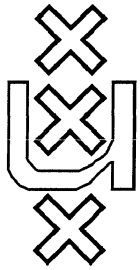
Institute for Language, Logic and Information

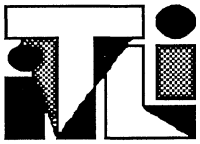
# MACHINE MODELS AND SIMULATIONS (REVISED VERSION)

Peter van Emde Boas

ITLI Prepublication Series
for Computation and Complexity Theory CT-88-05



University of Amsterdam



Institute for Language, Logic and Information
Instituut voor Taal, Logica en Informatie

# MACHINE MODELS AND SIMULATIONS (REVISED VERSION)

Peter van Emde Boas
Department of Mathematics and Computer Science
University of Amsterdam

Received August 1988

Correspondence to:

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Roetersstraat 15
1018WB Amsterdam

or

Faculteit der Wijsbegeerte
(Department of Philosophy)
Grimburgwal 10
1012GA Amsterdam

# Machine Models and Simulations (revised version)

Peter van Emde Boas
Depts. of Mathematics and Computer Science
Logic and Computation Theory group
University of Amsterdam
Nieuwe Achtergracht 166
1018 WV Amsterdam

&

CWI-AP6
Kruislaan 413
1098 SJ Amsterdam

**Abstract :** In Complexity Theory the use of informal estimates can be justified by appealing to the Invariance Thesis which states that all standard models of sequential computing devices are equivalent in the sense that the fundamental complexity classes don't depend on the precise model chosen for their definition. Similarly, the Parallel Computation Thesis states that parallel models have polynomially related time measures and that moreover parallel time is polynomially equivalent to sequential space.

In this chapter we give a survey on the world of machine models and the simulations between them. The sequential models satisfying the invariance thesis are gathered into a First Machine Class and the parallel models which satisfy the Parallel Computation thesis into a Second Machine Class. This seems not to exhaust all machine models proposed in the literature, not even all models which are introduced as being "reasonable".

Our survey takes us from the weakest universal models considered by mathematicians (like Minsky's two-counter machine) to the most powerful parallel models considered today where time stops to be a complexity measure.

July 1988

**Keywords :** Turing Machines, Random Access Machines, Time Complexity, Space Complexity, Invariance Thesis, Storage Modification Machines, Network Complexity, Alternating Machines, Parallel Machines, Parallel Computation Thesis, Simulations.

**Math. Subject Classes :** 68C25.
**Comp. Reviews Class :** F.1.2, F.2.

This paper is the revised draft of a chapter in a forthcoming Handbook of Theoretical Computer Science which will be published by North Holland Publ. Cie. in the fall of 1988.

# MACHINE MODELS AND SIMULATIONS - CONTENTS

# 1. INTRODUCTION.

## 1.1. The role of Invariance in Complexity Theory.

In this chapter I want to present a perspective on the role of machine models within computation theory in general and on the compatibility between a machine independent complexity theory and a machine based formalization for this theory in particular. The purpose is primarily to explain and illustrate the general theory on machine models and their mutual simulations rather than the rigid formal description of any specific class of models. However, with the generic description of a formalization as presented in section 1.2 in mind the reader should have no difficulty in understanding the various formal definitions of machine models as they occur in the literature.

In order to appreciate the impact of the existence of many machine models consider the situation in general computation theory. This theory knows a large variety 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 are equivalent in the following sense: each computation in formalism-1 can be simulated some way or another in formalism-2. Since the need for a computation theory arose out of the requirement to show that some problems were unsolvable by effective means this equivalence suffices for providing the researchers complete freedom of his choice of model. If one can prove that a problem is unsolvable in one model it is also unsolvable for all other formalized computing devices.

Traditionally, mathematics used to be far more constructive without having to bother about the precise notion of effective computability. The existence of a single formal concept in many disguises has allowed us to return to our informal, intuitive way of working, relying on what has become known as the *Inessential use of Church's Thesis* : ***Whatever is felt to be effective can be brought within the scope of our formal models*** . The basic models themselves remain on the shelves to be used at leasure and waiting to be trained to our students; their discovery has become history [16,51].

Church's Thesis is the kind of assertion the validity of which seems to be based on the lack of counterexamples. History has shown that all proposed models indeed are equivalent in the manner as explained above. Alternative models of computation are dreamt about in Science Fiction. I leave the question on whether the human brain can be modelled by one of our standard models or not to the philosophers and the neurophysiologists. In this chapter I restrict myself to models for which the Church's thesis indeed is valid.

When in the mid sixties the foundations were laid for a Theory of Complexity of Computation (see e.g. [35]), the family of machine models was only enlarged. The classical models from Recursion Theory, like the Unary Turing Machine [15] and Minsky's Multi-counter machine [71] turned out to be too unwieldy for modelling real life computations. Depending on the nature of the objects one likes to deal with during the computations (numbers (non-negative integers) or alphanumeric strings), two models have obtained a dominant position in machine-based Complexity Theory. The off-line multi-tape Turing machine [1] represents the standard model for string oriented computation and the Random Access Machine (RAM) as introduced by Cook & Reckhow [14] has become the idealized Von-Neumann number cruncher.

Other models were incorporated in this theory without much difficulty. One had to investigate the computational overheads in time and storage for performing the simulations of one model on another. Such simulations have become standard material in the introductory textbooks in Complexity Theory [1,65,123]. Our chapter will be dedicated to the quantitative aspects of these simulations.

At first sight it seems that all assertions on time and space consumed by machine computations are model dependent to the extent that absolute statements on complexity issues become meaningless. On the other hand complexity theory knows many notions like **NP**-completeness, which are treated with little or no considerations on the machine models involved. How can this contradictory state of affairs be explained? I firmly believe that complexity theory, as presently practiced, is based on the following assumption, held to be self evident:

**INVARIANCE THESIS:** ***There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's and RASP's with logarithmic time measures, and also the RAM's and RASP's in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with Polynomially bounded overhead in time, and constant factor overhead in space .***

It is clear that the way we operate with the hierarchy of fundamental complexity classes as described by D.S. Johnson in this handbook [48]: $\mathbf{LOGSPACE} \subseteq \mathbf{NLOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$ etc... is based on the Invariance Thesis. Without the information that machines simulate each other with polynomial overhead in time the classes **P** and **NP** become machine dependent. Invariance of the class **P** is claimed as a justification for basing complexity theory on such an inconvenient model as the Turing Machine [113]. Similarly the class **LOGSPACE** becomes machine dependent unless machines simulate each other with constant factor overhead in space.

For the interpretation of the Invariance Thesis it makes a difference whether one requires that a single simulation achieves both bounds on the overheads involved (strict interpretation) or whether one allows for a time-efficient simulation and an entirely different space-efficient simulation (which then may turn out to require an exponential overhead in time). As long as one investigates space or time bounded complexity classes independently the liberal interpretation suffices. On the other hand nearly all efficient simulations known in the literature achieve both bounds at the same time.

How does the theory community react if it is faced with evidence which seems to contradict the above Invariance Thesis? It seems that the standard strategy is to adjust the definitions when needed. The thesis becomes a guiding rule for specifying the right class of models rather than an absolute truth, and therefore this thesis, once being accepted, will never be invalidated. For example, in the mid seventies we became aware of the power of a RAM model with built-in multiplication and division in the uniform time measure, in combination with parallel bitwise logical operations. As a consequence the resulting MRAM model [37] was thrown out of the realm of reasonable machine models. Instead the MRAM satisfies the so-called *Parallel Computation Thesis*

[31] which claims that for parallel models one has the equality **//-PTIME = PSPACE**. Therefore the MRAM has established itself as one of the prominent members of a *Second Machine Class* [117,118]. Again the Parallel Computation Thesis has become a heuristic tool in providing the right definitions rather than an absolute dogmatic definition of the true parallel model.

The two families of machine models mentioned above, those sequential models which obey the Invariance Thesis and those parallel models which satisfy the Parallel Computation thesis, called the First and Second Machine Class hereafter, include a large number of known models but fail to incorporate all models proposed in the literature. There exist both intermediate models - sequential models which seem to consume less space than the first class machines - and super powerful parallel devices, achieving exponential speed-ups in time or worse.

This chapter of the handbook is further structured as follows: In section 1.2 we present a sketch of a formalization of the machine model concepts we need. Section 2 deals with sequential models, in particular the Turing Machines and the RAM's . Another sequential model is the Storage Modification Machine as described by Schönhage [104], but it turns out that it does not belong to the first machine class unless its space measure is defined in an unnatural way. We conclude this section with some remarks on non-uniform models like networks and their position with respect to the theory of machine models.

Section 3 gives a survey on the standard parallel machine models. The various ways the power of parallellism can be provided in a model will be presented in various sub-sections. This chapter is then concluded in section 4 where some models are illustrated which seem to fall outside our two machine classes.

## 1.2. Formalization.

In the preceding section I have used a large number of concepts which have so far not been defined formally. It is not the intent to provide such formal definitions in this chapter. I rather refer to the standard textbooks. In particular the recent encyclopedic text composed by Wagner and Wechsung [123] provides the reader with such a large number of definitions, references and simulation results that any attempt to compress this information in a chapter of 60 pages seems to be doomed to failure. Instead I rather indicate my perspective on the formalizations required for setting up a theory on machine models and simulations. Some familiarity with both Turing machines and the RAM model is supposed.

