MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN – POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

THE SECOND MACHINE CLASS 2, AN ENCYCLOPEDIC VIEW ON THE PARALLEL COMPUTATION THESIS

PETER VAN EMDE BOAS

Department of Mathematics and Computer Science, University of Amsterdam, Amsterdam The Netherlands

1. Introduction

Computation theory knows a large variety of models of computing devices, or formal calculi for effective computation. This divergence has not lead to a large proliferation of computation theories due to the basic observation that the resulting formalisms become equivalent: each computation in formalism 1 can be simulated some way or another in formalism 2.

Complexity theory takes a more refined look at these formalisms in as far as that machine models become equipped with features formalizing time and space consumption of such computations. As a consequence assertions about complexity features are *machine dependent* by their origin. On the other hand complexity theory deals with some fundamental concepts which are generally believed to be *machine independent*. More specifically, when dealing with concepts like Polynomial Time, or Logarithmic Space computability it is generally assumed that it is irrelevant whether our theory is based on Turing machines or some more computer like model like the RAM.

In Complexity theory we are dealing with a number of machine models, including the various varieties of Turing machines, the RAM and RASP models with uniform or logarithmic time measures, and reference Machines, forming together a *Standard Class* or *First Machine Class*. The relation between two models within this standard class can be expressed by the *Invariance Thesis* which we introduced in [38]:

For each machine M_i of one type having running time T_i and storage use S_i one can find in any other type of machinery a simulating device $M'_{s(i)}$ which simulates M_i with polynomially bounded overhead in time and constant factor overhead in storage.

The bounds on the overheads are more formally expressed by:

If T_j' and S_j' are the running time and storage use of device M_j' in the other formalism, then one has for some suitable chosen constants c and c' that $T'_{s(i)}(n) \leqslant c \cdot (T_i(n))^c + c$ and $S'_{s(i)}(n) \leqslant c' \cdot S_i(n) + c'$ respectively. The symbol n is used in this context to denote the length of the input of the computation. The constants c and c' are independent of n.

The important consequence of this fortunate state of the world is that, although precise time and space bounds are highly dependent on the machine model used, a number of fundamental complexity classes, which are defined in such a way that they are closed under the overhead factors mentioned above, are machine independent. These classes form the well-known hierarchy:

$$\begin{aligned}\mathbf{LOGSPACE} \subseteq \mathbf{NLOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}& \\ = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}&,\end{aligned}$$

where each inclusion hides the open problem of whether or not the inclusion is proper. The case **PSPACE** = **NPSPACE**, as shown by Savitch's theorem [28] is a remarkable exception. For further references see [13], [18], [23], [30], [35].

Note that the Invariance Thesis implies that the above sequence is machine independent, without the converse implication being evident. Even if the above classes are invariant the invariance thesis may fail to be true due to the fact that the efficient simulating devices exist but cannot be constructed in an uniform way; it is also conceivable that the time efficient simulations are not the same as the space efficient ones.

The family of machine models which are used in the definition of the above classes together represent a reasonable model for the concept of sequential computation. It was to be expected that a similar family of models would be invented for parallel computation. As it turns out this class consists of an even a wider variety of device types. The parallelism can be made entirely explicit as in the SIMDAG model of Goldschlager [15]. It can also be hidden inside a sequential model either by modifying the concept of an accepting computation as is done in the ALTERNATION model [3], or by enabling the model to operate on unreasonable long objects in unit time as is done in the case of the Vector Machines [25].

Rather than by proving bounds on the relative simulation efficiencies by which these models can simulate each other, these parallel machines are grouped together by sharing a property which is known as the so-called *Parallel Computation Thesis.* This thesis expresses that Polynomial Time on a parallel device equals Polynomial Space in the sequential case.

One could start to look for a Hierarchy for parallel machines which one could denote by:

$$\text{//}\mathbf{LOGSPACE} \subseteq \text{//}\mathbf{NLOGSPACE} \subseteq \text{//}\mathbf{PTIME} \subseteq \text{//}\mathbf{NPTIME} \subseteq \text{//}\mathbf{PSPACE} \subseteq \text{//}\mathbf{NPSPACE} \subseteq \text{//}\mathbf{EXPTIME},$$

where the symbol // indicates parallelism, to be replaced by an indicant for a specific machine model if needed. Existence of such a hierarchy is not trivial, due to the fact that for a naive definition of space consumption the self evident truth from the sequential case, that space consumption never exceeds computation time, no longer is true. This truth holds for the case of the Alternating machines, due to the fact that these are sequential devices with a modified concept of acceptation. However, for alternating devices the concept of nondeterminism no longer makes sense, so for these devices the above hierarchy becomes incomplete. For a number of other devices the truth of the problematic inclusion //**NPTIME** ⊆ //**PSPACE** can be established indirectly by use of the parallel computation thesis.

Clearly the time hierarchy:

$$\text{//}\mathbf{PTIME} \subseteq \text{//}\mathbf{NPTIME} \subseteq \text{//}\mathbf{EXPTIME} \subseteq \text{//}\mathbf{NEXPTIME}$$

makes sense for any type of parallel device, and one can investigate whether the above hierarchy can be shown to be invariant for a wider class of devices. It turns out that this is indeed the case. The *Parallel Computation Thesis* expresses this by stating that:

$$\text{//}\mathbf{PTIME} = \mathbf{PSPACE}, \qquad \text{//}\mathbf{NPTIME} = \mathbf{NPSPACE},$$

and therefore, by Savitch's theorem one has //**PTIME** = //**NPTIME**.

The above equality //**NPTIME** = **PSPACE** implies the inclusion //**NPTIME** ⊆ //**PSPACE**, provided **PSPACE** ⊆ //**PSPACE**, a fact which in most cases is easy to establish.

The question of whether the parallel computation thesis is true or not as asked for example by N. Blum in [2] only makes sense if one believes that there exists an absolute notion of parallelism. It seems far more natural to use the parallel computation thesis not as dogma but rather as a tool delineating a particular, frequently observed species of parallelism. The above equalities, if shown to be valid for two specific parallel devices, indicate that these devices simulate each other with polynomial time overhead (provided the running times looked at are not to small). They represent therefore a kind of invariance similar to that expressed by the invariance thesis, and establish the existence of a *Second Machine Class* of parallel devices, together with a relation which connects this second machine class to the first machine class of sequential devices.

In this paper we present a rather global survey on a number of parallel device types which obey the parallel computation thesis as stated above, and which therefore can be called true second machine class members. We also mention some devices which are either weaker or more powerful than required by the parallel computation thesis; these later devices therefore are either intermediate between the first and second machine class, or may lead the way to a third class of even more powerful devices waiting to be investigated.

