalent to the other standard sequential machine models like the Turing machine and the *RAM*. This equivalence amounts to the fact that these models simulate each other with polynomially bounded overhead in time and constant factor overhead in space, thus satisfying the so-called *invariance thesis* [17, 22].

For most sequential models there have been proposed parallel machine models based on the classical sequential version. For the Turing machine Savitch [15] has proposed a parallel version based on parallel recursive branching; a model based on nondeterministic forking on a shared set of tapes was described by Wiedermann [24], but this model turns out to be polynomially equivalent in time and space with the standard sequential devices. The richness of parallel models based on the *RAM* is even much greater, which makes it hard, if not impossible to refer to a small set of representative models. There are models based on shared memory and alternative models based on local storage and message passing. Hybrid combinations occur as well. Within each class there exist more refined distinctions like the resolution strategy for resolving write conflicts in shared memory models, the available arithmetic instructions and the mechanism for restricting the number of processors activated during a computation. Moreover, there exist sequential models which become computationally equivalent to parallel models due to their power to create and manipulate exponentially large values in a linear number of steps in the uniform time measure. Also, by exploiting the alternating mode of computation [5], some standard sequential devices become computationally equivalent to the parallel machines.

For a more detailed survey of parallel models I refer to [20, 22]. For the purpose of the present paper it suffices to give some impression of the overall landscape of parallel machine models.

It turns out that most parallel models proposed in the literature belong to the so-called *Second Machine Class* consisting of machine models which obey the *Parallel Computation Thesis*. This thesis expresses that the class of languages recognized in nondeterministic polynomial time on the parallel device is equal to the class *PSPACE* of languages recognized in polynomial space on a sequential device. Conversely all languages in *PSPACE* are recognized in deterministic polynomial time on the parallel machine.

Not all parallel models obey the above parallel computation thesis. Some weak models turn out to be polynomial time equivalent to the sequential models (the parallel Turing machine proposed by Wiedermann [24] being a typical example). Other models, like the *P–RAM* presented by Fortune and Wyllie [7] deviate from the thesis by recognizing exponentially time bounded languages in polynomial nondeterministic time on the parallel device; some parallel devices even recognize arbitrary languages in constant time [13]. The second machine class therefore represents a frequently occurring version of the power of uniform unrestricted parallelism rather than the union of all possible parallel machine models. Second machine class members can be characterized as providing the right mixture of exponential growth potential together with the proper degree of uniformity. The exponential growth potential is required for the implementation of the transitive closure algorithm on a

directed graph of exponential size (which models the computation graph of some *PSPACE*-bounded machine), or the direct solution of the *PSPACE*-complete problem *QBF* in polynomial time. The uniformity is required for performing the simulation of a polynomial-time computation of the nondeterministic version of the parallel machine in polynomial space. See [22] for more details on the standard strategies for proving membership in the second machine class.

In this paper we propose (as far as we know for the first time) a parallel version of the storage modification machine which belongs to the second machine class. To our knowledge few parallel versions of pointer machines have been investigated in the complexity theory literature. The earliest reference known to us concerns a parallel version of the Kolmogorov-Uspenskii machine which was proposed by Barzdin [2, 3]. This machine operates like an irregular cellular array of finite state automata in a graph which is dynamically changed by the individual nodes interacting with their neighbourhood. A single computation step resembles a parallel rewrite step in a graph grammar derivation. In this model all nodes are active in every computation step; if their neighborhood matches the pattern required by the instruction the node will transform its environment. The Hardware Modification Machine (*HMM*) introduced by Dymond and Cook [6] behaves in a similar way. This model indeed has been investigated for its complexity behavior. From Lam and Ruzzo [11] it follows that the machine is equivalent with constant factor time overheads with a restricted version of the *P–RAM* of Fortune and Wyllie. From this result one can observe that the *HMM* represents another example of the class of devices which are located beyond the second machine class - its nondeterministic version accepts *NEXPTIME* in polynomial time.

The computational power of our *ASMM* model originates from the possibility of traversing pointers in their *reverse* order. By using reverse directions, an *ASMM* can address, from a given node x, all the nodes that are associated with x by pointing to x (hence the name[2]). More than one node can be reached on a path by traversing pointers in the reverse direction. Note that at this point it is crucial that we have based ourselves on the *SMM* rather than the older *KUM* model; in an undirected graph traversing pointers in the reverse direction makes no sense.

As in the standard *SMM* model the finite control accesses the storage structure by means of a single center node. The power of traversing reversed pointers is used only in two types of instructions: the *new* and the *set* instruction. The first argument of the above two instructions is a path which now may contain reverse pointers. This path therefore no longer denotes a single node but a set of nodes (which in fact may be empty). The action described by the instruction now will be performed for all nodes in this set in parallel. The second argument of the *set* instruction is required to be a path consisting of forward pointers only; it therefore always denotes a single node. Therefore the action performed by the two instructions above is deterministic.