### 1.2.1 Machines and computations.

A *Sequential Machine model* **M** is a class of related devices $M_i$, called *Machines*, which in principle can be described as mathematical objects using the language of ordinary set theory. These devices consist of a finite part, called *Program* or *Finite control*, in combination with a potential infinite part called *Memory*. The memory is a depository for finite chunks of information either consisting of symbols from a finite set called *Alphabet* or consisting of *Numbers*. The memory has a regular structure of cells where this information is stored. This structure is mostly a linear order but alternative regular structures like higher dimensional grids or trees are also allowed.

Although the memory is in principle infinite it will always be the case that only a finite part of the memory is *used*. This is achieved by selecting a special content (*blank symbol* or simply the number zero ) for those locations where nothing so far has happened.

The finite control can be represented by a Program in a suitable type of assembly code. Typically there will be instructions for *fetching or storing information from memory*, for *modifying the visible cells in memory*, for testing of *conditions* and *performing conditional or unconditional jumps* in the program, and for performing *in- and output.*

For the purpose of in- and output two special sections of memory are dedicated; one from which information is read one symbol at a time and one by which information is communicated to the outside world, again one symbol at a time. With regard to the input section of memory two interpretations offer themselves. One can regard the input characters as entities which offer themselves for being processed only once like signals recieved from another galaxy; if one wants to read the input symbol for a second time one must first store it in some other part of the memory. This leads to the mode of *on-line* computations. If on the other hand the input symbols are permanently available for inspection and can be read as often as one likes (presumably in some other order than the sequential order of the input string itself) one uses the mode of *off-line* computations.

A *configuration* is a full description of the state of a machine and its memory. Typically a configuration will consist of a location in the program in the finite control, the structure and contents of the finite part of the memory which has participated in the computation so far, and the in- and output sections of memory. For all components of memory the configuration must moreover indicate on which cells the interaction between finite control and memory is currently taking place.

Next one introduces the so-called *transition relation* between two configurations. Configuration C2 is obtained by a transition from configuration C1 , notation C1 |---- C2 , if performing a single instruction of the program in the finite control of the machine takes us from C1 to C2 . *Computations* now can be described by the reflexive and transitive closure |-----* of this relation |---- . The word Computation refers both to the pairs in |----* and at the sequences of configurations, connected by |---- which establish the presence of a pair in |----* .

At this point the distinction between *Deterministic* and *Nondeterministic* machines can be made. For a deterministic machine there exists for every configuration C1 at most a single configuration C2 such that C1 |---- C2 , whereas for nondeterministic devices there may exist several such configurations C2 . For virtually all devices in the literature the source of nondeterminism is the fact that the program itself indicates that more than one instruction can be performed in the given configuration, and this leads to *bounded nondeterminism* : the number of possible configurations C2 is finite and bounded by a number dependent on the machine program only. *Unbounded nondeterminism* is obtained if also the interaction with the memory can be nondeterministic. For example on a RAM one may consider an instruction which loads a random integer in a memory location. In this chapter we will restrict ourselves to bounded nondeterminism.

Given the transition relation one next introduces *initial, final, accepting* and *rejecting* configurations. Initial configurations are characterized by an initial state in the finite control, a memory which is blank except for the input, an empty output memory, and all communication devices between program and memory located at some standard position. The input x completely determines an initial configuration which we denote by $C_i(x)$ . A configuration C is called final if there exist no configuration C' such that C |---- C' . Depending on the state in the finite control in a final configuration but independent on the contents of the memory a final configuration may be called accepting or rejecting.

A *full computation* is a computation which starts in an initial configuration and which does not terminate in a non-final one; either the computation is infinite in which case it is called a *divergent computation* or it terminates in a final configuration. The computation then is called accepting or rejecting in case its final configuration has this attribute.

Machines can be used for several purposes. Machine M *recognizes* the language L(M) consisting of those inputs x for which an accepting computation starting with $C_i(x)$ exists. Machine M *accepts* the language D(M) consisting of those inputs x for which some terminating computation starting with $C_i(x)$ can be constructed. The machine M finally *computes* a relation F(M) consisting of those pairs <x,y> which consist of an input x and an output y such that there exists an accepting computation which starts in $C_i(x)$ and which terminates in a configuration where y denotes the contents of the output memory.

A *Parallel Machine Model* is obtained if we omit from the above definition the condition that there exists just one finite control. In a parallel machine finite control is replaced by a *set of processors* the size of which becomes infinite in the same way as memory is infinite: the number of processors has no fixed bound, but in every configuration only a finite number of them have participated in the computation so far. Each processor has its own way of interacting with memory. It can be the case that large amounts of memory are accessible to all processors (*shared memory* ) or that processors have their own *local memory*. The condition that only one symbol at a time gets

communicated between the input or output section of memory and the finite control is relaxed for the case of parallel machine models. Processors can also have the possibility to exchange information directly by communication channels without the use of passive memory. In an extreme case like the cellular automata model [115] there only exist processors and there is no memory.

Transitions for a parallel processor are the combined result of transitions within each of the active processors. It can be the case that a global transition is obtained as the effect of a local transition in each of the active processors at the same time (*Synchronous computation* ). The alternative is that processors proceed at their own speed in a obscure way (*Asynchronous computation* ). In both cases two or more processors can interfere with each other while communicating with shared memory; this holds in particular when one processor attempts to write at a location where another processor is reading or writing at the same time. It belongs to the precise definition of the parallel model to stipulate what will happen if those *read/write conflicts* arise, and whether these conflicts may arise at all.

### 1.2.2 Simulations.

Given the above global description of machines and computations we next have to explain what we mean by a simulation of machine M1 by a machine M2 . Intuitively a simulation of M1 by M2 is some construction which shows that everything M1 can do on some input x can be performed by M2 on this input as well. The evidence that the behavior on input x is preserved should be retrievable from the computations themselves rather than just from the input/output relations established by the devices M1 and M2 . Let us see whether we can turn this intuition in a more formal definition. As it turns out it is difficult to provide for a general formal definition; as soon as one proposes a definition one finds examples of simulations which are not covered by this formal definition.

A first problem is that it is quite possible that M2 cannot process the input x at all, since the structure of the objects which can be stored in the input part of memories of the two machines may be quite different. If M1 is a Turing Machine which operates on strings and M2 is a RAM operating on numbers the sets of possible inputs for the two devices are disjoint. In order to overcome this hurdle one therefore allows that M2 operates on some suitable encoding of the input x rather than on x itself. The encoding has to be of a rather simple nature, in order to prevent various fraudulent interpretations which would allow the recognition of non-recursive sets using simulations.

Next one could introduce a relation between configurations C1 of M1 and C2 of M2 which expresses the fact that C2 represents the configuration C1 . Let us denote such a relation by $C1 \approx C2$ . Ideally one would have for a simulation: if $C1 \vdash C3$ and $C1 \approx C2$ then there exists a configuration C4 of M2 such that $C3 \approx C4$ and $C2 \vdash C4$ . This kind of lock-step simulation turns out to be far too restrictive for incorporating existing simulations. For example, such simulations are doomed to preserve the number of steps in a computation.

A next try is to require: if $C1 \vdash^* C3$ and $C1 \approx C2$ then there exists a configuration C4 of M2 such that $C3 \approx C4$ and $C2 \vdash^* C4$ . Still this is far too restrictive. For example it requires that each configuration of M1 has its analogue in M2 and that the general order of the

computation by M1 is preserved by the simulation. The first requirement will turn simulations into an asymmetrical concept which is not unreasonable. The second condition will however exclude such constructions as backward simulations, or recursive simulations where earlier configurations are re-computed, like in the simulation which proves Savitch's theorem on the relation between deterministic and nondeterministic space bounded complexity classes.

A next attempt will consider an entire computation of M1 and require that for each configuration C1 occurring in it there exists a corresponding configuration C2 in the simulating computation of M2 . This would allow both backward and recursive simulations but still it would exclude simulations where the entire transition relation of M1 becomes an object of a computation of M2 and where the effect of M1 on some input is simulated by letting M2 evaluate the reflexive transitive closure of this transition relation. This latter type of simulation will turn out to be quite common in the theory of parallel machine models.

The conclusion of the above attempts show how hard it is to define a simulation as a mathematical object. The initial definition seems to be the best one can provide, and I leave it to the reader to develop a feeling for what simulations can do by looking at the examples given in the sequel and elsewhere in the literature.

### 1.2.3 Time and Space complexity.

Given the above intuitive description of a computation it is hardly a problem to define the time taken by a terminating computation: take the number of configurations in the sequence which describes this computation. What we have done here is to assign one time unit to every transition.

In the case where a single transition may involve manipulation of objects from an infinite set (like in the case of a RAM) this approach (*uniform time measure* ) may turn out to be unrealistic, and one uses a measure where each transition is weighted according to the amount of information manipulated in the transition; the *logarithmic time measure* for the RAM represents an example.