In our survey the reasons which enable these machines to obey the parallel computation thesis will be the central topic of discussion; details of the models considered will be omitted. As indicated in the next section it turns out that validity of the parallel computation thesis is closely related to a characterization of **PSPACE** in terms of transitive closure of directed graphs.

This paper is a companion paper to the survey presented in [39], where new models and insights have been included, and some earlier devices have been omitted. The topic of a more complete taxonomy of machine models remains a future target for a truly encyclopedic endeavor.

2. PSPACE and transitive closure

PSPACE is the class of those language which can be recognized in polynomial space on some sequential device, in particular on a standard single tape Turing machine. Now one can investigate for a given input x, a given Turing machine M and a given spacebound S the graph $G(x, M, S)$ of all configurations c of machine M which use space $\leqslant S$; an edge connects two configurations c_1 and c_2, if there is a one step transition from c_1 to c_2. This graph has the following properties:

1. For every input x and spacebound S there exist an unique node corresponding to the initial configuration on input x.

2. Assuming that a suitable notion of acceptation has been chosen the accepting configuration is unique.

3. The number of nodes in the graph $G(x, M, S)$ is bounded by some exponential function $2^{c \cdot S}$, where the constant c depends on M but not on x.

4. The graph $G(x, M, S)$ can be encoded in such a way that an S-space bounded Turing machine on input x and a description of M can write the encoding of $G(x, M, S)$ on some write only output tape.

5. If M is deterministic, then every node in $G(x, M, S)$ has outdegree $\leqslant 1$; if M is nondeterministic, then the outdegree of some nodes can be $\geqslant 2$, but for a suitable restriction of the Turing machine model the outdegree can be assumed to be $\leqslant 2$ as well.

6. The input x is accepted in space S by M iff there exists a path from the unique initial configuration on input x in $G(x, M, S)$ to an (or with a suitable restriction on the model, the unique) accepting configuration. This

path can be assumed to be loop-free, and therefore its length can be assumed to be $\leqslant 2^{c \cdot S}$.

The above properties suggest the following Universal algorithm for testing membership in languages in **PSPACE**: assume that L is recognized by M in space n^k, then in order to test whether $x \in L$ we first construct $G(x, M, |x|^k)$, next we compute the reflexive transitive closure of this graph, and finally investigate whether in this transitive closure there exists an edge between the unique initial configuration on x and the unique accepting configuration. Although it seems that this is a rather heavy method for simulating a single computation, it is exactly this transitive closure algorithm which underlies almost all proofs that some particular parallel machine obeys the parallel computation hypothesis.

Various descriptions of the transitive closure algorithm lead to various insights. In the first place we can represent the graph $G(x, M, S)$ by a Boolean matrix $M(x, M, S)$, where 1 denotes the presence of an edge and 0 denotes the absence of an edge. The row and column indices denote configurations. Clearly these configurations can be written down in space S; the total number (and therefore also the size N of the matrix $M(x, M, S)$) is exponential in S; $N = 2^{c \cdot S}$. If we let $M(x, M, S)[i, i] = 1$ for all $i \leqslant N$, then the transitive closure of M can be computed by $c \cdot S$ squarings of $M(x, M, S)$ in the Boolean matrix multiplication. If N^3 processors are available each squaring can be performed in time $O(S)$ (needed for adding N Boolean values), so the entire transitive closure algorithm takes time $O(S^2)$. If, moreover, a timebound T on the computation is given this time is reduced to $O(S \cdot \log(T))$, due to the fact that no paths longer than T edges have to be investigated. Finally, if the parallel model has a concurrent write feature the time needed for adding the N Boolean values can be reduced to $O(1)$, and in this case the time for the transitive closure algorithm becomes $O(\log(T))$. Note, however, that, in order to perform this algorithm, the matrix $M(x, M, S)$ must be constructed first.

We next investigate the following recursive function path (order, i, j) which evaluates to **true** in case there exists a path from node i to node j of length $\leqslant 2^{\text{order}}$:

```
proc path = (int order, node $i, j$) bool:
    if order = 0 then $i = j$ or $i$ succ $j$ # there is an edge from $i$ to $j$ #
    else
        bool found := false;
        forall node $n$ while not found do
            found := found or (path(order 1, $i$, $n$) and path(order 1, $n$, $j$))
        od;
        found
    fi;
```

Existence of an accepting computation now can be evaluated by the call found ($c \cdot S$, init, final), where init and final are the unique initial and final configurations in $G(x, M, S)$; given a timebound T the initial parameter order can also be chosen to be $\log(T)$. With recursion depth $c \cdot S$ and parameter size $O(S)$ this recursive procedure can be evaluated in space $O(S^2)$; this is the main ingredient of the proof of Savitch's theorem [28] which implies that **PSPACE** = **NPSPACE**:

If the language L is recognized by some nondeterministic Turingmachine in space $S(n) \geqslant \log(n)$, then it can be recognized by some deterministic Turingmachine in space $S(n)^2$.

A third formulation of the transitive closure algorithm yields the **PSPACE**-completeness of the problem ***Quantified Boolean Formulas*** **(QBF)** [24]:

QUANTIFIED BOOLEAN FORMULAS.

INSTANCE. A formula of the form $Q_1 x_1 \ldots Q_n x_n [P(x_1, \ldots, x_n)]$, where each Q_i equals $\forall$ or $\exists$, and where $P(x_1, \ldots, x_n)$ is a propositional formula in the boolean variables $x_1, \ldots, x_n$.

QUESTION. Is this formula true?

The idea is to encode nodes in $G(x, M, S)$ by a boolean valuation to a sequence of $k := c \cdot S$ boolean variables. From the proof of Cook's theorem which establishes the **NP**-completeness of **SATISFIABILITY** (see [5] or [13]) one obtains the existence of a propositional formula P_0 in $2 \cdot k$ variables: $P_0(x_1, \ldots, x_k, y_1, \ldots, y_k)$, where the variables $x_1, \ldots, x_k$ encode node i, the variables $y_1, \ldots, y_k$ encode node j and P_0 expresses that $i = j$ or there exists an edge from i to j. On now can define by induction a sequence of Quantified propositional formulas P_d such that $P_d(x_1, \ldots, x_k, y_1, \ldots, y_k)$ expresses the presence of a path of length $\leqslant 2^d$ between node i and node j. In a naive approach the formula P_d would include only existential quantifiers, but P_d would include two copies of P_{d-1}; by a standard trick from complexity theory we reduce the number of occurrences of P_{d-1} in P_d to one; this trick, however, introduces universal quantifiers:

$$\begin{aligned} P_d(x_1, \ldots, x_k, y_1, \ldots, y_k) = \exists z_1, \ldots, z_k \big[\forall u_1, \ldots, u_k, \forall v_1, \ldots, v_k \\ \big[((u_1, \ldots, u_k = x_1, \ldots, x_k \textbf{ and } v_1, \ldots, v_k = z_1, \ldots, z_k) \textbf{ or} \\ (u_1, \ldots, u_k = z_1, \ldots, z_k \textbf{ and } v_1, \ldots, v_k = y_1, \ldots, y_k)) \textbf{ implies} \\ P_{d-1}(u_1, \ldots, u_k, v_1, \ldots, v_k)\big]\big]. \end{aligned}$$