Our model may be considered to be a member of the class of sequential machines which operate on large objects in unit time and obtain their power of parallelism thereof. Other models of this character are the vector machines of Pratt and Stockmeyer [14], the *MRAM* proposed by Hartmanis and Simon [9] and simplified by Bertoni et al. [4], and also the *EDITRAM* presented by Stegwee et al. [18, 22].

Following [22] we denote the class of languages accepted in polynomial time by the *ASMM* model by *ASMM–PTIME*. The class of languages accepted in polynomial time by nondeterministic *ASMM* devices is denoted by *ASMM–NPTIME*. The class *PSPACE* as indicated above, denotes the class of languages recognized in polynomial space on a Turing machine. The fact that the *ASMM* is a true member of the second machine class is now expressed by the equality:

ASMM–PTIME = ASMM–NPTIME = PSPACE

In the proof of this equality we use the well known *PSPACE*-complete problem:

QUANTIFIED BOOLEAN FORMULAS (*QBF*) [19] :

QUANTIFIED BOOLEAN FORMULAS:
INSTANCE: A formula of the form $Q_1x_1 \ldots Q_nx_n[P(x_1, \ldots, x_n)]$,
where each Q_i equals $\forall$ or $\exists$, and where $P(x_1, \ldots, x_n)$

[2]compare with *content-addressable associative memory*

is a propositional formula in the boolean variables $x_1, \ldots, x_n$.

QUESTION: does this formula evaluate to *true*?

2 The *SMM* and the *ASMM* models

Our *ASMM* model is based on the Storage Modification Machine as introduced by Schönhage in 1970 [16]. The *SMM* model resembles the *RAM* model as far as it has a stored program and a similar flow of control. It has a single storage structure, called a Δ-*structure*. Here Δ denotes a finite alphabet consisting of at least two symbols. We denote the reverse of a direction $a \in \Delta$ as $\bar{a}$. Furthermore, $\bar{\Delta} = \{\bar{\alpha} | \alpha \in \Delta\}$ is the set of reverse directions and we let $\tilde{\Delta} = \Delta \cup \bar{\Delta}$.

A Δ-structure X is a finite directed graph each node of which has $k = |\Delta|$ outgoing edges which are labeled by the k elements of Δ. In Schönhage's formalization, a Δ-structure is a triple (X, c, p), where X denotes the finite set of nodes, $c \in X$ is the *center*, and $p : X \times \Delta \rightarrow X$ is the pointer mapping; $p(x, \alpha) = y$ means that the α-pointer from x goes to y.

There exists a map p^* from Δ^* to X defined as follows: For the empty string ϵ one has $p^*(\epsilon) = c$, and otherwise $p^*(wa) = p(p^*(w), a)$ is the end-point of the a-labeled pointer starting in $p^*(w)$.

The map p^* does not have to be surjective. Nodes which can not be reached by tracing a word w in Δ^* starting from the center c will turn out to play no subsequent role during the computations of the *SMM*. In the *ASMM* model pointers can be traversed in the opposite direction, and therefore these nodes no longer can be disregarded as being garbage.

The storage of an *SMM* or an *ASMM* is a dynamically changing Δ-structure, which initially consists of a single node, the center. The *ASMM*'s operation is described by a *program*, which is a finite sequence of *labels* and *instructions*. Labels can be used in control flow statements ; they should occur exactly once in case the machine is deterministic. Nondeterminism is introduced by allowing multiple occurrences of the labels referred to in jump or conditional jump instructions. In the text below we separate labels and instructions by a colon, whereas instructions are ended by semicolons.

The instruction repertoire of the *SMM* and the *ASMM* includes the *common* instructions (the λ's are labels and $\beta \in \{0, 1\}$)

input λ_0, λ_1;
output β;
goto λ;
halt;

The *input* instruction reads an input bit β and transfers control to λ_β. The other instructions are straightforward.

Furthermore there exist three *internal* instructions which operate on memory - in this case a Δ-structure X. For the *SMM* the arguments in these instructions are strings over Δ. For the *ASMM* the single argument of *new* and the first argument of *set to* are strings over $\tilde{\Delta}$; the other arguments (second argument of *set to* and both arguments of the *if* instruction) are strings over Δ. All arguments are finite strings which are written literally in the program. We first describe their meaning for the *SMM*:

1. *new* W: creates a new node which will be located at the end of the path traced by W; if $W = \epsilon$ the new node will become the center; otherwise the last pointer on the path labeled W will be directed towards the new node. All outgoing pointers of the new node will be directed to the former node $p^*(W)$

2. *set* W *to* V: redirects the last pointer on the path labeled by W to the former node $p^*(V)$; if $W = \epsilon$ this simply means that $p^*(V)$ becomes the new center; otherwise the structure of the graph is modified.