A crude measure for the space consumed by a computation is the number of cells in the memory which have been affected by the computation. A more refined measure takes also in account how much information is stored in such a cell. It is moreover tradition not to charge for the memory used by the input and the output, in case the corresponding sections of the memory are clearly separated from the rest of the memory.

Having assigned a time and space measure to an individual computation, the next subject is to assign such measures to a machine. The way these measures are extended depends on what the machine is intended to do with the input: recognize, accept or compute. Moreover, one is interested not so much in the dependence of time and space on the precise input but rather in a more global relation between time and space of computation and the *length of the input.*

The length of input x depends on the encoding (in the standard case where x denotes some mathematical object), but traditionally one looks at codes where strings over a finite alphabet denote themselves, and where numbers are denoted using a binary or decimal notation, unless it is explicitly stated to be a *unary* notation. Assuming that inputs are represented using an alphabet of at least two symbols, it holds that the number of inputs x of length n increases as an exponential function in n .

If f(n) is a function from the set of non-negative integers **N** to itself, machine M is said to *terminate in time* f(n) ( *space* f(n) ) in case every computation on an input x of length n consumes time (space) $\leq$ f(n) . The deterministic machine M *accepts in time* f(n) (*space* f(n)) if every accepting computation on an input of length n consumes time (space) $\leq$ f(n) . The nondeterministic machine M *accepts in time* f(n) (*space* f(n)) if for every accepted input of length n an accepting computation can be found which consumes time (space) $\leq$ f(n) .

A set of inputs L is *recognized in time* f(n) (*space* f(n)) *by machine* M , if L = L(M) and M terminates in time f(n) (space f(n)) . The set L is *accepted in time* f(n) (*space* f(n)) *by machine* M if L = D(M) and M accepts in time f(n) (space f(n)).

An alternative definition is to let L be recognized by M in time f(n) (space f(n)) in case L = L(M) for a machine which accepts in time f(n) (space f(n)) . This definition turns out to be equivalent to the one given for nice models and nice functions f(n) where it is possible to *shut-off* those computations which exceed the amount of resources allowed. In the general case the definitions become different.

The set of languages *recognized by some machine model* **M** in time f(n) (space f(n)) consists of all languages L which are recognized in time f(n) (space f(n)) by some member M of **M** . We denote this class by **M-TIME**(f(n)) (**M-SPACE**(f(n)) . The class of languages recognized simultaneously in time f(n) and space g(n) will be denoted **M-TIME&SPACE**(f(n),g(n)) . The intersection of the classes **M-TIME**(f(n)) and **M-SPACE**(g(n)) consisting of the languages recognized in both time f(n) and space g(n) will be denoted **M-TIME,SPACE**(f(n),g(n)) .

Standard functions for resource bounds in this theory are logarithms, polynomials, exponential functions and their combinations. These functions involve machine or simulation dependent constants which are denoted by k in the sequel. In general one considers a sequence of such functions, describing the generic behavior of a much larger class as far as orders of magnitude are concerned. We will look in particular to the following sequences*:

**Log** = { k.log(n) | k $\in$ **N** }

**Lin** = { k.n | k $\in$ **N** }

**Pol** = { $k.n^k + k$ | k $\in$ **N** }

**Expl** = { k.exp(k.n) | k $\in$ **N** }

**Exp** = { $k.\exp(n^k)$ | k $\in$ **N** }

Next one can define that a set L is recognized (accepted) in time (space) **F** if it is recognized (accepted) in time (space) f for some f $\in$ **F** . For a machine model **M** this leads then to the fundamental complexity classes:

* All logarithms in this paper, also if not stated explicitly, are to the base 2 .

| | |
|---|---|
| **M-LOGSPACE** | = {L \| L is recognized by some M ∈ **M** in space **Log** } |
| **M-PTIME** | = {L \| L is recognized by some M ∈ **M** in time **Pol** } |
| **M-PSPACE** | = {L \| L is recognized by some M ∈ **M** in space **Pol** } |
| **M-EXPLTIME** | = {L \| L is recognized by some M ∈ **M** in time **Expl** } |
| **M-EXPTIME** | = {L \| L is recognized by some M ∈ **M** in time **Exp** } |
| **M-EXPLSPACE** | = {L \| L is recognized by some M ∈ **M** in space **Expl** } |
| **M-EXPSPACE** | = {L \| L is recognized by some M ∈ **M** in space **Exp** } |

The above classes clearly are machine dependent. It is possible, however, to obtain the hierarchy of fundamental complexity classes as indicated in the introduction by selecting for **M** the model of standard off-line multi-tape Turing Machines [1]. Denoting the Deterministic Turing machines by **T** and their non-deterministic counterpart by **NT** we obtain:

| | |
|---|---|
| **LOGSPACE = T-LOGSPACE** | **NLOGSPACE = NT-LOGSPACE** |
| **P = T-PTIME** | **NP = NT-PTIME** |
| **PSPACE = T-PSPACE** | **NPSPACE = NT-PSPACE** |
| **EXPTIME = T-EXPTIME** | **NEXPTIME = NT-EXPTIME** |
| **EXPSPACE = T-EXPSPACE** | **NEXPSPACE = NT-EXPSPACE** |

It now follows from the elementary properties of the Turing machine model that the hierarchy mentioned in section 1.1 exists. Moreover, all the fundamental complexity classes mentioned above have obtained their standard meaning (note however the notational distinction between the classes described by exponential bounds with linear and polynomials in the exponent , and compare with the definitions used in the chapter by D.S. Johnson [48]). All inclusions represent notorious open problems in complexity theory, except for the equality **PSPACE = NPSPACE** which follows from Savitch's theorem [94].

### 1.2.4 Simulation overheads and the machine classes.

Having defined the standard notions of complexity we return to the quantitative aspects of simulations. The problem we had in section 1.2.2 with a proper definition of a simulation now returns. In that section we looked at the simulation of one device M1 by another device M2.

We say that *model* **M2** *simulates* **M1** *with time (space) overhead* f(n) ,

Notation: **M1 ≤ M2 (time** f(n) ) or **M1 ≤ M2 (space** f(n) ) ,

if the following holds:

for every machine $M1_i$ of **M1** there exists a corresponding machine $M2_{s(i)}$ of **M2** such that $M2_{s(i)}$ simulates $M1_i$ , and such that moreover for every input x of $M1_i$ , if c(x) is the encoded input of $M2_{s(i)}$ which represents x, and if t(x) is the time (space) needed by $M1_i$ for processing x , then the time (space) required by $M2_{s(i)}$ for processing c(x) is bounded by f(t(x)) .

In this definition *processing* stands for either recognizing, accepting, rejecting the input or evaluating some ( possibly partial and/or multi-valued) function on this input. As explained before the encoding c(x) must be recursive (and even of a reasonably low complexity). If s(i) is recursive as well the simulation is said to be effective. Clearly, in abstract complexity theory, it can be shown that simulations can be made effective, and that recursive overheads always exist.

If **F** denotes a class of functions the above definition can be extended to *model* **M2** *simulates* **M1** *with time (space) overhead* **F** in case the simulation holds for some overhead f in **F** .

Notation: **M1 ≤ M2 (time F )** or **M1 ≤ M2 (space F )**

This enables us to identify some important classes of simulations:

| | |
|---|---|
| **M1 ≤ M2 (time Pol )** | *Polynomial time simulation* |
| **M1 ≤ M2 (time Lin)** | *Linear time simulation* |
| **M1 ≤ M2 (space Lin)** | *Constant factor space overhead simulation* |

A special case of a linear time simulation is the so-called *real-time* or *constant-delay* simulation. This is a simulation according to one of the attempted definitions which we presented in section 1.2.2 , where the configurations of M1 are represented by corresponding configurations of M2 preserving the order; moreover the number of configurations of M2 between the representatives of two successive representations of M1 configurations is bounded by a constant. We denote the existence of such a real-time simulation by **M1 ≤ M2 ( real-time )** .

If a single simulation achieves both a time overhead f(n) and a space overhead g(n) this can be expressed by the notation:

**M1 ≤ M2 (time** f(n) **&** **space** g(n) **)**

If both overheads can be achieved but not necessarily for the same simulation we express this fact by the notation:

**M1 ≤ M2 (time** f(n) **,** **space** g(n) **)**

Again these notations are extended to classes of functions where needed.

If simulations exist in both directions, achieving the same overheads we replace the symbol ≤ by the equivalence symbol ≈ . For example **M1 ≈ M2 (time Pol )** , expresses the fact that the models M1 and M2 simulate each other with polynomial time overhead.

The above machinery enables us to define what I mean by the *First Machine Class* and the *Second Machine Class* : Let **T** denote the model of Turing Machines and let **M** be another model. Then **M** is a first class machine model if **T ≈ M ( time Pol & space Lin )** . We say

that **M** is a second class machine model if **M-PTIME = PSPACE** . So first class machine models are exactly those models which satisfy the Invariance Thesis under the strict interpretation, whereas second class machines are those devices which make the Parallel Computation Thesis true.