Substituting for $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ the codings of the initial and final node in $G(x, M, S)$ in $P_k(x_1, \ldots, x_k, y_1, \ldots, y_k)$ we obtain a closed quantified boolean formula, whose truth expresses the existence of an accepting computation. It is not difficult to see that P_k is a formula of length $O(k^2)$

in $O(k^2)$ variables (counting each variable as a single symbol; clearly a representation in a finite alphabet will introduce another factor $\log(k)$ in the length of the formula). From this one concludes that **QBF** is **PSPACE**-complete.

The **PSPACE**-completeness of **QBF** explains the *alternating* nature of a number of known **PSPACE**-complete problems. **SATISFIABILITY** is the prototype of a solitaire game, where the player has to look for some configuration with a particular property or for a sequence of simple moves leading to some particular goal state, but the alternating quantifiers turn **QBF** into a two person game. Two players Elias and Alice in turn choose the truth values to be assigned to existentially or universally quantified variables in the order of their nesting inside the formula. Elias tries to establish the truth of the formula whereas Alice tries to show that the given formula is false. The truth of the entire formula is equivalent with the existence of a winning strategy for Elias in this game.

Starting with this game-theoretic interpretation of **QBF** several authors have investigated the endgame analysis of games inspired by real life games. A useful intermediate game is **GENERALISED GEOGRAPHY** [33]; from there one can reach **HEX**, **CHECKERS**, **GO** (provided one introduces a termination rule enforcing a sufficiently fast termination of the game), and even **HEX** on the traditional hexagonal board; see [7], [11], [22], [27]. For people interested in the position of **CHESS**: this Royal game is not on the list because it turns out to be even more difficult than **PSPACE** [12].

From the above one should not conclude that in general solitaire games are at most **NP**-hard; beside the earlier **PSPACE**-completeness of the **BLACK PEBBLE GAME** [14], one has nowadays examples of group theoretical problems which have been shown to be **PSPACE**-hard [19], [20]. Together with problems which encode **PSPACE**-bounded computations in a more direct way (like Reif's **GENERALISED MOVER'S PROBLEM** [26]) this has lead to an interesting Zoo of **PSPACE**-complete problems. It should, however, be no longer a surprise that *alternation* forms a fundamental concept in one of the machine models in the second machine class.

3. A weak parallel machine

The *Parallel Turing Machine* (abbreviated PTM), introduced by Wiedermann in [41] should not be confused with the devices introduces by Savitch under the name *Recursive Turing machines* in [29]. In both cases one considers a Turing machine with a nondeterministic program where a choice of possible successor states leads to the creation of several devices, each continuing in one of the possible configurations. But in Savitch's model the entire configuration is multiplied, whereas in Wiedermann's model only the finite control and the heads are multiplied, thus leading to a proliferation of Turing automata all processing the same collection of tapes.

Wiedermann's device consists of a finite control with k d-dimensional worktapes, the first of which contains the input at the start of the computation. Each control has one head on every tape. The program of the device is a standard nondeterministic Turingprogram for a machine with k d-dimensional single-head tapes; however, in stead of chosing a next state when facing a nondeterministic move the machine creates new copies of its control and heads, which go on computing on the same tapes. There are no read conflicts; write conflicts are resolved by prohibiting them to occur; if two heads try to write different symbols at the same square the computation aborts and rejects; if two heads try to write the same symbol this symbol is written and the heads move on.

An accepting computation is a computation where every finite control which is created during the computation halts in an accepting state.

The crucial observation which makes this model weaker than the true members of the second machine class which we will meet in the sequel is related to the achievable degree of parallelism. Although the machine can activate an exponential number of copies of itself in polynomial time, these copies operate on the same tapes and therefore only a polynomial number of essentially different copies will be active at any moment in time. If two finite controls are in the same internal state and have their heads positioned on the same tape squares, their behavior will be equal from that time onwards; hence these two controls actually merge into a single copy. This leads to an upper bound of $q \cdot S^k$ on the number of different automata being active at the same time, where q denotes the number of states in the program and S denotes the space used by the device. Since space is bounded by time this leads to a polynomial bound on the number of different copies.

Based on this observation it becomes possible to simulate this parallel machine by a standard deterministic Turingmachine with Polynomial time overhead: the simulator maintains on some additional worktape a list of all active finite controls with their head positions, and by maintaining a pair of old and new worktapes the machine can process the updates of each control in sequence, taking care of the needed multiplication of controls and checking for write conflicts.

It should not be inferred from this simultation that the device is a standard first class machine, since it is not clear whether the above simulation can be modified in such a way that the space overhead becomes a constant factor. For the special case of a single tape parallel machine a constant factor space overhead is achieved by storing with each tape cell the set of states achieved by heads scanning this cell; this set (being a subset of the fixed set of states of the machine) can be written down in an amount of space which is independent of the input size. But for the case of more tapes it is not sufficient to mark the tape cells by the states in which they are scanned, since one must also know which heads belong together to a single finite control, and this requires the encoding of head positions for each device. Therefore the naive simulation as indicated above requires space $O(S^k \cdot \log(S))$.

Wiedermann observes that one can recognize the language of palindromes with a PTM with two one-dimensional tapes, where the first tape is a read-only input tape, in space $O(1)$ if the space of the input tape is not counted. Still the heads on the read-only input tape may multiply, thus storing information on this tape by their positions. It seems therefore unlikely that a constant factor space overhead simultation is possible since palindromes cannot be recognized in constant space by a standard machine. This particular example, however, seems to depend on the particular interpretation of the space on the input tape not being counted, and an example where the input head is common for all copies of the finite control seems to be required for a more convincing separation result. The above simulations show that **P** = **PTM-PTIME** and that **PSPACE** = **PTM-PSPACE**; the PTM model therefore does not obey the parallel computation thesis, unless **P** = **PSPACE**.