3. *if* $V = W$ (*if* $V \neq W$) *then* $<$ *instr* $>$: depending on whether $p^*(V)$ and $p^*(W)$ coincide or not, the conditional instruction $<$ *instr* $>$ (conditional jump suffices) is executed or skipped.

In the *ASMM* model the Δ-structure can be addressed by words (also called *paths*) over the alphabet of normal and reverse directions $\tilde{\Delta}$. Every word $W \in \tilde{\Delta}^*$ addresses the (possibly empty) set of all the nodes reachable from the center by following the consecutive directions (reversed if barred) in W.

The notion of 'addressing' is formalized by the mapping $P : \tilde{\Delta}^* \rightarrow 2^X$, defined by:

$$\begin{aligned} P(\epsilon) &= \{c\} \\ P(W\alpha) &= \{p(x,\alpha) | x \in P(W)\} \\ P(W\bar{\alpha}) &= \{x | p(x,\alpha) \in P(W)\}. \end{aligned}$$

If $V \in \Delta^*$, then $P(V)$ is a singleton set and we will frequently abuse notation by confusing a node with a path addressing it—e.g. referring to the center as ϵ. A node x is said to be *directly* addressable if it is reachable from the center by normal (non-reversed) directions, i.e. $\exists\, V \in \Delta^* : P(V) = \{x\}$.

In order to facilitate the descriptions of the internal instructions, we define a mapping $Q : \tilde{\Delta}^* \rightarrow 2^X$, from a path to the set of nodes from which the last pointer on this path originates, by:

$$\begin{aligned} Q(\epsilon) &= \emptyset \\ Q(W\alpha) &= P(W) \\ Q(W\bar{\alpha}) &= P(W\bar{\alpha}). \end{aligned}$$

The *new* and *set* change the Δ- structure from (X, c, p) to (X', c', p') as follows:

new W;
Here, $W \in \tilde{\Delta}^*$ determines where new nodes are inserted. If $W = \epsilon$, then a new center c' is created such that $X' = X \cup \{c'\}$ and $p'(c', \delta) = c$ for all $\delta \in \Delta$. Otherwise, if $W = U\tilde{\alpha}$ ($\tilde{\alpha}$ is either α or $\bar{\alpha}$), then for every node $u \in Q(W)$ a new node x_u is created such that $X' = X \cup \{x_u | u \in Q(W)\}$, $p'(u, \alpha) = x_u$, $\forall \delta \in \Delta\; p'(x_u, \delta) = p(u, \alpha)$, and $c' = c$. All other pointers remain unchanged.

set W *to* V;
Here, $W \in \tilde{\Delta}^*$ determines which pointers are redirected to the node determined by $V \in \Delta^*$. If $W = \epsilon$, then $c' = P(V)$ becomes the new center. Otherwise, if $W = U\tilde{\alpha}$, then for every node $u \in Q(W)$, $p'(u, \alpha) = P(V)$ and $c' = c$. In both cases X' is the restriction of X to the nodes which are reachable from c'.

The third internal instruction is the *if* statement. Since both paths in this instruction consist of forward pointers only, the meaning of this instruction is equal for the *SMM* and the *ASMM*.

The *time complexity* we use is simply the number of instructions executed. We do not concern ourselves with the *space complexity*; see [12, 21] for a discussion of the space complexity of the *SMM*.

3 An illustration of the power of associativity

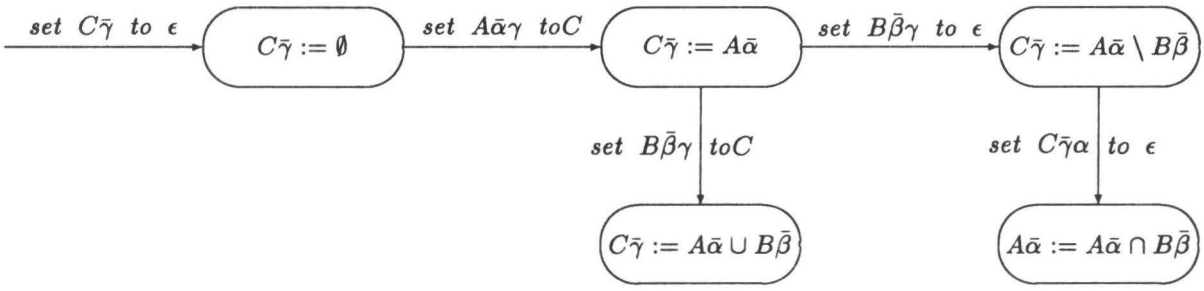
We demonstrate the power of the *ASMM* model by showing the capability to manipulate arbitrarily large sets in constant time.

