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

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 computation thesis, complexity theory, pspace, second machine class, simulation.

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 that uses records of pointers to records to describe the directed graph[1]:

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

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. The graph thus has regular outdegree $|\Delta|$. To complete the analogy between an *SMM* and a Pascal program, the latter must be restricted to the use of only one variable; the pointer `head`. By repeated application of the Pascal `new` statement, the program can create an arbitrarily large data structure. This is addressed with expressions like `head^.b^.a^.b^.b^.a`. Similarly, the *SMM* addresses its storage with words (strings) over Δ, like *babba*. In the *SMM* model, there is only a conceptual head pointer—at any time, one node, call the *center*, is distinguished as the one from where addressing starts. Thus the centre, whose identity can change dynamically, is addressed by the empty

[1] In Pascal `^T` denotes the type '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.

word ϵ, and 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, 25], 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 we 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. In our reading the Parallel Computation Thesis entails the equivalence of deterministic and nondeterministic polynomial time on the parallel model. The models for which the thesis was originally formulated obey this more restricted thesis as well. And indeed those models for which nondeterministic polynomial time seems to exceed *PSPACE* nowadays are held to be more powerful.

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, and its equivalents [24, 25] 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].

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

(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*$\langle instr \rangle$: depending on whether $p^*(V)$ and $p^*(W)$ coincide or not, the conditional instruction $\langle instr \rangle$ (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 and reverse directions in W.

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

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

Note: It will often be convenient to give a name to an address path $V \in \Delta^*$. In the code fragments presented in this paper, we will use paths having such a name v as a prefix, in addition to fully explicit paths. This serves two purposes. First, fixed nodes that have been given descriptive names can be addressed by their name rather than some arbitrary path (we say that a node is fixed iff it has a constant address). Second, if we are using one of the pointers from a fixed node to traverse part of the graph, it can be given

a name that more closely resembles its function: that of a variable. We will use variable names without specifying which path they stand for, omiting the details of the creation of spare nodes to provide the required[3] pointers.

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 $Q : \tilde{\Delta}^* \to 2^X$, 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*.

[3] In the case of the *ASMM*, when we use an address like $v\bar{x}$ with v a variable name, it is desirable for v not to be an x-pointer, i.e. that the address that v stands for doesn't end with the direction x.

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*.

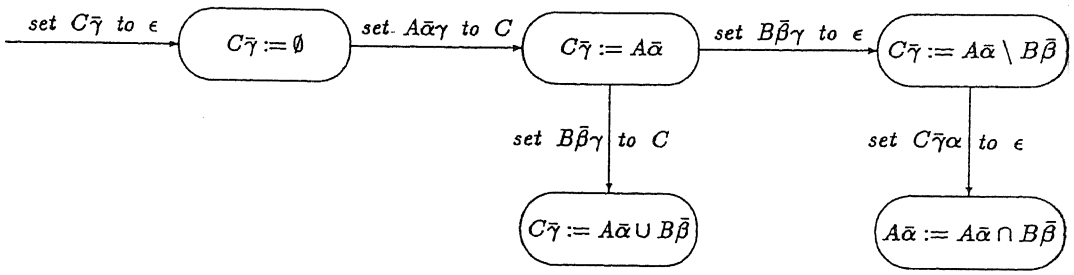
3 An illustration of the power of associativity

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 the instruction *set* $A\bar{\alpha}\beta$ *to* ϵ; removes from $P(A\bar{\alpha})$ the nodes which are also in $P(B\bar{\beta})$.

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;
$\vdots$
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.

For a clear understanding of the construction, it is important to distinguish between truth-values and their representation. Conceptually, the algorithm works with truth values, 1 (true) and 0 (false). The leaves of the 2^k copies of B are assigned truth values in the obvious way according to which variable they represent. The other nodes are assigned default truth values, which are 0 for a disjunction, and 1 for a conjunction or a negation (recall that the quantifiers have been transformed into disjunction and conjunction). Next, a bottom-up process repeatedly changes defaults, that are in disagreement with their evaluated children, into the correct evaluation. In order to facilitate this process, we use a mixed representation of truth values, as follows.

Call a node $u' \in C(u)$ *active* if its x-pointer is directed to u or *passive* if its x-pointer is directed to c. A truth value is represented by the active state iff that truth value disagrees with the default of its parent, and with the passive state otherwise. To summarize:

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. Note that by keeping the x-pointer of r directed to x_0, it is made active by default.

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 noted 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 $type(x_i) = \wedge$.