The PTM model has the interesting property that for several practical problems in **P** an impressive speed-up by pipelining can be achieved, even though the device is not a second machine class member. For details I refer to [41].

4. The Alternation model

The concept of *Alternation* [3] leads to machine models which obey the parallel computation thesis without providing any intrinsic parallelism at all. As a computational device an Alternating Turing machine is very similar to a standard Nondeterministic Turing machine; only its mode of acceptations has been modified.

Since the machine is nondeterministic the computation can be represented as a computation tree the branches of which represent all possible computations. The leaves, i.e., the terminal configurations (where the machine halts) are designated to accept or to reject as usual on basis of the designation of the included state as being accepting and rejecting. For the standard nondeterministic machine such a computation tree is considered to represent an accepting computation as soon as a single accepting leaf can be found. But for the alternating machine the notion of acceptation is far more complicated.

The main idea is to equip states in the Turing machine program with labels **existential** and **universal**. Configurations inherit the label of the state included in this configuration. Next one assigns a quality **accept**, **reject** or **undef** to every node in the computation tree according to the following rules:

(1) The quality of an accepting (rejecting) leaf equals **accept** (**reject**).

(2) The quality of an internal node representing an Existential configuration is **accept** if one of its successor configurations has quality **accept**.

(3) The quality of an internal node representing an Existential configuration is **reject** if all of its successor configurations have quality **reject**.

(4) The quality of an internal node representing a Universal configuration is **reject** if one of its successor configurations has quality **reject**.

(5) The quality of an internal node representing a Universal configuration is **accept** if all of its successor configurations have quality **accept**.

(6) The quality of a node with one successor equals the quality of its successor.

(7) The quality of any node whose quality is not determined by application of the above rules is **undef**.

Clearly the quality **undef** arises only if the computation tree contains infinite branches, but even nodes which have infinite offspring can obtain a definite quality since, for example, an accepting son of an existential node overrides the **undef** label of another son.

By definition an Alternating device accepts its input in case the root node of the computation tree, representing the initial configuration on that input, obtains the quality **accept**.

The above treatment is a minor simplification of the presentation in [3] in as far as that the feature of negating states is not discussed. It should be clear that the notion of an alternating mode of computations makes sense for virtually every machine model and is not restricted to Turing machines.

Time (Space) consumed by an alternating computation is measured to be the maximal Time (Space) consumption along any branch in the computation tree.

The alternating device, being an incarnation of a standard device in disguise, clearly inherits the simulation results for the first machine class devices. As a consequence there exists a device independent hierarchy for alternating classes:

$$\mathbf{ALOGSPACE} \subseteq \mathbf{APTIME} \subseteq \mathbf{APSPACE} \subseteq \mathbf{AEXPTIME}$$

which by the parallel computation thesis is connected to the standard hierarchy:

$$\mathbf{APTIME} = \mathbf{PSPACE}$$

but the other classes are shifted versions in the standard hierarchy as well:

$$\mathbf{ALOGSPACE} = \mathbf{P}, \quad \mathbf{APSPACE} = \mathbf{EXPTIME},$$
$$\mathbf{AEXPTIME} = \mathbf{EXPSPACE}.$$

Note that the alternating devices have no nondeterministic mode of computation.

In the sequel I will indicate the reasons for the above equalities to be valid.

First consider the inclusion **APTIME** $\subseteq$ **PSPACE**. It suffices to show that the quality of the initial configuration in a polynomial time bounded computation tree can be evaluated in polynomial space. Clearly this quality can be evaluated by a recursive procedure which traverses the nodes of the computation tree. This procedure has a recursion depth proportional to the running time of the alternating device, whereas each recursive call requires an amount of space proportional to the space consumed by the alternating device. Hence the space needed by the deterministic simulator is proportional to the space-time product of the alternating device, which in turn is bounded by the square of the running time.

The reverse inclusion **PSPACE** $\subseteq$ **APTIME** follows as soon as we show how to recognize the **PSPACE**-complete problem **QBF** in polynomial time on an Alternating machine. This is almost trivial: let a machine quess the valuations for the quantified variables where the universally (existentially) quantified variables are guessed in a universal (existential) state; these valuations are guessed in the order of the nesting in the formula. Next the formula is evaluated in a deterministic mode and the machine accepts (rejects) if the result becomes **true** (**false**).

The equality **AEXPTIME** $=$ **EXPSPACE** is obtained by a standard padding argument from the equality shown above.

The inclusion **ALOGSPACE** $\subseteq$ **P** is shown as follows: note that a logspacebounded alternating device for a given input has only a polynomial number of configurations. These configurations can be written on a worktape. The terminal configurations obtain a quality based on the included state. Next by repeatedly scanning the list of configurations the quality of intermediate configurations can be determined by application of the rules (2)–(6). This scanning process terminates if during a scan no new quality can be determined. Since during each sweep either at least one quality is determined or the process terminates the time needed for this procedure is bounded by the square of the size of the list of configurations, whence the running time for this process is polynomial.

The reverse implication $P \subseteq$ **ALOGSPACE** is shown as follows. Assume that the language L is recognized in time $T(n)$ by a standard single tape Turing machine M. It suffices to show how an alternating device can recognize L in space $\log(T(n))$. Consider therefore the standard computation diagram of the computation of M on input x. This diagram can be represented in the form of a table of K by K symbols, where $K = T(|x|)$. The top row of this table describes the initial configuration on input x, and the bottom row describes the final configuration which should be an accepting one. Each intermediate symbol is completely determined by the three symbols in the row directly above it since the machine M is deterministic.

The alternating device now guesses the position in the bottom row of the indicant of an accepting state, and certifies this symbol by generating in a universal state three offspring machines which guess in an existential state the symbols in the three squares above it. These guesses are certified in the same way, all the way up to the top row, where guesses are certified by comparison with the input x. The amount of space required by this procedure is proportional to the space required for writing down the position of the square considered in the diagram, which is $O(\log(T(|x|)))$. Since the machine M is deterministic, only those guesses which are correct can be certified (to be shown by induction on the row number) and therefore the guesses are globally consistent. Again by a simple padding argument the equality **APSPACE** = **EXPTIME**, is an easy consequence of the equality **ALOGSPACE** = **P**.

5. Machines operating on huge objects

One of the first machine models for which the validity of what later was to become known as the Parallel Computation Thesis, has been established is the Vector Machine model of Pratt and Stockmeyer [25], shortly later succeeded by the MRAM of Hartmanis and Simon [16], [17]. These models have in common that their power originates from the possibility to operate on objects of exponential size in unit time.