This section illustrates how we have turned these two Theses from dogmatic specifications of the truth about machine models into a heuristic tool for classifying machines. It would be nice if all models which have been proposed in the literature would turn out to be either first or second class, but it is unlikely that that will be the outcome of an investigation. On the other hand it would be disappointing if it turns out that, beside the Turing Machines, there exists no other first class machine model. This would present strong evidence that there is no basis for the intuitive approach to complexity theory where one freely moves from Turing Machines to RAM's and vice versa. Again this is unlikely to become the result of our investigations. The truth will be somewhere in the middle.

The first class machine models are precisely the machines for which the fundamental hierarchy: **LOGSPACE** $\subsetneqq$ **NLOGSPACE** $\subsetneqq$ **P** $\subsetneqq$ **NP** $\subsetneqq$ **PSPACE = NPSPACE** $\subsetneqq$ **EXPTIME** $\subsetneqq$ $\subsetneqq$ **NEXPTIME** etc... represents the complexity hierarchy for this model. Here the liberal interpretation of the Invariance Thesis would suffice also. However, since it seems always to be the case that a model which satisfies the liberal interpretation of the Invariance Thesis for all modes of computation also satisfies the strict interpretation we have selected the strict reading for the definition of the first machine class. For us the first machine class represents the class of reasonable sequential devices.

For the second machine class it is questionable whether there exists any justification for this class as being reasonable at all. It seems that the marvelous speed-ups by the parallel models of the second machine class require severe violations of basic laws of nature (see [122]); stated otherwise, if physical constraints are taken into account, all gains of parallelism seem to be lost [10,106]. So naturalness can never be a motivation for selecting precisely this particular class of parallel devices to become the representative one. Instead the motivation is drawn from the intrinsic stability of this class, its large number of members, and the variety of mechanisms by which this power can be obtained. And, as I shall argue in the sequel, there exists at least one second machine class model which is a reasonable model of a device we use in our daily life.

# 2. SEQUENTIAL MODELS.

## 2.1. Turing Machines.

Turing Machines have established themselves as the standard machine model in contemporary complexity theory. Only in the related area of Analysis of Algorithms this role is taken over by the RAM model. Since the Turing Machine model is very well known I will abstain from giving formal definitions. Instead I will illustrate how this model behaves as a machine class.

It turns out that there does not exist any model which can be called *The Turing Machine model*. There exists a large variety of models, depending on features like structure, dimension and number of tapes, numbers of heads on a tape, restrictions on the use of tapes, the availability of head-to-head jumps etc. Every particular set of choices from these features leads to another machine model.

If even this basic model already shows a chaotic proliferation of subspecies, why is it then the case that this model is so popular, and that complexity theory as a whole seems not to suffer from this variability ? It becomes clear at this point that I have still some unpaid debts. In section 1.2.4 I have defined the First Machine Class using Turing Machines as a reference class, but, as indicated above, there exists no such standard model but an entire family of models. So I must first provide the reader with a more specific model of the Turing Machines. Take the familiar version with one dimensional tapes, one head on each tape, one read-only input tape, one write-only, one-way output tape, and $k \geq 2$ two-way infinite work-tapes. I must show that all further Turing Machine models satisfy the Invariance Thesis. These models should simulate each other with polynomial overhead in time and constant factor overhead in space; moreover the required simulations should achieve both overheads at the same time. These simulations would make it clear that the concept of the first machine class indeed does not depend on the particular type of Turing model chosen.

There remains just one troublesome question: *Is the invariance thesis indeed true for all Turing models* ?

### 2.1.1 Formalization.

In the Turing Machine model the memory is structured in the form of a finite collection of *tapes* which consist of *tape-cells* each capable of storing a single symbol from the *work-tape alphabet*. The cells are arranged in the structure of a finite dimensional grid in Euclidean space. So cells have addresses which can be represented by integer vectors. For each tape there exist one or more communication devices connecting the finite control with the tape. These devices are called *heads*. Each head is positioned at some tape cell. Several heads may be located at the same cell.

In the program of the Turing Machine the various types of instructions have been integrated into a single type: the machine reads the ordered set of symbols on the work-tapes which are currently scanned by the ordered sequence of heads. Depending on these symbols and the internal state of the finite control, the machine will overwrite the scanned symbols with new ones, move some (possibly all) of the heads to an adjacent cell on the corresponding tape and proceed to a new state in the finite control.

In the simplest case the machine has only one tape, with tape alphabet $\Sigma$, and a set of internal states $K$. The program of the finite control now consists of a set of quintuples $<q,s,q',s',m> \in K \times \Sigma \times K \times \Sigma \times \{L,0,R\}$. Here the set $\{L,0,R\}$ denotes the set of possible head moves : Left, stay put or Right. The meaning of this quintuple is: *if in state* q *the head is scanning symbol* s *then print symbol* s', *perform move* m *and proceed to state* q'. In this single tape model there is no special input or output tape. The input is written on the unique tape in the initial configuration with the unique head scanning the leftmost input symbol. If one wants this model to produce output one obtains such output by an ad-hoc convention from the final configuration (for example, the output consists of all non-blank tape symbols written to the left of the head in the final configuration).

If we denote configurations of the single tape machine in the format $\$\Sigma^*K\Sigma^*\$$, with the state symbol written in front of the currently scanned tape symbol, the transitions between two successive configurations are described by a very simple Context Sensitive Grammar. This opens the way to encode Turing Machine computations in a large variety of combinatorial structures, like tilings [34,56,93] or regular expressions [111], leading to a large collection of *Master reductions* in Computation Theory. The popularity of the single tape model is based on the use in reductions and other simulations.

In the more complicated models the set $\Sigma$ is replaced by a power $\Sigma^k$ where k denotes the number of tapes. The set of moves becomes also a k-fold Cartesian product, where moreover each set of moves is adjusted to the dimension of the tape the corresponding head is moving on. In case several heads are moving on a single tape a new coordinate must be added in order to let the heads "feel" that they are looking at the same square: the coincidence pattern of the heads which can be modeled as a partition of the set of heads.

The coordinates of an instruction always can be split in an *observation* and a *action* part: in the above case the first state and symbol represent the observation and the rest represents the action. If a single observation can lead to at most one action the machine is deterministic and otherwise the machine is nondeterministic. Nondeterminism is bounded by definition.

The tape alphabet contains a special symbol called *blank* to represent the contents of a cell which has never been visited so far by a head (and also is no part of the input). Tapes being completely homogeneous it is unimportant where in the initial configuration a head is positioned on a work-tape, provided the entire work-tape is blank, and that (in the case of many heads on a tape) all heads are located at the same cell.

Since it makes no sense to read left of the first input symbol and since the output tape (if present) is one-way it can be presumed that in- and output tapes are semi-infinite; their cells are indexed by the non-negative integers. The model can contain semi-infinite work-tapes as well.

There exist several restricted types of work-tapes which have obtained special names: A *Stack* is a semi-infinite work-tape with the special property that when the head makes a left move, the previous contents of the cell are erased (*popping the stack*). After a write the head can move right (*pushing a symbol on the stack*). The head can not move left of the origin but it can feel the bottom of the stack and test it for emptyness. A *Queue* is a semi-infinite tape with two right-only heads. The first head is a write-only head, whereas the second one is a read-only head. The second head therefore can read (only once) everything the first head has written before. Also a queue can be

tested for emptyness. A *counter* is a stack with a single letter alphabet. Its purpose is to count a number n by storing a string of n copies of its tape symbol , and testing whether the number equals zero by checking whether the stack is empty. The push and the pop move now correspond to an increment and a decrement of the corresponding integer.

For higher dimensional tapes it makes a difference whether the heads can move only along the edges in the grid or whether they can proceed in diagonal steps as well. Fast moves which have been considered are fast rewinds (a head proceeds in one step to its original position) or head-to-head jumps (one head moving in one step to the position of another).

### 2.1.2 Simulation Overheads.

Most results on simulations of enhanced Turing Machines on more simple ones are well known and documented in the literature. I will list a number of results of this type and make a few comments. I will omit in this list the trivial real-time constant factor space simulations which exist between a Turing Machine model and a more enhanced version: adding new features will never increase time or space consumption as long as the additional features are not used.

Rather than specifying the entire model I will in the formulation of the overheads just mention the relevant features. In the list I also provide a key reference.

**a) 1 tape ≤ 2 stacks** ( **real-time & space Lin** ) Minsky [71]
**b) 1 stack ≤ 2 counters** ( **time Expl & space Expl** ) Minsky [71]
**c) 1 tape ≤ 2 counters** ( **time** exp(exp(k.n)) **& space** exp(exp(k.n))) Minsky [71]
**d) m counters ≤ 1 tape** ( **real-time & space** log(n) ) Vitányi [121]
**e) m tapes ≤ 1 tape** ( **time** $k.n^2$ **& space LIN** ) Hopcroft & Ullman [43]
**f) m tapes ≤ 2 tapes** ( **time** k.n.log(n) **& space LIN** ) Hennie & Stearns [40]
**g) multi-head tapes ≤ single-head tapes** ( **real-time & space Lin** ) Fischer et. al. [22]
**h) multi-head tapes + jumps ≤ single-head tapes** ( **real-time & space Lin** ) Kosaraju[53]
**i) 2-dim tape ≤ 1 tape** ( **time Pol & space** $k.n^2$ ) Folklore
**j) 2-dim tape ≤ 1 tape** ( **time Pol & space** k.n.log(n) ) Folklore
**k) 2-dim tape ≤ 2 tapes** ( **time** $k.n^{3/2}$.log(n) **& space** k.n.log(n) ) Stoss[114], see also [7]
**l) 2-dim tape ≤ 2 tapes** ( **time Pol & space Lin** ) Hemmerling [39]

The results a) and b) together imply c) with 4 counters and a single exponential blow-up in time and space. But in fact two counters suffice (at the price of one more exponential blow-up). This is Minsky's result that two counter machines are universal.