When traversing the list of x_i, the algorithm needs to be able to detect its end. Since the nodes in B already use the 1-pointer to point to their children (or single child in case of a $\neg$ node), we have the x_i direct their 1-pointer to the centre, and thus by comparing vx with ϵ can tell whether v addresses 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;	create r, root of T
set vx *to* 0;	classify it
set v *to* 0;	start X traversal
λ : *new* $v\bar{x}0$;	0-children for $C(x_i)$
set $v\bar{x}0x$ *to* $v0$;	classify in $C(x_{i+1})$
new $v\bar{x}1$;	1-children for $C(x_i)$
set $v\bar{x}1x$ *to* $v0$;	classify in $C(x_{i+1})$
set v *to* $v0$;	advance to x_{i+1}
if $v1 = \epsilon$ *then goto* λ;	repeat for all x_i

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 $vx = \perp$ *goto* λ_2;	do nothing at leaves
if $vx = \neg$ *goto* λ_1;	$\neg$ node has no 0-child
new $v\bar{x}0$;	create 0-child
set $v\bar{x}0x$ *to* $v0$;	classify in $C(v0)$
λ_1: *new* $v\bar{x}1$;	create 1-child
set $v\bar{x}1x$ *to* $v1$;	classify in $C(v1)$
λ_2:	

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 *to* 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

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

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 $instr[i]$, the only interesting values of which are *new* and *set*.

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

Second and last, consider the case $instr[i] = set\ W\ to\ V$. If $W = U\tilde{\alpha}$ 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 in 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 from introducing a few edit operators that are available on most real life text editors anyhow. In the *ASMM* model it turns out that traversing pointers in the reverse direction is all we need to obtain full parallel power. At the same time, the fact that the storage structure of the *ASMM* is manipulated by a finite program that 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 becomes equivalent with a restricted version of the *P–RAM* of Fortune and Wyllie. This 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.

Clearly there are other models which could serve as a parallelized version of the *SMM*. In our model the set-to instruction is rather limited. Since its second argument addresses a single node, it cannot be used for setting different pointers to different destinations. This severely limits the scope of proofs that our machine is indeed so powerful. A more conventional approach, based on the construction of the transition graph of a polynomial space bounded Turing machine, and the computation of its transitive closure by pointer jumping—as suggested by the referee—is rendered infeasible by the limitation of the set-to instruction. Overcoming this limitation would require a different flavour of set-to instruction. A natural possibility is to allow the conventional set-to instruction of the *SMM* to be executed in parallel with respect to many different 'centers', the latter being specified by a third argument which is a string in $\tilde{\Delta}$. This model has some drawbacks, however. One is the possibility of conflicts arising when a pointer must be set to one node when addressed through one center, and to another node when addressed through another center. Resolving this problem would probably detract from the elegance of the model, one of its prime features. Another problem is that it becomes harder to manage all the pointers, since there is no simple way in which to direct a bunch of them to some fixed node where they can be 'out of the way'. Thus it is not a strict generalization of our model, although it should be possible to simulate our set-to instruction with this new one by keeping around an extra direction to always point to the real center.

Another modification suggested by the referee amounts to replacing the flow-of-control nondeterminism by a nondeterministic data manipulation instruction. For example both arguments in the *set* W *to* V instruction may become strings over $\tilde{\Delta}$; the effect of this instruction is that each node addressed by W redirects its outgoing pointer towards one of the nodes addressed by V.

This instruction makes it possible to guess some truth value for an arbitrarily large set of propositional variables in a single instruction, and suggests a proof that *NP* is included in *ASMM–NLOGTIME* (assuming that the model also is upgraded to allow the input to be read in logarithmic time). Consequently this model would fail to be a member of the Second Machine Class; since the purpose of this paper is the design a version of the *SMM* belonging to the Second Machine Class we abstain from investigating this suggestion in more detail.

We like to use this opportunity to acknowledge for these suggestions of the referee and his other useful remarks which we have used in revising the manuscript.

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. 92–102

[12] Luginbuhl, D.R. and Loui, M.C., *Hierarchies and space measures for pointer machines*, Inf. and Comput., 1993, to appear; also: 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, pp. 1–66.

[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.

[25] Wiedermann, J., *Weak parallel machines; a new class of physically feasible parallel machine models*, I.M. Havel & V. Koubek (Eds.), proc. Mathematical Foundations of Computer Science 1992, Springer Lecture notes in Computer Science 629 (1992) pp. 95–111.