The model allows the following natural representation of sets. If W is a word over Δ, and $\alpha \in \Delta$ a direction, then $P(W\bar{\alpha})$ is the set of all nodes having their α-pointer directed to the node $P(W)$.

Assume that our alphabet is $\Delta = \{A, B, C, \alpha, \beta, \gamma\}$ and that the A, B, and C-pointers from the center go to three different nodes $P(A)$, $P(B)$ and $P(C)$, none of which is the center. We will now consider the sets $P(A\bar{\alpha})$, $P(B\bar{\beta})$ and $P(C, \bar{\gamma})$ and see how the standard set operators can be applied to them by using appropriate *set to* instructions. We have chosen A, B and C to be directions so that the instructions with which we will implement the set operators cannot affect the addressing of the nodes $P(A)$, $P(B)$ and $P(C)$. As long as no such interference exists, we can generalize to the case where A, B and C are not elements of Δ but words over Δ.

The instruction *set* $A\bar{\alpha}\beta$ *to* B; has the effect of adding to $P(B\bar{\beta})$ the set $P(A\bar{\alpha})$, while removing from $P(C\bar{\beta})$ the nodes which are also in $P(A\bar{\alpha})$.

The figure below now shows how the standard set operators, shown as assignment statements in the boxes, can be implemented in terms of *set to* instructions. The center ϵ is used to direct pointers away from A or C.



The following program illustrates how in linear time a set $P(\bar{\alpha})$ of exponential size can be constructed (with a singleton alphabet):

new $\bar{\alpha}$;
set $\bar{\alpha}\bar{\alpha}$ *to* ϵ;
⋮
new $\bar{\alpha}$;
set $\bar{\alpha}\bar{\alpha}$ *to* ϵ;

Initially only the center exists, so all nodes point to the center. If at some point 2^k nodes exist, all of which point to the center, then after the *new* instruction, each of these 2^k nodes now points to one of 2^k newly created nodes, which again point to the center. Next the *set* instruction makes all 2^{k+1} nodes point to the center. Hence after k repetitions of these two instructions the size of the set $P(\bar{\alpha})$ has become 2^k.

In the next section we will see how these and similar constructions are used to process large amounts of data in parallel.

4 *PSPACE = ASMM–PTIME = ASMM–NPTIME*

The proof of membership in the Second Machine Class is usually split into two parts:

Lemma 1 *PSPACE* $\subseteq$ *ASMM–PTIME*

We prove this by sketching an *ASMM* which solves the *PSPACE*-complete problem *QBF* in polynomial time.

Lemma 2 *ASMM–NPTIME* $\subseteq$ *PSPACE*

We prove this by showing how to simulate t steps of a nondeterministic *ASMM* on a Turing machine using $O(t^2)$ space.

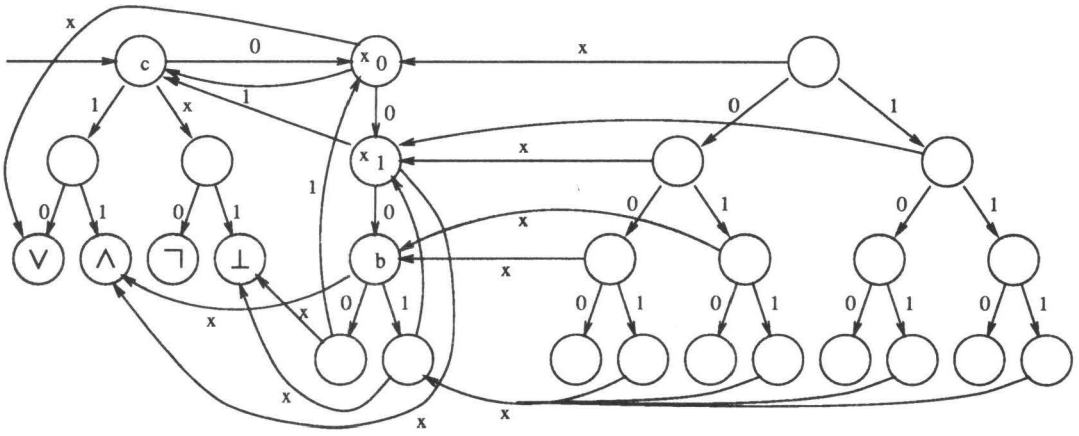


Figure 1: storage structure for $\exists x_0 \forall x_1 : x_0 \wedge x_1$

4.1 $QBF \in ASMM\text{-}TIME(n^2)$

The *ASMM* algorithm we present for solving *QBF* in polynomial time proceeds in 8 stages. Let $X = \{x_0, \ldots, x_{k-1}\}$ be the set of variables in the formula of length n, let $\Delta = \{0, 1, x\}$ and let c be the center. Basically, the algorithm expands the formula by rewriting the quantifiers, one by one, innermost first, as follows:

$$\forall x_i F(x_i) \Longrightarrow F(0) \wedge F(1),$$

$$\exists x_i F(x_i) \Longrightarrow F(0) \vee F(1).$$

The resulting, fully expanded formula, can be viewed as a tree. It consists of a complete binary tree T of depth k, with an instance of the formula body B rooted at each leaf of T. In each such instance, the variables are replaced by their truth values assigned to them along the path down to the leaf. The algorithm does little more than to build and evaluate this tree.