All these models are derived from the RAM model with uniform time measure by extending the arithmetic with new powerful instructions. In the Vector Machine this extension consists of the introduction of a new type of registers, called vectors, which can be shifted by amounts stored in the arithmetical registers of the RAM. The contents of the vector registers can also be subjected to parallel bitwise Boolean operations like **and**, **or**, or **xor**. This makes it possible to program the concatenation of the contents of two vectors and to perform various masking operations. The MRAM model was obtained by realizing that shifting a vector amounts to multiplication or division by a suitable power ot two. Hence the separation between vectors and arithmetic registers is an inessential feature in the model; the same power can be achieved by introducing multiplication and division in unit time, preserving the bitwise Boolean operations.

Restrictions of the model have been investigated. For example one can forsake one of the two shift directions (right shift of one register is simulated by left shifting all the others); as a consequence one can drop the division instruction, which yields the MRAM model as proposed in [16]. More recently it has been established that the combination of multiplication and division, in absence of the bitwise Boolean instructions suffices as well; see [1], [34]. This result shows the power of a purely arithmetical model.

In this survey I mention a model which has been obtained by moving in the other direction, by stressing the pure symbol manipulation instructions and dropping the powerful arithmetic. This is the EDITRAM model proposed in [37] as a model of the text editor you may have in mind while editing texts behind your terminal.

In the EDITRAM we extend the standard RAM with a fixed finite set of textfiles. Standard arithmetic registers can be used as cursors in a textfile. Beside the standard instructions on the arithmetic registers the EDITRAM has instructions for: (1) reading a symbol from a file via a cursor, (2) writing a symbol into a file via a cursor, (3) positioning a cursor at the end of a file (thus computing its length), (4) positioning a cursor into a file by loading an arithmetic value into the cursor, (5) systematic replacement of string 1 by string 2 in a textfile, (6) concatenation of textfiles, (7) copying of segments of textfiles as indicated by cursor positions, and (8) deletion of segments of textfiles as indicated by cursor positions.

In the systematic string replacement instruction (5) the arguments string 1 and string 2 are to be presented by literals in the program; substitution of the contents of an entire textfile for a single character would allow a doubly exponential growth of the size of textfiles, which is more than we are aiming for.

The time complexity of the model is defined by using the logarithmic time measure for the arithmetic registers. So an edit instruction is charged according to the logarithm of the values of the involved cursors, and its cost therefore is proportional to the logarithm of the length of the (affected portion of) the textfile.

In order to verify that the EDITRAM obeys the parallel computation thesis we must prove the two inclusions **EDITRAM – NPTIME** $\subseteq$ **PSPACE** and **PSPACE** $\subseteq$ **EDITRAM – PTIME**.

The proof of the first inclusion is typical for the proof of this inclusion for similar models. Given an input we must test in Polynomial space whether the a given EDITRAM machine will accept this input or not. But by Savitch's theorem our simulation may be nondeterministic. Therefore we first guess the trace of some accepting computation and write it down on some tape. The accepting computation being polynomially time bounded we can write down the sequence of instructions in the program of the EDITRAM which are executed. Moreover, since we use logarithmic time measure on the arithmetic registers we can also maintain a log on the register values in polynomial space. We cannot maintain a log on the contents of the textfiles, since their length may grow exponentially. Instead we introduce a recursive procedure char(time, position, textfile) which evaluates to the character located at the given position in the given textfile after performing the instruction at the given time. The arguments of this procedure can be written down in polynomial space, due to the fact that the growth of the length of a textfile is bounded by a simple exponential function in the time (both systematic string replacement and concatenation will at most multiply the length of a textfile by a constant). Given this procedure it is possible to certify that the trace written on the tape indeed represents an accepting computation.

From the meaning of the individual instructions one can write down lines of code which express the value of char(time, position, textfile) in terms of similar values after the previous instruction time-1. In the case where the present instruction is a systematic string replacement we face the problem to figure out where the character at the given position was located before the replacement, since this requires information on the number of occurrences of the replaced pattern preceding this position in the given textfile; therefore the entire textfile, up to the given position must be recomputed by recursive calls, but this is by far the most complicated case. For details I refer to [37].

The total space required by the evaluation of this procedure is bounded by the product of the size of an individual call (which we indicated to be polynomial) and the recursion depth (which is bounded by the running time of the EDITRAM computation being simulated, which was also assumed to be polynomial). This completes the proof of the first inclusion.

For the case of the Vector machine and the MRAM a similar recursive procedure can be defined which evaluates the contents of a given bit of a given vector or a given bit of an given arithmetic register at some given time. In the Vectormachine the size of a vector grows exponentially but not worse, whereas for the MRAM all arithmetic registers may grow exponentially in length. Due to the presence of carries the simulation of a multiplication becomes as complicated as the case of a string replacement in the EDITRAM. Divisions have been reduced to multiplications inside the MRAM model itself at an earlier stage of the proof. Also the length of register addresses remain bounded by the standard trick enabling the machine to use consecutive registers in its memory.

In general these simulations achieve the required spacebound at the price of a huge consumption of time; the same values are computed over and over again by the recursion.

Next we consider the inclusion **PSPACE** $\subseteq$ **EDITRAM – PTIME**. It suffices to show how to solve the **PSPACE**-complete problem **QBF** in polynomial time on a deterministic EDITRAM. Consider a given instance $Q_1 x_1 \ldots Q_n x_n [P(x_1, \ldots, x_n)]$ of **QBF**. Our algorithm is performed in three stages:

1. Remove the quantifiers in the order of their nesting from inside to outside by programming the transformations:

$$\forall x_i [F(\ldots, x_i, \ldots)] \Rightarrow (F(\ldots, 0, \ldots) \wedge F(\ldots, 1, \ldots)),$$
$$\exists x_i [F(\ldots, x_i, \ldots)] \Rightarrow (F(\ldots, 0, \ldots) \vee F(\ldots, 1, \ldots)).$$

Clearly each transformation preserves the truth of the involved formula; the involved formula F is a quantifier free formula, due to the order of the quantifier eliminations. After elimination of all quantifiers a formula of exponential size is obtained which still is equivalent to the given instance of **QBF**.

2. Evaluate the resulting formula by systematic string replacements of the type:

$$\begin{array}{llll} (0 \vee 0) \Rightarrow 0, & (0 \vee 1) \Rightarrow 1, & (1 \vee 0) \Rightarrow 1, & (1 \vee 1) \Rightarrow 1, \\ (0 \wedge 0) \Rightarrow 0, & (0 \wedge 1) \Rightarrow 0, & (1 \wedge 0) \Rightarrow 0, & (1 \wedge 1) \Rightarrow 1, \\ (\neg 0) \Rightarrow 1, & (\neg 1) \Rightarrow 0, & (0) \Rightarrow 0, & (1) \Rightarrow 1 \end{array}$$

these transformations can be produced by local systematic string replacements.