Result d) is less well known. It is based on an oblivious simulation of a single counter on a single tape, where oblivious means that the position of the head of the simulator depends on the time only and not on the contents of the tape. The simulation uses a redundant number representation and an extremely ingenious method of simulating a recursive procedure without a stack. It is easy to see that, given an oblivious simulation of one counter one can as well simulate k counters on the same tape, without loss of space or time.

Result e) belongs to the folklore, whereas f) again is based on an oblivious simulation of a single tape on two tapes. This simulation uses a technique of tape segmentation where a tape is decomposed in regions of exponentially increasing sizes, which are located farther and farther away from the current head position and which are serviced with an exponentially decreasing frequency.

Result g) originally was given by Fisher, Meyer and Rosenberg [22]. It was later improved upon in the sense that less tapes are required for the simulation by Leong & Seiferas [55]. The extension h) with jumps has a long history but this problem was wrapped up finally by Kosaraju [53].

The least known case, as far as the Invariance Thesis is concerned is given by the simulations i),and j). The standard simulations of two dimensional tapes on single dimensional tapes require more than constant factor space overheads, in combination with time overheads $k.n^3$ and $k.n^2.\log(n)$. The same is true for the time efficient simulation k) by Stoss which is again based on tape segmentation. The only reference known to me where a simulation with a constant factor space overhead is given is Hemmerling's report at Greifswald [39]; the only textbook showing this reference is the text by Wagner and Wechsung [123].

Simulation l) is based on the idea to store address-content records but not using absolute but relative addresses related to the previous record. This requires that the entire visited region of the higher dimensional tape is traced by a path which visits every cell, from every cell proceeds to one of its neighbors on the grid, and finally is no longer than a constant multiple of the number of cells visited. It should not visit a cell more than a constant number of times. It turns out that a full traversal of a depth-first search tree in the subgraph of the grid which represents the visited area of the tape has these properties.

The simulations i),j),k) and l) all extend to higher dimensions. The time overheads in i), j) and k) become respectively $k.n^{d+1}$, $k.n^2.\log(n)$ and $k.n^{2-1/d}.\log(n)$ , whereas the space overhead in i) becomes $n^d$ . The space bounds in j) and k) and both bounds in Hemmerling's simulation are independent of the dimension.

### 2.1.3 Other Turing machine simulations.

An important property of the Turing Machine model is the constant factor speed-up both in space and time. By replacing the tape alphabet $\Sigma$ by its k-th power $\Sigma^k$ one can compress k symbols into a single square. This makes it possible to design a machine which works k times as fast and uses k times less space. This transformation is a reverse of the simulations which prove that a tape alphabet with two symbols is universal, since in this simulation arbitrary symbols are encoded by binary strings. Clearly the anomaly of the constant factor speed-up disappears if one assigns a weight to every step of a Turing machine proportional to the amount of information processed. After compressing k symbols into one the manipulation of the symbol would become k times as expensive in a weighted time measure. On the other hand the constant factor speed-up is very convenient - one can get rid of all constant factors implied by O(f) terms.

The above constant factor speed-up for time and space is a well known property of Turing machines. What is less well known is the original proof given by Hartmanis & Stearns[38]. Most proofs which you can find in textbooks start with a pre-processing stage were the input is

condensed (k symbols into one); this pre-processing requires linear time provided the machine has two tapes; otherwise pre-processing requires quadratic time which leads to a constant factor speed-up in time which holds only for time bounds larger than $n^2$. Next the original computation is simulated, but in order to be certain that in a single move of the simulation one can simulate k steps of the original computation one must scan the neighborhood first and subsequently perform some updates to this neighborhood as well. This then leads to a speed-up of *k moves simulated by 6 moves*. Hence, in order to achieve a true speed-up by a factor k one must condense the tape by a factor 6.k rather than k. In the original proof one keeps one block of the tapes in the finite control and this allows a *k moves for 1 move* simulation.

Another important issue is whether overheads as indicated in the above scheme are optimal or not. Traditional folklore has the result that simulation of two tapes on a single one requires quadratic time overhead, since palindromes can be recognized in linear time on two tapes, but for a single tape model time $\Omega(n^2)$ is needed, as can be shown by a crossing sequence argument. Note that this result does not imply anything on the optimality of a square overhead for simulating two work tapes by a single work tape or related problems. Such problems have been solved only recently using the technique of Kolmogorov Complexity ; see for example [57,58,63,64,77].

The following inclusions express some well known relations between different resources:

m) **T-TIME**( f(n) ) $\subsetneq$ **NT-TIME**( f(n) ) $\subsetneq$ **T-TIME**( $2^{k.f(n)}$ )

n) **T-SPACE**( f(n) ) $\subsetneq$ **NT-SPACE**( f(n) ) $\subsetneq$ **T-SPACE**( $f(n)^2$ )

o) **T-TIME**( f(n) ) $\subsetneq$ **T-SPACE**( f(n)/log(f(n)) ) Hopcroft, Paul & Valiant [42]

Relations m) and n) provide the known bounds on the relation between determinism and nondeterminism. For the time bounded classes the bounds are essentially trivial. The second inclusion in n) represents Savitch's theorem [94]. For the case of linear time it is known that the first inclusion in m) is a proper one [78]. If beside a time bound f(n) also a space bound s(n) is known Wiedermann [126] has recently improved the upper bound in m) to:

**NT-TIME&SPACE**( f(n),s(n) ) $\subsetneq$ **T-TIME**( $f(n).s(n)^2.2^{k.s(n)}$ )

This also improves upon the time bound which results from the proof of Savitch's theorem:

**NT-TIME&SPACE**( f(n),s(n) ) $\subsetneq$ **T-TIME**( $2^{k.s(n).\log(f(n))}$ )

Result o) represents the unique result which indicates that space may be a more powerful resource than time, but beware: the proof given in the reference is valid for one-dimensional tapes only. The result extends to higher dimensional Turing Machines but then it requires a new proof [80]. There exist also versions for the RAM model [79] and for the Storage Modification Machine [33].

The proof of o) involves several techniques: first the Turing Machine computation is made *block-respecting* with a block-size $k(n) \approx f(n)^{2/3}$ ; this means that both time and tapes are divided into segments of size k(n) such that segment boundaries are only crossed at the end of a time slot. The overhead for this part of the simulation is linear in time and space. Computing onwards for one block of time now only requires knowledge of the tape segments scanned at the beginning of a block period. The initial configurations of these blocks are collected into a *computation graph* which turns out to be a directed graph with indegree m+1 where m equals the number of tapes. A

correspondence is made between solutions of a *pebble game* on this directed graph and space efficient simulations of the original computation. The fact that the graph of $m := f(n)^{1/3}$ nodes can be pebbled with m/log(m) pebbles now yields the logarithmic gain in space efficiency in o) .

For the case of single tape Turing Machines , denoted $\mathbf{T_1}$ , a better bound is known:

$\mathbf{T_1}$-**TIME**( f(n) ) $\subsetneq$ $\mathbf{T_1}$-**SPACE**($\sqrt{f(n)}$ ) , provided $f(n) \geq n^2$ ; see [76].

The result requires a constructability condition on the timebound f(n) . Contrary to the Hopcroft, Paul & Valiant theorem the space-efficient simulation can be made polynomial time as well: one has in fact:

$\mathbf{T_1}$-**TIME**( f(n) ) $\subsetneq$ $\mathbf{T_1}$-**TIME&SPACE**( $f(n)^2$,$\sqrt{f(n)}$ ), Ibarra & Moran [44].

The above result extends also to the case of a Turing machine with a two-way read-only input tape and one one-dimensional work tape. The simulation overheads become slightly different, due to the fact that the position of the input head must be stored. The space overhead there becomes $\sqrt{(f(n).\log(n))}$ . The time overhead remains polynomial ( $f(n)^2$ ) , and the constructability condition has to be adapted as well.

Another difference with the Hopcroft, Paul & Valiant result is that the simulation extends to nondeterministic computations as well. There one has:

$\mathbf{T_1}$-**NTIME**( f(n) ) $\subsetneq$ $\mathbf{T_1}$-**NTIME&SPACE**( $f(n)^{3/2}$,$\sqrt{f(n)}$ ) Ibarra & Moran [44].

Recently this result was further improved to

$\mathbf{T_1}$-**NTIME**( f(n) ) $\subsetneq$ $\mathbf{T_1}$-**NTIME&SPACE**( f(n),$\sqrt{f(n)}$ ) Torenvliet & v. Emde Boas [116], Liskiewicz & Lorys [61],

by using nondeterminism for a guess-and-certify strategy comparable to the strategies used in several efficient simulations of parallel models.