Figure 1 depicts the structure built for the example formula $\exists x_0 \forall x_1 : x_0 \wedge x_1$. The part on the right represents the expanded formula (where some of the x-pointers have been omitted for clarity). We now briefly summarize each of the 8 stages:

1. Build a list of nodes $c, x_0, x_1, \ldots, x_{k-1}, b$ linked through the 0-pointer. Using the 0, 1-pointers, build a representation of the formula body as a binary tree B rooted at b. The non-leaf nodes of B represent the connectives (and, or, not) while the leaves represent instances of variables.

2. Build a complete binary tree T of depth k using the 0, 1-pointers. For a node at depth i, its 0-subtree represents the case $x_i = 0$ and its 1-subtree the case $x_i = 1$.

3. Build 2^k copies of B rooted at the leaves of T.

4. For every leaf u of B representing an instance of x_i, let the 2^k copies of u direct their x-pointer to either u or c depending on the connective of u's parent and the value assigned to x_i.

5. For every non-leaf u of B, let the 2^k copies of u direct their x-pointer to either u or c depending on the connectives of u and its parent.

6. For every x_i, let the 2^i nodes of T at level i direct their x-pointer to either x_i or c depending on the quantifiers of x_i and x_{i-1}.

7. Evaluate all copies of B in parallel.

8. Evaluate T.

Note that by building a $|\Delta|$-ary tree of depth, say d, close to the center, $|\Delta|^d$ pointers become available to the machine for use as temporary/local pointer variables. For the purpose of traversing X and B we will use the otherwise unspecified paths $v, w \in \Delta^* \times \Delta \setminus \{x\}$. We don't want these paths to end with an x because we will use expressions like $v\bar{x}$, where it is undesirable to address a node along the path v.

The following invariant will hold throughout the execution:

$$\forall v \in X \cup B : \; v\bar{x} \subseteq C(v),$$

where $C(v)$ is the set of copies of v. For $v = x_i$, these are the 2^i nodes at depth i in T.

An explanation about the representation of truth values is in order here. A copy $u' \in C(u)$ of a node $u \in X \cup B$ can have its x-pointer directed to either u (u' is *active*) or c (u' is *passive*). This is done in a way which facilitates the evaluation of the parent of u. This parent is assigned a default value according to the table below. Now u' is active iff its value invalidates the default value of its parent, as shown in the table.

parent type	$\wedge$	$\vee$	$\neg$
parent default	1	0	1
active value	0	1	1

As an example, suppose a $\neg$ node $u' \in C(u)$ has a parent $\vee$ node. The parent gets a default value of 0, which is to be changed into a 1 iff either of its children evaluates to 1. Thus, 1 is the active value for u'. The value 0 is passive for u', since it agrees with the default 0 value of its $\vee$ parent. Since a $\neg$ node is 1 by default, u' is active by default, hence its x-pointer is initially directed to u.

The representation of the truth value at a node u therefore depends on the type of the logical connective associated to the parent of u in the tree. This holds also for the nodes in the tree T which are associated to the variables x_i. In this tree the copies of the variables have been treated as logical connectives according to the type of the quantifyer binding this variable.

In the algorithm above stages 1, 2 and 3 are used for building the tree; during stage 4 the truth values are assigned to all variable occurrences in the copies of B, and in stages 5 and 6 all intermediate nodes are given their default values. During the final two stages the entire tree is evaluated.

We next describe each of the above stages in some more detail.

In stage 1 the input is examined and used to construct a linearly sized list and tree representing the formula. We represent the type of a node $u \in X \cup B$ by directing its x-pointer to one of the special nodes $\vee, \wedge, \neg, \perp$. As mentioned before, these four symbols will also be used as paths addressing the nodes. The leaves of B are of type $\perp$ and have their 1-pointer directed to the appropriate x_i. Existentially quantified x_i have $type(x_i) = \vee$ and universally quantified x_i have $type(x_i) = \wedge$. In order to distinguish the nodes x_i from nodes in B, we link a node of type $\neg$ to its child with the 1-pointer, and have the x_i direct their 1-pointer to c. Since no node in B has its 1-pointer directed to the center, comparing vx with ϵ tells whether v adresses a node in X or in B.