3. Check whether the resulting literal equals 0 or 1.

Note that after a single cycle through the replacements in 2 the depth of the involved propositional expression has been decreased by at least 1; If the depth of the propositional kernel of the given instance was k, then after the transformations of stage 1. the depth of the intermediate formula is $k+n$ (n being the number of quantifiers eliminated). Therefore the number of iterations in stage 2 is polynomial.

It remains to show how to perform the transformations in stage 1. Clearly it suffices to locate and read the innermost quantifier and to form the conjunction or disjunction of two copies of the propositional kernel provided the quantified variable has been replaced by 0 and 1 respectively in these copies. But since our EDITRAM program allows only literal strings as arguments in systematic replacement instructions we must program the later substitutions. We design therefore a subroutine which copies the string of characters representing variable x_i into a special purpose textfile (which encoding will include some binary representation of its index i), and next subjects all occurrences of variables in the propositional kernel F to a treatment of systematic replacements which will turn all occurrences of x_i into a special pattern, and which will leave all other variables undisturbed. Then by substituting 0 or 1 for the special pattern, the required substitutions are obtained. For details of this subroutine see [37]. This completes the proof of the second inclusion.

The proof of the corresponding inclusions in the original papers on vector machines and MRAMs involved a direct simulation of the transitive closure algorithm by subroutines which build the matrix $M(x, M, S)$ into a register and which compute its transitive closure by iterated squarings. A main ingredient is the programming of a routine which builds a bitstring consisting of the 2^K bitstrings representing the first 2^K integers, separated by markers, and of bitstrings to be used as masks for extracting a given bitposition from these integers in parallel in a single instruction. The idea to simplify these proofs by the use of **QBF** as a **PSPACE**-complete problem was also used in [1].

6. Machines with true parallelism

In this section we pay attention to the models which provide visible parallelism by having multiple processors operate on shared data and/or shared channels.

There are several methods of controlling the creation of parallel processors. Some models have an infinite collection of identical processors which operate synchronously in parallel. In other models processors by their own action can create a finite number of new processors running in parallel, and in this way an arbitrary large tree of active processors can be activated as time proceeds. In this section I will concentrate on the first type of models; the other type has been discussed in some detail in [39].

In the SIMDAG model [15] there exists a single global processor which can broadcast instructions to a potentially infinite sequence of local processors, in such a way that only a finite number of them are activated. The mechanism to keep the number of processors activated in a single step finite uses the *signature* of the local processors. Each local processor contains a read only register, called signature, containing a number which uniquely identifies this local processor. The Global processor, in broadcasting an instruction includes a threshold value, and all local processors whose signature is less than the threshold value transmitted perform the instruction, while the others remain inactive. Since the Global processor can at most double the value of its threshold during a single step it follows that the number of subprocessors activated is bounded by an exponential of the running time of the SIMGAG computation.

The local processors operate both on local memory and on the shared memory of the global processor, where write conflicts are resolved by priority; the local processor with the lower index wins in case of a write conflict.

As such the priority solution is one out of a number of possible strategies for resolving write conflicts. Other strategies which have been investigated are Exclusive Write (no two processors can write in the same global register at all), Common Write (if two processors try to write different values at the same time in the same register then the computation jams but writing the same value is permitted), and Arbitrary Write (one of the writers wins but it is nondeterministically determined which one). The computational power of the parallel RAM models based on these resolution strategies has been compared in [8], [9] under the assumption that the individual processors have arbitrary computing power. In these investigations the priority model has established itself as the most powerful one.

As an example of a machine of the other type I mention the Recursive Turing machine [29]. In this model every copy of the device can spawn off new copies which start computing in their own environment of worktapes, and which communicate with their originator by means of channels shared by two copies of the machine. Clearly models based on such local communication channels are much slower in broadcasting information to a collection of subprocessors and in gathering the answers. The validity of the parallel computation thesis is not disturbed by this delay due to the fact that the slowdown is at worst polynomial: in order to activate an exponential number of processors by spawning off subprocessors polynomial time suffices in case a complete binary tree is formed, and the time consumed for activating this tree usually is polynomial in terms of the time consumed by writing down the data to be processed by the subprocessors.

A crucial difference between the SIMDAG model and the Recursive Turing machine model is that in the SIMDAG model the local processors, if they are active at all at some time, all execute the same instruction on data which may be different. As a consequence it is possible to write down the trace of executed instructions of a SIMDAG computation in polynomial space. In the Recursive Turing machine each processor, once being activated, performs its own program except for the impact of communication with its originator or its offspring. It becomes therefore impossible to write down the complete computation trace in polynomial space if an exponential number of processors is activated.

For the above types of device it becomes relevant to restrict the power of the arithmetic instructions involved. Otherwise, as we will see in the sequel, the machine may become to powerful.

In the SIMDAG model the instruction repertoire for local and global processors involves additive arithmetic and parallel Boolean operations, combined with moderate shifting (division by 2). The writing of data in global storage by local processors is made conditional by stipulating that writing the value 0 is suppressed. In this way the **or** of a list of Boolean values computed by the local processors can be computed is a single write instruction by letting each local processor write a 1 in a fixed register in global memory if his bit equals 1 whereas the value 0 is not transmitted to the global memory.

Based on the above incomplete description one can indicate why the parallel computation thesis is true for the SIMDAG model. The inclusion

SIMDAG − NPTIME $\subseteq$ **PSPACE** is shown by an argument similar to that used for the corresponding inclusion in case of the EDITRAM.

The trace of a nondeterministic SIMDAG computation can be guessed and be written down on a worktape in polynomial space. Next one defines a pair of recursive procedures global(time, register) and local(time, register, signature) which evaluate to the value stored at the given time in the given register of the global and given local processor respectively. The arguments of these recursive procedures can be written down in polynomial space (due to the restrictions on the arithmetics of the SIMDAG) and the recursion depth is bounded by the running time of the SIMDAG computation. Using these recursive procedures the guessed trace of the SIMDAG computation can be certified to be a correct accepting computation. As before the time needed for the simulation is very large due to the recomputation of intermediate results.

The converse inclusion **PSPACE** $\subseteq$ **SIMDAG − PTIME** is shown by presenting an implementation of the transitive closure algorithm which run in polynomial time.