It is interesting to combine the above result with the breadth-first simulation of Wiedermann. One obtains a result with a sublinear function in the exponent [126]:

$\mathbf{T_1}$-**NTIME**(f(n)) $\subsetneq$ **T-TIME**($f(n)^2.2^{k.\sqrt{f(n)}}$) .

## 2.2 Register Machines.

### 2.2.1 The variety of models of register machines.

Register machines [14] have become the standard model in computation theory for the analysis of concrete algorithms. On the one hand these register machines can be recognized as the model of a standard computer which has been reduced to its minimal essential instruction repertoire; on the other hand these devices don't suffer from a doom hanging over all real computers: they have neither finite word length nor finite address space.

The popularity of the model seems to be based on the fact that, even though no real life RAM's exist, all models seem to be equipped with a highly efficient compiler for some ALGOL like language, which enables the authors of the various textbooks to use whatever high level language features they feel to be required for the description of their algorithms. It is clear that, in order to lead to realistic estimates on the complexity of the algorithms dealt with, some knowledge on the efficiency of the compiled code is presupposed but I won't go into the details of this issue.

In the basic RAM model the machine consists of a finite control where a program is stored, one (or more) accumulator register, denoted Acc, an instruction counter, and an infinite collection of memory registers R[0], R[1], ..... The accumulator and the memory registers have an unbounded wordlength, but in any configuration a finite number will be stored in these registers.

The instruction repertoire of the RAM (and that of other machines we will meet in the sequel) can be divided in four categories:

1) instructions which steer the flow of control: **goto, accept, reject, halt,** and various types of conditional jumps (**if** condition **then goto** , **if** condition **then skip** ) ; here the condition is a simple test (accumulator = 0 ? , accumulator > 0 ? , accumulator = R[i] ? , .... ) . The meaning of these instructions is self explanatory. For all other types of instructions after completion the next instruction in the program will be executed.

2) instructions for input and output: Read and Print. Depending on the precise model this instruction either can load entire integers to and from arbitrary registers, or these transports can require the accumulator as an intermediate storage location (read accumulator, print accumulator). An even more restricted form of transput is the model where a read instruction will input a single bit from the input channel and jump to a subsequent instruction label, depending on whether the bit equals 0 or 1.

3) instructions for transport of data between accumulator and memory. The name RAM ( = Random Access Machine) indicates the main feature here: the machine uses indirect addressing. There exist three types of load instructions and two types of store instructions:

| | | | |
|---|---|---|---|
| **Load** = i | Acc := i | | |
| **Load** i | Acc := R[i] | **Store** i | R[i] := Acc |
| **LoadI** i | Acc := R[R[i]] | **StoreI** i | R[R[i]] := Acc |

4) instructions performing arithmetic. In the weakest model one only has the Increment instruction: Acc := Acc+1 ; the standard model has both addition and subtraction, where the second argument is fetched from memory: **Add** j Acc := Acc + R[j] . More powerful models extend this arithmetic with multiplication and division. For these powerful models it also becomes possible to treat the register contents as bit-strings rather than numbers; in this interpretation we have instructions for concatenation and bit-wise Boolean operations.

In the sequel we will denote by SRAM the model which only has the successor instruction; RAM denotes the standard model with addition and subtraction, whereas MRAM also has the multiplication and division. If the Bitwise Boolean instructions are also available we introduce another B in the name (for example MBRAM ). Prefixing with N denotes nondeterminism.

As with the Turing machines we have a huge collection of different models, but, as it turns out, also for every model at least two different methods of measuring time and space consumption of a RAM computation. In the *Uniform measure* every instruction is counted as one step, regardless the size of the values operated on. In the *Logarithmic measure* every instruction is given a weight equal to the sum of the logarithms of all the quantities involved in the instruction; this includes the lengths of addresses involved in direct or indirect addressing.

Nondeterminism can be incorporated in the model by legalizing a hideous error made by the beginning programmer: multiple use of a label. If the machine jumps to a label which occurs twice in the program one of the occurrences is chosen nondeterministically.

In real life computers a major breakthrough was von Neumann's idea to replace the specially wired program in the finite control by a program in memory which might even be modified by the computer during the course of its computation. A similar idea brings us to the RASP model [14] (Random Access Stored Program). Here the memory has been divided into the registers with even and with odd addresses. Two adjacent addresses together store an instruction, with the operation code in the even address and the argument address in its odd neighbor. By writing in such an odd location the machine can obtain the effect of indirect addressing without having it in its instruction repertoire. It has been shown that for the purpose of complexity theory the RAM and the RASP are fully equivalent (they simulate each other in real time with constant factor space overhead) and therefore I will not deal with this model in the sequel. The proof can be found in textbooks like [1].

It should be observed that even the weakest model (the successor RAM) remains universal as a computing device when the instructions for indirect addressing are removed. In this case all addresses accessed during a computation are represented explicitly in the program. The machine will use only a fixed finite set of its registers during the computation, and can be described as a finite control equipped with a fixed number of counters. Minsky has shown that two registers suffice for universal computing power [71]. On the other hand the same result shows that the uniform space measure is not a proper complexity measure; it fails to obey the Blum axioms [4].

The use of indirect addressing leads to two less desirable effects. It becomes possible to use registers in a very sparse way, leaving big gaps in the memory where nothing has happened. It also becomes less reasonable to presume that at the start of the computation all registers are properly initialized at zero; what if someone else has used the machine before you ? If you have to initialize your memory who is going to pay for this ? Both these problems have been considered and solved [1], but it is not clear whether the solutions proposed are measure independent.

### 2.2.2 Time measures for RAM's

As indicated above there exist two time measures for the RAM model. In the *Uniform Measure* every instruction is counted for one unit of time. In the *Logarithmic Measure* every instruction is charged for the sum of the lengths of all implicit or explicit data manipulated by the instruction. This length is based on a size function on numbers based on the binary logarithm with some ad-hoc clause for small values:

size(n) = **if** $n \leq 1$ **then** 1 **else** $\lfloor \log_2(n) \rfloor + 1$ **fi** .

For example, the cost of an **AddI** j instruction, with the effect Acc := Acc + R[R[j]] will be something like size(x) + size(j) + size(R[j]) + size(R[R[j]]) , where x denotes the current value of the accumulator.

In the sequel the use of uniform or logarithmic measure will be indicated by prefixing in complexity class denotations the words **TIME** or **SPACE** with **U** and **L** respectively.

It is clear that a computation in the uniform time measure costs less than in the logarithmic measure. The gap between the two measures may become as large as a multiplicative factor equal to the size of the largest value involved in the computation. How large this value can grow depends on the available arithmetic instructions. This leads to the following simulations:

a) **SRAM-Utime ≤ SRAM-Ltime** (**time** n.log(n) )
b) **RAM-Utime ≤ RAM-Ltime** ( **time** $n^2$ )
c) **MRAM-Utime ≤ MRAM-Ltime** ( **time EXP** )

The same inclusions hold for the corresponding nondeterministic classes. As can be seen the gap between uniform and logarithmic measure for the MRAM is so large that no simulation with polynomial time overhead can be guaranteed. Indeed, the MRAM in the uniform time measure has to be discarded from the realm of reasonable models: it is a member of the second machine class [3].

It is an elementary programming task to see that the bitwise Boolean instructions can be simulated in time polynomial in the length of the operands by the models which don't have them. This leads to the inclusions:

d) **SBRAM-Utime ≤ SRAM-Utime** ( **time** n.log(n) )
e) **SBRAM-Ltime ≤ SRAM-Ltime** ( **time** n.log(n) )
f) **BRAM-Utime ≤ RAM-Utime** ( **time POL** )
g) **BRAM-Ltime ≤ RAM-Ltime** ( **time POL** )
h) **MBRAM-Utime ≤ MRAM-Utime** ( **time POL** )
i) **MBRAM-Ltime ≤ MRAM-Ltime** ( **time POL** )

For d) and e) the standard tricks to translate between numbers and their bit-patterns don't work; one must explicitly preserve the binary strings in arrays. The simulation h) is not an invocation of the above observation but a rather deep result about the second machine class [3]: bitwise Boolean instructions are not needed for the power of parallellism in the presence of both

addition and multiplication. Again the results hold also for nondeterministic models.

The absence or presence of multiplicative and parallel bit-manipulation operations is of relevance for the correct understanding of some results in the area of analysis of algorithms. Frequently for the analysis of concrete problems with low complexities it is assumed that the machine used can perform innocent looking multiplicative instructions on small values, since these instructions occur as well on real world computers. It is also common to use the uniform time measure. At the same time one is interested to obtain complexity bounds which are precise up to logarithmic factors. It is therefore relevant to know to which extent these complexity bounds depend on the precise instruction repertoire provided.