In stage 2 the parallel power of the machine is used to build an exponentially large tree in linear time. This is achieved by the piece of code below:

new v;
set vx *to* 0;
set v *to* 0;
λ : *new* $v\bar{x}0$;
set $v\bar{x}0x$ *to* $v0$;
new $v\bar{x}1$;
set $v\bar{x}1x$ *to* $v0$;
set v *to* $v0$;
if $v1 = \epsilon$ *then goto* λ;

The construction of 2^k copies of B in stage 3 proceeds analogously. Note that by now all the leaves of T have their x-pointer directed to b. Traversing B in preorder, we do the following at each node v:

```
    if vx = ⊥ goto λ2;
    if vx = ¬ goto λ1;
    new vx̄0;
    set vx̄0x to v0;
λ1: if vx = ⊥ goto λ2;
    new vx̄1;
    set vx̄1x to v1;
λ2:
```

In stage 4, all the x-pointers in the copies of leaves of B are installed. Let w be a leaf of B ($wx = \bot$) with $w1 = x_i$ the variable it represents. We show how to install the x-pointers in all copies $C(w)$ of w. We assume that w has the active value 1. The case for 0 is analogous. The code fragment

```
    set v to 0;
λ : set v0x̄ to ϵ;
    if v ≠ w1 then set vx̄0x to v0;
    set vx̄1x to v0;
    set v to v0;
    if v1 = ϵ then goto λ;
```

ends with $b\bar{x}$ equal to the set of leaves of T which have 1, the active value, assigned to x_i. In a similar fashion we can traverse the path from b to w, to end up with the active nodes of $C(w)$ pointing to w and the passive ones pointing to c.

The next two stages, 5 and 6, prepare the evaluation by giving default values to copies of non-leaves of B and nodes in T. If the copies of a non-leaf node $w \in B$ are by default passive (e.g., when both w and its parent are of type $\vee$), then we can make $w\bar{x} = \emptyset$ by the single instruction

```
set wx̄ to ϵ;
```

Otherwise, if they are by default active, then we can make $w\bar{x} = C(w)$ by traversing the path from x_0 to w with v like in the previous stage. Since this procedure sets $v\bar{x} = C(v)$ for all v on the path to w, we must deal with the non-leaves of B in postorder. Note that this stage ends with $v\bar{x} = C(v)$ for all $v = x_i$. This is correct for the x_i which are active by default, i.e. those whose type differs from that of their parent x_{i-1}.

To give all non-leaves of T the correct default value, we must therefore reset the x-pointers from the passive x_i (having the same type as their parent) to the center, which is achieved by the following code:

```
    set v to ϵ;
λ : if vx = v0x then set v0x̄ to ϵ;
    set v to v0;
    if v01 = ϵ then goto λ;
```

We don't direct the x-pointer from the center to either $\wedge$ or $\vee$, hence the root of T will remain active. Now all that's left to be done is the evaluation itself. This is done bottom up—by a post-order traversal of B and then from x_{k-1} back to x_0. With the other cases being analogous, we restrict ourselves to the evaluation of an $\wedge$-node $w \in X \cup B$. Let v be its parent. The default value of w is 1, which is passive if v has type $\wedge$, or active if v has type $\vee, \neg$. The value of w should become 0 if either of its children has value 0, which is active for them. It should now be clear that the code fragment

```
    if vx = Λ then goto λ1;
    set w0x̄0̄x to ε;
    set w1x̄1̄x to ε;
    goto λ2;
λ1 : set w0x̄0̄x to w;
    set w1x̄1̄x to w;
λ2 :
```

evaluates node w. The technique used here is essentially the same as in section 3 for computing a union. Because of our symmetric representation, it works for both $\vee$ and $\wedge$.

When evaluation is complete, the root of T, r, will have its x-pointer directed to either x_0 or c. It need not even be directly addressable in order to turn this into an *if then* test. We can still check whether $0\bar{x}$ is empty or not by trying to modify a pointer from a directly addressable node, like x_0. We know that $p(x_0, x) \in \{\wedge, \vee\}$. The instruction

set $0\bar{x}xx$ *to* ϵ;

changes $p(x_0, x)$ to ϵ iff $0\bar{x}$ is nonempty, which is equivalent to $p(r, x) = x_0$. Recall from stage 6 that this is how $p(r, x)$ was initialized to its default value, 0 for $\vee$ and 1 for $\wedge$. Combining this information we obtain the value of the formula.

Regarding the time complexity, the most time-consuming stage is number 4, where for each leaf of B, both X and B are traversed, requiring at most n^2 steps. Hence the complete algorithm runs in quadratic time.