This algorithm first loads the K by K matrix $M(x, M, S)$ in global memory. For convenience assume that K is a power of 2. The matrix is constructed by letting processor $i+K\cdot j$ evaluate the entry $M(x, M, S)[i, j]$. By inspecting its signature the processor, (using the Boolean operations and the division by 2) can determine first the values of i and j and next unravel these values as bitpatterns in order to see whether the two encoded Turing-machine configurations are equal or are connected by a single step. By a global write the matrix is loaded into global memory.

After formation of the matrix the transitive closure is computed by iterated squaring. Each squaring is computed by letting local processor $i+K\cdot j+K\cdot K\cdot k$ read the values of $M(x, M, S)[i, k]$ and $M(x, M, S)[k, j]$, form the **and** of these two values and write the result (conditionally) in $M(x, M, S)[i, j]$. This requires therefore a constant number of steps.

After these squarings the existence of an accepting computation is determined by the global processor by inspecting the proper matrix entry.

A more refined analysis of the running time shows that the total running time consists of three contributions:

(1) $O(\log(K)) = O(S)$ for evaluation of the matrix size K,

(2) $O(S)$ for unraveling the configurations and computing $M(x, M, S)$,

(3) $O(\log(T)) = O(S)$ for computing the transitive closure of $M(x, M, S)$.

This more refined analysis shows why the power of the arithmetics in the model is a crucial factor in establishing whether the model obeys the parallel computation thesis or not. Assume that we can use multiplication in unit time the first contribution is reduced to $O(\log\log(K)) = O(\log(S))$; given parallel Boolean instructions the unraveling can be distribution over $O(S)$ processors and therefore contribution (2) becomes $O(\log(S))$ as well. As a consequence for the resulting PSIMDAG (for Powerful SIMDAG) model one obtains **NEXPTIME** $\subseteq$ **PSIMDAG – PTIME**. Hence it is unlikely that such a model obeys the parallel computation thesis, unless **PSPACE** = **NEXPTIME**. The above idea underlies the objection against the parallel computation thesis as put forward by N. Blum [2].

There exist in the literature several other models which seem to be more powerful than a second class machine. Fortune and Wyllie [10] have described a hybrid of the SIMDAG with a parallel machine based on branching, called P-RAM in [10] and called MIMD-RAM in my earlier survey [39] for which it is already established that **P-RAM – PTIME** = **PSPACE** but **P-RAM – NPTIME** = **NEXPTIME**. Savitch [31] has defined a model based on an MRAM which creates parallel copies by branching, called the LPRAM, and proves that **LPRAM – PTIME** = **PSPACE** and **LPRAM – NLOGTIME** = **NP**. These devices indicate that there may exist a third machine class, but a more precise characterization of such a class in terms of invariance properties remains open.

To conclude this section I like to mention one more model, which is inspired by the current vectorized supercomputers: the Array Processing Machine (APM) proposed by van Leeuwen and Wiedermann [40]. This machine has the storage structure of an ordinary RAM but contains besides the traditional accumulator also a *vector accumulator* which consists of a potentially unbounded linear array of standard accumulators.

The array processing machine combines the instruction set of a standard RAM with a new repertoire of vector instructions which operate on the vector accumulator. These instructions allow for reading, writing, transfer of data and arithmetic on vectors of matching size which consist of consecutive locations in storage and/or an initial segment of the vector accumulator.

Each operation on the vector accumulator destroys its previous content. Conditional control on a vector operation is possible by the use of a *mask* which consists of an array of Boolean values (0 or 1) of the same size as the vector operands; the vector instruction now is performed only at those locations corresponding to occurrences of 1 in the mask. A complete address for a vector operation therefore may consist of four integers: lower and upperbound of the vector argument and the mask respectively.

The power of parallelism is provided to the model by the time measure used: uniform time or logarithmic time, where every vector instruction is charged as much as its most expensive scalar component. So in a vector LOAD the logarithmic time complexity is proportional to the logarithm of the upperbound of the operand and/or mask plus the logarithm of the largest value loaded into the vector accumulator.

In their paper [40] the authors prove that the above array processor is a member of the second machine class by providing mutual simulations with respect to the SIMDAG, where it turns out that the simulations require polynomial (more specifically n^4) overhead. In both directions the simulations require nontrivial programming techniques, and an $O(\log^2(n))$ implementation of Batcher's sort is an essential element of the simulation.

Inspection of the model shows that a proof that the APM obeys the parallel computation thesis is easily obtained by the techniques used for other devices. To prove **PSPACE** $\subseteq$ **APM – PTIME** one can show that **QBF** can be solved in polynomial time on an APM; here the main ingredient is the construction of n vectors in storage of length 2^n, where vector j contains the value of bit j in the binary representation of the numbers $0 \ldots 2^n - 1$; Using those vectors one can evaluate a given propositional kernel in linear time using the vector instructions, and the resulting vector can be folded together according to the quantifiers in order to provide the final result. For the converse inclusion: **APM – NPTIME** $\subseteq$ **PSPACE** the usual technique of writing down a computation trace and certifying it by means of a recursive procedure should work. The detour via **PSPACE** implies the existence of mutual polynomial time overhead simulations of SIMDAG and APM but it does not provide explicit simulation overheads as indicated above.

7. Conclusion

During the two years passed since I wrote my earlier survey [39], the situation concerning the status of the parallel computation thesis has been clarified. A number of new second machine class members have been proposed, whose main characteristic is that they are closer to the conceptual models programmers have in mind while thinking about the machines they are using. Both the EDITRAM and the APM are derived from real life devices (Text editors and vectorprocessors).

At the same time the awareness has grown that the physical limits of parallelism turn the parallel computation thesis into an unrealistic dream; see for example [4], [36]. The subject of the power of parallelism with polynomial bounds on the number of processors has become a research topic in itself, and the classes **SC** and **NC** which are subclasses of **P** have become widespread in algorithm design [6].

I have abstained from discussing these realistic down-scaled version of parallelism in the present survey. Also untreated is the relation between parallel machine models and complexity in terms of circuits etc. Clearly a truly complete taxonomy of machine models should include such models as well.

Acknowledgements. This paper has been influenced by the discussions I had about my earlier survey with a lot of colleagues; amongh these I like to mention N. Blum, E. Börger, R. Freivalds, J. Gruska, K. Mehlhorn, L.

Stockmeyer, L. Torenvliet, J. van Leeuwen and J. Wiedermann. The assistance of L. Torenvliet during my struggle for preparing a first paper on a Macintosh was essential for finishing the manuscript before the deadline.

References

[1] A. Bertoni, G. Mauri and N. Sabadini, *A Characterization of the class of Functions computable in Polynomial time on Random Access Machines*, Proc. ACM STOC 13 (May 1981), 168–176.