Assume by way of example that we are given the instructions which perform left- and right shifts of bit-patterns (multiplications and divisions by powers of 2 ). Assume moreover that we are allowed to perform these operations on arguments $\leq n^k$ , where $n$ denotes the largest value in the input and $k$ denotes a fixed constant. It turns out to be possible to obtain a large collection of other multiplicative or bit-manipulation instructions at a time overhead $O(1)$ in the uniform time measure by use of the basic technique of table look-up. Any operation can in principle be stored in a table of size $n^{2k}$ ; the restricted multiplication allows us to simulate indexing in an $n^k$ by $n^k$ two-dimensional array, and therefore the operator can be performed in time $O(1)$ once the table has been precomputed. But for many specific operations of a sufficiently "local" nature the same result can be obtained using a table of size $O(n)$ (or even size $O(\sqrt{n})$ ) by splitting the operands in pieces of length $\log(n)/2$ or even less and recomputing in constant time the total result from the piecewise results by some combining function which depends on the instruction. Since the table size becomes negligible compared to $n$ so does the time required for precomputing its values. The parallel bit-wise logical operators and ordinary multiplication have this locality property, and so does the instruction which counts the number of 1-bits in a binary number.

This observation is a generalization of the strategy used by Schmidt and Siegel in [100]. It shows that there hardly exists an "innocent" extension of the standard RAM model in the uniform time measure; either one only has additive arithmetic, or one might as well include all reasonable multiplicative instructions on small operands at once.

The relation between the various RAM models and Turing Machines can be obtained from a number of well known results found in the textbooks. There exist a number of improvements which are less well known. First the overhead for simulating a TM on a RAM:

**j)** $T \leq$ **SRAM-Utime** ( **real-time** ) Schönhage [104]
**k)** $T \leq$ **SRAM-Ltime** ( **time** k.n.log(n) ) Aho, Hopcroft & Ullman [1]
**l)** $T \leq$ **RAM-Utime** ( **time** k.n/log(n) ) Hopcroft, Paul & Valiant [41]
**m)** $T \leq$ **RAM-Ltime** ( **time** n.loglog(n) ) Katajainen ea. [50]

Next the overheads for the reverse simulations:

n) **SRAM-Utime ≤ T ( time** $n^2.\log(n)$ **)** Cook & Rechow [14]

o) **RAM-Utime ≤ T ( time** $n^3$ **)** Cook & Rechow [14]

p) **MRAM-Utime ≤ T ( time EXP )** (I can't do any better)

q) **SRAM-Ltime ≤ T ( time** $n^2$ **)** Cook & Rechow [14]

r) **RAM-Ltime ≤ T ( time** $n^2$ **)** Cook & Rechow [14]

s) **RAM-Ltime ≤ T ( time** $n^2/\text{loglog}(n)$ **)** Wiedermann [124]

t) **MRAM-Ltime ≤ T ( time POL )** trivial

The above results are obtained by allocating address/value pairs of the RAM on Turing Machine work-tapes and estimating the time needed for processing these structures. The improvement s) is obtained by ordering these records by total length rather than by their addresses. Clearly most time in the simulation is spent by searching for a record on the linear tape. If higher dimensional tapes are used this overhead is reduced to the effect that a factor $n^2$ in the simulation overhead can be replaced by $n^{1+1/d}$ where d denotes the dimension of the worktapes used [123].

### 2.2.3 Space measures for RAM.

It is well known that RAM space should not be measured by counting the number of registers used. Minsky's result that a two counter machine has universal computing power [71], shows that counting the number of RAM registers used during a computation does not yield a complexity measure satisfying the Blum axioms [4]. Still it is the case that this uniform space measure is commonly used in the theory of Analysis of Algorithms. This use can be justified on the basis that in concrete algorithms the intermediate results stored in registers always are bounded in terms of the input values: either by a polynomial, or by a simple exponential function. Estimating the sizes of intermediate results represents an essential part of the analysis of algorithms in the area of algebraic and number theoretical computations.

In the theory of machine models every RAM register is charged for the size of its contents. Such a size function is invoked in the definition of the Logarithmic time measure anyway, so why not use the same function for the space measure as well. Having available such a size function, there are several ways to proceed. A rather crude way is to charge every register used for the size of the largest value produced during the entire computation. A more refined method is to charge every register for the largest value stored there during the entire computation. The latter heuristic leads to an expression:

$$\text{Space used} = \sum_{i=0}^{\text{maxaddr}} \text{size2}(i, \max(i)) \qquad (*)$$

where maxaddr is the index of the highest address accessed during the computation, and max(i) is the largest integer ever stored in the register with index i during the computation. In the above expression for the space measure I have added the additional parameter i to the size function size2 in order to make it possible to consider size functions which are dependent on the address of the register.

An even more refined method would be to look at individual RAM configurations. Charge every register used in some configuration for the size of the value currently stored there and compute the sum of these register costs for the registers which are currently used. The maximum of

this sum, taken over all the configurations then would become the space measure. In this measure a copy of a large value which is repeatedly transported from one register to another one, after which the first register gets cleared is counted only twice, whereas in the measure expressed by the above formula (*) it gets charged for every register it visits. This difference, however, turns out to be non-problematic, and therefore I will restrict myself to space measures described by the above expression (*). See Wagner & Wechsung [123] for several more alternatives.

The traditional RAM model supports the use of uninitialized storage: registers which have never been accessed before contain a value zero. It is quite possible that there are registers with index in the range 0,...,maxaddr which are never accessed during the computation, and therefore in the above formula something must be said about the space consumed by those unused registers. In a survey paper [95] W. Savitch considers the size function:

$$size_W(i,x) = \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } \lfloor \log(x)+1 \rfloor \textbf{ fi } = size(x)$$

The resulting measure charges unused registers for an amount of one bit for the value 0 stored there. As a consequence it becomes possible to consume exponential space during polynomial time by performing a program like:

addr:=1 ; **for** i **from** 1 **to** n **do** addr:= addr + addr ; R[addr]:= 1 **od**

A more generally accepted measure (see for example [1]) uses the same size function for used registers but gives the unused registers for free:

$$size_S(i,x) = \textbf{if } R[i] \text{ is unused } \textbf{then } 0 \textbf{ else } size(x) \textbf{ fi}$$

This solves the anomaly stated above but introduces a new problem which seems to have been overlooked: *How to simulate a RAM on a Turing Machine with constant factor overhead in space* ? The standard trick of storing address-value records on a work tape requires additional space for the addresses, whereas a sequential allocation of all registers in the range 0,...,maxaddr requires space proportional to Savitch's measure. So none of the two proposed measures leads to invariance of space in an evident way. Hence, taking the standard simulation of a RAM on a Turing Machine, as a point of departure we arrive at the following size measure:

$$size_b(i,x) = \textbf{if } REG[i] \text{ is unused } \textbf{then } 0 \textbf{ else } size(i) + size(x) \textbf{ fi}$$

We claim that $size_b$ represents the intuitively correct way of measuring space on a RAM. It is not the definition of the logarithmic space measure for RAM's one encounters in the literature. Most textbooks provide either no formal definition at all, or provide a definition based on $size_S$. This is due to the emphasis in these textbooks on time complexity. In the interesting case of a register access within an instruction using indirect addressing, the size of the address is charged for in the time measure. This is achieved by charging for the contents of the register used for the indirect

addressing, but not as an intrinsic part of the cost of accessing the register which is reached by the indirection.

Basing the RAM space measure on $size_b$ has several advantages. First observe that the constant factor space overhead for simulation of a Turing Machine on a RAM remains intact, although the standard simulation which stores tape cells in consecutive registers has to be abandoned. This simulation would introduce an $\Omega(S.\log(S))$ space overhead due to the lengths of the addresses of $S$ registers. But by using the standard trick of "one tape = two stacks" , and by storing a single stack in a single register, a simulation with constant factor space overhead is obtained (at the price of increasing the time overhead by a factor $S$ or $S^2$ depending on the time measure used). So our proposal validates the Invariance Thesis.

Another advantage of the use of $size_b$ is connected with the simulation of uninitialized storage as suggested in [1], Exercise 2.12. When using $size_s$ the space overhead becomes $\Omega(S.\log(S))$ whereas it is a constant factor space overhead when using $size_b$ . A similar observation can be made about the standard method of compacting sparsely used registers into a dense set by the creation of address-value pairs on the RAM itself.

Having chosen the right measure the space complexity of the RAM becomes fully equivalent to that of the Turing machine, and therefore I abstain from mentioning any specific further simulation results.

### 2.2.3.1 The problematic simulation of a RAM on a Turing Machine.

The reader might ask at this point whether the difference between $size_s$ and $size_b$ is really important or not. Surprisingly it is. As is shown in [108,109] the Invariance Thesis, which is evidently valid if we use the measure based on $size_b$ , becomes problematic if the definition based on $size_s$ is used. Using the extremely complicated simulation which I will sketch below, it is shown that for deterministic off-line computations a Turing machine can simulate a RAM based on $size_s$ with constant factor overhead in space. However, since this simulation requires exponential overhead in time, this is insufficient to show that the RAM model (again with the $size_s$ space measure) is a first class machine model ! It can be shown that for on-line computations a constant factor overhead simulation does not exist, and the case of nondeterministic computations presents us with a wide open problem.