4.2 $ASMM\text{–}NTIME(t) \subseteq SPACE(t^2)$

The simulation which proves this inclusion is relatively straightforward and employs previously known methods [14, 9]. We can write down in polynomial space a trace of the computation containing information on the sequence of instructions executed. Since the machine being simulated is nondeterministic this trace is guessed. Next it is verified by means of a system of recursive procedures and some other arrays containing polynomially sized information that this trace indeed represents an accepting computation. The *if*, *new* and *set to* statements pose the main problems, since their impact on the Δ-structure requires repeated recomputations of the current state of the Δ-structure. In polynomial space we cannot explicitly store the possibly exponentially large Δ-structure of the ASMM-machine, so an implicit representation is called for. This will consist of three arrays, and three mutually recursive functions. The arrays are

1. *instr*$[i]$ holds the instruction executed at step i
2. *nodes*$[i]$ holds the number of nodes at time i
3. *center*$[i]$ holds the center at time i

The simulation starts at time 0 and has step i ($i \geq 1$) leading to time i. Each array is of length t, the number of steps to be simulated, and each array element fits in t bits since the number of nodes can at most double after each step. Every node will have a unique number, and the resulting ordering of nodes is used for numbering nodes created by a *new* instruction. More precisely, a *new* W; instruction at step i is simulated as follows:

If $W = \epsilon$, then *center*$[i]$ = *nodes*$[i-1]$ and *nodes*$[i]$ = *nodes*$[i-1] + 1$.

Otherwise, if $W = U\tilde{\alpha}$, then *center*$[i]$ = *center*$[i-1]$ and *nodes*$[i]$ = *nodes*$[i-1] + |Q(W)|$. Semantically, if $Q(W) = \{x_0 < x_1 < \ldots < x_{k-1}\}$, then at time i, $p(x_j, \alpha)$ = *nodes*$[i-1] + j$, for $j < k = |Q(W)|$.

For all other instructions, *nodes*$[i]$ = *nodes*$[i-1]$ and *center*$[i]$ = *center*$[i-1]$, except that the instruction *set* ϵ *tto* V; sets *center*$[i]$ to $P(V)$. In order to compute $P(V)$ and to simulate the *if* instruction, we use the following functions:

$p(x,\alpha,i)$ returns the number of the node $p(x,\alpha)$ at time i

$P(x,W,i)$ returns whether $x \in P(W)$ at time i

$Q(x,W,i)$ returns whether $x \in Q(W)$ at time i.

These functions satisfy the equations

$$\begin{aligned}
Q(x,\epsilon,i) &= \mathit{false} \\
Q(x,U\alpha,i) &= P(x,U,i) \\
Q(x,U\bar{\alpha},i) &= P(x,U\bar{\alpha},i) \\
P(x,\epsilon,i) &= (x == \mathit{center}[i]) \\
P(x,U\alpha,i) &= (\exists\ 0 \le y < \mathit{nodes}[i] : P(y,U,i) \wedge p(y,\alpha,i) == x) \\
P(x,U\bar{\alpha},i) &= P(p(x,\alpha,i),U,i) \\
p(x,\alpha,0) &= 0
\end{aligned}$$

which shows that they can be easily computed, apart from the case $p(x,\alpha,i)$ for positive values of i. The action of p in this case depends on the value of $\mathit{instr}[i]$, the only interesting values of which are *new* and *set*.

Consider first the case $\mathit{instr}[i] = \mathit{new}\ W$. If $x \ge \mathit{nodes}[i-1]$ then (using $Q(y,W,i)$) the difference $x - \mathit{nodes}[i-1]$ can be used to find the y in $Q(W)$ which 'generated' and now points to x (unless $W = \epsilon$, in which case $p(x,\alpha,i) = \mathit{center}[i-1]$). Now $p(x,\alpha,i) = p(y,\alpha,i-1)$. On the other hand, suppose $x < \mathit{nodes}[i-1]$. If $W = U\bar{\alpha}$ (i.e. α-pointers may have changed) and $Q(x,W,i-1)$, then x has generated $p(x,\alpha,i) = \mathit{nodes}[i-1] + |\{y < x | q(y,W,i-1)\}|$. Otherwise $p(x,\alpha,i) = p(x,\alpha,i-1)$.

Second and last, consider the case $\mathit{instr}[i] = \mathit{set}\ W\ \mathit{to}\ V$. If $W = U\bar{\alpha}$ and $Q(x,W,i-1)$, then $p(x,\alpha,i)$ is the unique y satisfying $P(y,V,i-1)$. Otherwise $p(x,\alpha,i) = p(x,\alpha,i-1)$.

These functions can easily be coded on a Turing Machine using recursion (stackframes). The recursion depth is bounded by ct, where c is a constant depending only on the maximum path length of the ASMM program. Each stackframe holds a return address and some node numbers and counters each of which fits in t bits. Together with the three arrays, space $O(t^2)$ suffices for the simulation of t steps of the ASMM.