[2] N. Blum, *A note on the "Parallel Computation Thesis"*, Inf. Proc. Letters 17 (1983), 203–205.

[3] A. K. Chandra, D. C. Kozen and L. J. Stockmeyer, *Alternation*, JACM 28 (1981), 114–133.

[4] B. Chazelle and L. Monier, *Unbounded Hardware is equivalent to Deterministic Turing-machines*, TCS 24 (1983), 120–123.

[5] S. A. Cook, *The Complexity of Theorem Proving Procedures*, Proc. ACM STOC 3 (May 1971), 151–158.

[6] –, *The Classification of Problems which have fast Parallel Algorithms*, in: M. Karpinski ed., Proc. Fundamentals of Computation Theory 1983, Borgholm Sweden (August 1983), Springer LCS 158 (1983), 78–93.

[7] S. Even and R. E. Tarjan, *A Combinatorial Problem which is complete in Polynomial Space*, JACM 23 (1976), 710–719.

[8] F. E. Fich, P. L. Radge and A. Widgerson, *Relations between Concurrent-Write Models of Parallel Computation*, Proc. PODC 3 (August 1984), 179–189.

[9] F. E. Fich, F. Meyer auf der Heide, P. L. Radge and A. Widgerson, *One, Two, Three ...Infinity: Lower Bounds for Parallel Computation*, Proc. STOC 17 (May 1985), 48–58.

[10] S. Fortune and J. Wyllie, *Parallelism in Random Access Machines*, Proc. STOC 10 (May 1978), 114–118.

[11] A. S. Fraenkel, M. R. Garey, D. S. Johnson, T. Schaefer and Y. Yesha, *The Complexity of Checkers on an $N \times N$ Board, preliminary report*, Proc. FOCS 19 (October 1978), 55–64.

[12] A. S. Fraenkel and D. Lichtenstein, *Computing a Perfect Strategy for $n \times n$ Chess requires time Exponential in n*, Proc. ICALP 8 (July 1981), Springer LCS 115 (1981), 278–293.

[13] M. S. Garey and D. S. Johnson, *Computers and Intractability, a guide to the Theory of NP-completeness*, Freeman, San Francisco 1979.

[14] J. R. Gilbert, T. Lengauer and R. E. Tarjan, *The Pebbling Problem is Complete in Polynomial Space*, SICOMP 9 (1980), 513–524.

[15] L. M. Goldschlager, *A Universal Interconnection Pattern for Parallel Computers*, JACM 29 (1982), 1073–1086.

[16] J. Hartmanis and J. Simon, *On the Power of Multiplication in Random Access Machines*, Proc. SWAT 15 (October 1974), 13–23.

[17] –, –, *On the Structure of Feasible Computations*, in: M. Rubinoff and M. C. Yovits eds., Advances in Computers 14, Acad. Press, 1976, 1–43.

[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

[19] M. Jerrum, *The Complexity of finding Minimum-length Generator Sequences (extended abstract)*, Proc. ICALP 11 (July 1984), Springer LCS 172, 270–280.

[20] –, *The Complexity of finding Minimum-length Generator Sequences*, report CRS-198-83. DCS, Univ. Edinburgh, submitted to TCS.

[21] D. S. Johnson, *The NP-Completeness column: an ongoing guide*, J. of Algorithms, Quarterly since December 1981.

[22] D. Lichtenstein and M. Sipser, *Go is **PSPACE-hard***, Proc. FOCS 19 (October 1978), 48–54.

[23] M. Machtey and P. Young, *An Introduction to the General Theory of Algorithms*, Theory of Computation Series, North Holland, New York 1978.

[24] A. R. Meyer and L. J. Stockmeyer, *The Equivalence Problem for Regular Expressions with Squaring requires Exponential Time*, Proc SWAT 13 (October 1972), 125–129.

[25] V. R. Pratt, and L. J. Stockmeyer, *A characterization of the power of vectormachines*, JCSS 12 (1976), 198–221.

[26] J. H. Reif, *Complexity of the mover's problem and generalizations, extended abstract*, Proc. FOCS 20 (November 1979), 421–427.

[27] S. Reisch, *HEX ist **PSPACE**-vollständig*, Acta Informatica 15 (1981), 167–191.

[28] W. J. Savitch, *Relations between Deterministic and Nondeterministic tape Complexities*, JCSS 4 (1970), 177–192.

[29] –, *Recursive Turingmachines*, Inter. J. Comput. Math. 6 (1977), 3–31.

[30] –, *The Influence of the Machine Model on Computational Complexity*, in: J. K. Lenstra, A.H.G. Rinnooy Kan and P. van Emde Boas eds., Interfaces between Computer Science and Operations Research, Math. Center Tracts 99 (1978), 3–32.

[31] –, *Parallel Random Access Machines with Powerful Instruction sets*, Math. Systems Theory 15 (1982), 191–210.

[32] – and M. J. Stimson, *Time bounded Random Access Machines with Parallel Processing*, JACM 26 (1979), 103–118.

[33] T. J. Schaefer, *Complexity of some Two-person Perfect-information Games*, JCSS 16 (1978), 185–225.

[34] A. Schönhage, *On the Power of Random Access Machines*, Proc. ICALP 6 (July 1979), Springer LCS 71 (1979), 520–529.

[35] –, *Storage Modification Machines*, SICOMP 9 (1980), 490–508.

[36] A. Schorr, *Physical Parallel Devices are not much faster than Sequential ones*, IPL 17 (1983), 103–106.

[37] R. A. Stegwee, L. Torenvliet and P. van Emde Boas, *The Power of your Editor*, Report RJ 4711 (50179) 5/21/85, IBM Research Division, San Jose CA, or Report FVI-UVA-85-03.

[38] C. F. Slot and P. van Emde Boas, *On Tape versus Core; an application of Space Efficient perfect Hash functions to the invariance of Space*, Proc. STOC 17 (May 1984), 391–400.

[39] P. van Emde Boas, *The Second Machine Class: Models of Parallelism*, in: J. van Leeuwen and J. K. Lenstra eds., Parallel computers and computations, CWI Syll. 9, Center form Mathematics and Computer Science, Amsterdam 1985, 133–161.

[40] J. van Leeuwen and J. Wiedermann, *Array Processing Machines*, in: L. Budach ed., Fundamentals of Computation Theory 1985, Cottbus GDR, September 1985, Springer LCS 199, 257–268. (A more extended version is available as report RUU-CS-84-13, Rijksuniversiteit Utrecht, December 1984.)

[41] J. Wiedermann, *Parallel Turing Machines*, report RUU-CS-84-11, Rijksuniversiteit Utrecht, November 1984.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985