5 Conclusion

Of all the parallel models which have been shown to belong to the Second Machine Class, the *ASMM* is the first to obtain its power from the use of associative addressing, thus making it an interesting addition to the realm of Second Machine Class devices. It provides another example that a small modification of a machine model can enforce a substantial increase of computational power. In [4] it was shown that this increase is provoked by adding multiplicative instructions to the unit-time standard *RAM* model. Similarly the *EDITRAM* model obtains its power by introducing a few edit operators which are available on most real life text editors anyhow. In the *ASMM* model it turns out that traversing pointers in a reverse direction is all we need for obtaining full parallel power. At the same time, the fact that the storage structure of the *ASMM* is manipulated by a finite program which interacts with the Δ-structure by means of a single center seems to be the main reason why the machine has not become too powerful. As shown by Lam and Ruzzo [11], a model where the nodes become independently active finite automata suffices for making the nondeterministic version more powerful than *PSPACE* (except for the unlikely case that *PSPACE* = *NEXPTIME*). This situation resembles the relation between the *SIMDAG* described by Goldschlager [8], where a single processor broadcasts its instructions to a collection of peripheral processors and the *P–RAM* model of Fortune and Wyllie [7] where the local processors are independent.

References

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] Barzdin', Ya. M., *Universal pulsing elements*, Soviet Physics-Doklady 9 (1965) 523–525.

[3] Barzdin', Ya. M., *Universality problems in the theory of growing automata*, Soviet Physics-Doklady 9 (1965) 535–537.

[4] Bertoni, A., Mauri, G. and Sabadini, N., *Simulations among classes of random access machines and equivalence among numbers succinctly represented*, Ann. Discr. Math. 25 (1985) 65–90.

[5] Chandra, A.K., Kozen, D.C. and Stockmeyer, L.J., *Alternation*, J. Assoc. Comput. Mach. 28 (1981) 114–133.

[6] Dymond, P.W. and Cook, S.A., *Hardware complexity and parallel computation*, Proc. 21st Ann. IEEE Symp. Foundations of Computer Science, 1980, pp. 360–372.

[7] Fortune, S. and Wyllie, J., *Parallelism in random access machines*, Proc. 10th Ann. ACM Symp. Theory of Computing, 1978, pp. 114–118.

[8] Goldschlager, L.M., *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach. 29 (1982) 1073–1086.

[9] Hartmanis, J. and Simon, J., *On the structure of feasible computations*, in Rubinoff, M. and Yovits, M.C. (Eds.), Advances in Computers, Vol. 14 , Acad. Press, New York, 1976, pp. 1–43.

[10] Kolmogorov, A.N. and Uspenskii, V.A., *On the definition of an algorithm*, Uspehi Mat. Nauk 13 (1958) 3–28 ; AMS Transl. 2nd ser. 29 (1963) 217–245.

[11] Lam, T.W. and Ruzzo, W.L., *The power of parallel pointer manipulation*, Proc. 1st Ann. ACM Symp. Parallel Algorithms and Architectures, 1989, pp. ??–??.

[12] Luginbuhl, D.R. and Loui, M.C., *Hierarchies and space measures for pointer machines*, Report UILU-ENG-88-2245, Department of Electr. Engin., University of Illinois at Urbana-Champaign, 1988.

[13] Parberry, I., *Parallel speedup of sequential machines: a defense of the parallel computation thesis*, SIGACT News 18, nr. 1, 1986, pp. 54–67.

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

[15] Savitch, W.J., *Recursive Turing machines*, Inter. J. Comput. Math. 6 (1977) 3–31.

[16] Schönhage, A., *Storage modification machines*, SIAM J. Comput. 9 (1980) 490–508.

[17] Slot, C. and van Emde Boas, P., *The problem of space invariance for sequential machines*, Inf. and Comp. 77 (1988) 93–122.

[18] Stegwee, R.A., Torenvliet, L. and van Emde Boas, P., *The power of your editor*, Report RJ 4711 (50179), IBM Research Lab., San Jose, Ca., 1985.

[19] Stockmeyer, L., *The polynomial time hierarchy*, Theor. Comp. Sci. 3 (1977) 1–22.

[20] van Emde Boas, P., *The second machine class 2: an encyclopaedic view on the Parallel Computation Thesis*, in: Rasiowa, H. (Ed.), Mathematical Problems in Computation Theory, Banach Center Publications, Vol. 21, Warsaw, 1987, pp. 235–256.

[21] van Emde Boas, P., *Space measures for storage modification machines*, Inf. Proc. Lett. 30 (1989) 103–110.

[22] van Emde Boas, P., *Machine models and simulations*, in: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, North-Holland Publ. Comp. 1990, to appear. Also: report ITLI-CT-89-02, Univ. of Amsterdam, Feb. 1989.

[23] Wagner, K. and Wechsung, G., *Computational Complexity*, Mathematische Monographien Vol. 19, VEB Deutscher Verlag der Wissenschaften, Berlin (DDR), 1986, also: Reidel Publ. Comp., Dordrecht, 1986.

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