In this subsection I present a sketch of this complicated simulation. The straightforward simulation does not work here since no space is reserved for the address parts of the address-value records which must be stored on the worktape of the Turing Machine. On the other hand, the problem evaporates as soon as the size of the data stored in RAM memory exceeds or is just proportional to the space needed by these addresses. Therefore the problem will only arise in the situation where small chunks of data (say characters) are stored in registers with large addresses, encoding in this manner information about the address rather than on the data itself. This situation is illustrated by the following recognition problem:

$$L_0 = \{w_1\$w_2\$...\$w_k\$w_0 \mid w_i \in \{0,1\}^* \textbf{ and } w_0 \in \{w_i \mid 1 \leq i \leq k\}\ \}$$

It is not hard to see that on a RAM the above language can be recognized on-line by reading the words $w_i$ as addresses, and storing a 1 in the corresponding registers. If we consider the typical case where the k words $w_i$ in the input have length $|w_i| = m$, the above algorithm will consume space O(m+k) on a RAM with space measure based on $size_s$, whereas it requires space O(m.k) if the measure based on $size_b$ is used. The latter amount of space is a provable lower bound for Turing Machine on-line recognition, since the Turing Machine on-line acceptor must write a full description of its input on its worktapes before it can process the last word $w_0$. Clearly this lower bound argument collapses if off-line processing is allowed, and it is not difficult to construct an off-line Turing Machine acceptor which runs in space O(m+log(k)).

The space measure based on $size_s$ seems to underestimate the true space consumption when one uses a sparse table. If such a situation arises in actual computing the problem may be solved by hashing techniques. Hashing will map a sparse set of logical addresses on a much denser set of physical addresses. Hence, rather than storing the address-value pairs directly, we represent the addresses by their hash codes for a suitable hash function, and reconsider the resulting space requirements. Two observations now can be made:

1) In order that the computations are not disrupted by confusion of registers the hash function used should be perfect with respect to the addresses used during the computation; it should be a 1-1 function if restricted to those arguments on which it will be evaluated.
2) If the actual number of addresses used during the computation equals k, and if the hash function has a range c.k for some constant c, the physical address-value records can be allocated sequentially on a worktape, so the hash codes no longer need to be stored. Using a variable size format for the remaining value records, it can be achieved that inside this hash table an amount of space is consumed proportional to the space as measured by the $size_s$ space measure.

The above idea leads to the following problem on perfect hashing which must be solved: Given a set A of k elements of a Universe U = {0,...,u-1} of size u. Find a perfect hash function f which scatters this set A completely into a hash table of size c.k for some fixed constant c.

Functions satisfying the above requirements evidently exist as set theoretical objects. But for the problem at hand, these functions themselves should satisfy some space requirements as well. In the above situation, the RAM to be simulated may quite well consume no more space than O(k+log(u)), and consequently, the manipulation of the hash function should not require more space than O(k+log(u)) as well. Otherwise the advantage gained by its use will be lost during its manipulation. So the final constraint on the above function f reads: it should be of program size O(k+log(u)) and require space O(k+log(u)) for its evaluation.

Hash functions satisfying the first two requirements were constructed in a very explicit manner by Fredman, Komlós & Szemerédi [26], but their construction did not satisfy the last requirement. Mehlhorn [68] showed that for the program size the upper bound could be met and even be improved to a tight O(k+loglog(u)) bound but his function required far more space for evaluation. In [108,109] C. Slot and the present author succeeded in matching the Mehlhorn bound in

combination with an O(k+log(u)) evaluation space. The required space-efficient logical-to-physical address translation therefore exists. See also [27,45,69,100].

The next question is how the simulator can ever hope to find it. Since we are looking for a deterministic simulation the method of guessing a hash function and using it is not allowed. One has to try out a large collection of hash functions and select a good one. Being good here means being perfect with respect to the set of addresses in the simulated computation. Whether a hash function is good must be certified. In the process of certification it is not possible to make a list of all addresses encountered and keeping track of possible collisions under the hash function, since this will re-introduce the space consumption we a trying to eliminate. However, it is possible within the available space bounds to check for every slot in the range of the hash function that there exists at most one address which is hashed onto this slot. Since the computation is deterministic this process can be repeated for every slot individually. At this point the simulation breaks down for both probabilistic and nondeterministic modes of computation. Under these modes of computation one never knows whether a second run of the computation will access the same registers as the first run.

As long as the hash function has not collected its certificate of being perfect, the simulation has to cope with the possibility that the hash function is in fact not perfect and that the simulator therefore may confuse registers. It may provide wrong answers, it may consume far more space than it should, and it may even diverge on bounded memory. The first problem is not a serious problem since no answer of the simulation will be believed before the hash function has been certified, and the second problem can be solved using Savitch's trick of incremental space. It is the third problem which causes the greatest trouble. Since the space consumption O(k+log(u)) may turn out to be o(log(n)) where n denotes the input length, detection of loops on bounded storage by counting is not allowed. Luckily this problem has been solved already by Sipser [107], who proposed a simulation by backward search from the accepting configuration, under which loops are prevented and no extra space is consumed.

Details of the above simulation and the theory on which it is based can be found in [109]. I hope that the sketch above suffices to convince the reader that the simulation is quite complicated, requires a lot of re-computations, causes an exponential time overhead, and fails to solve the problem for alternative modes of computation. As such it provides no evidence that the Invariance Thesis is satisfied if the RAM space measure is based on $size_S$ . This measure should therefore be faded out from the literature.

## 2.3 Storage Modification Machines.

The Storage Modification Machine (SMM) is a model introduced by Schönhage already in 1970. The model is similar to the model proposed by the Soviet mathematicians Kolmogorov & Uspenskii (KUM) in 1958 [52], but for a precise comparison I refer to the discussion in Schönhage's paper. For the purpose of this section I will base myself on the 1980 version published in SIAM. J. Comput. [104]. The author advocates his model as *a model of extreme flexibility* and therefore *it should serve as a basis for an adequate notion of time complexity*.

The model resembles the RAM model as far as it has a stored program and a similar flow of control. Instead of operating on registers in memory it has a single storage structure, called a $\Delta$-*structure*. Here $\Delta$ denotes a finite alphabet consisting of at least two symbols. A $\Delta$-structure $S$ is a finite directed graph each node of which has $k := \#\Delta$ outgoing edges which are labeled by the $k$ elements of $\Delta$. The main difference between the SMM and the KUM can now be explained: the KUM operates on undirected instead of directed graphs.

There exists a designated node $a$ in $S$, called the *center* of $S$. There exists a map $p^*$ from $\Delta^*$ to $S$ defined as follows: For the empty string $\lambda$ one has $p^*(\lambda) := a$, and otherwise $p^*(wa) :=$ the end-point of the edge labeled $a$ starting in $p^*(w)$. The map $p^*$ does not have to be surjective; however, nodes which can not be reached by tracing a word $w$ in $\Delta^*$ starting from the center $a$ will turn out to play no subsequent role during the computations of the SMM, and therefore nodes may as well be assumed to have disappeared when they become unreachable.

The program of the Storage Modification Machine consists, similar to the RAM program, on the one hand of flow of control instructions (**goto**, **accept**, **halt**, ...), and transput instructions (read and print - in this case a read will input a single bit and act like a conditional jump, depending on the value of the bit read), and on the other hand instructions which operate on memory - in this case a $\Delta$-structure $S$. There exist three types of instructions of the latter type:

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

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

**if** $v = w$ ( **if** $v \neq w$ ) **then ....** : the conditional instruction (conditional jump suffices); here it is tested whether the nodes $p^*(v)$ and $p^*(w)$ coincide or not.

Simple it may look like the effect of a Storage Modification Machine in action is extremely hard to trace. An alternative name for the machine, coined by Knuth, is *pointer machine*, and indeed: its semantics is as complicated as any pointer based algorithm. The model suffers from the *paradoxes of assignment*: After **new** $w$ it is not necessarily true that $p^*(w)$ denotes the new

node, and also the Hoare formula $p^*(v) = x$ { **set** w **to** v } $p^*(w) = x$ is invalid. But for the purpose of this chapter it are the complexity properties of the model which are relevant and not the problem of proving correctness of SMM programs.

In order to discuss complexity we need a time and a space measure. For the time measure there exists only one candidate: uniform time measure. One might consider to charge an instruction according to the length of the paths traced during the instruction, but since the arguments of the instructions are denoted explicitly in the program this length is a constant, independent of the $\Delta$-structure currently in memory. Therefore such a weighted measure will differ from the uniform measure by no more than a constant factor which is fully determined by the program.

Schönhage has not introduced a space measure. The reasonable candidate seems to be the number of nodes in the current $\Delta$-structure, and, in fact, this corresponds to the definition given in [33]. This definition has, however, some problematic aspects. There exist simply too many $\Delta$-structures of n nodes. Schönhage presents for the number X(k,n) of $\Delta$-structures of n nodes over an alphabet $\Delta$ of size k the following estimates:

$$n^{nk-n+1} \leq X(k,n) \leq \binom{kn+1}{n} . n^{nk-n+1} / (nk+1)$$

From the above estimate it follows that in space n one can encode O(n.log(n)) bits of information rather than O(n.log(k)) bits as on n squares of a Turing Machine tape with alphabet $\Delta$. This leads to a similar situation as we have seen in section 2.3.3.1: consider the following on-line recognition problem:

$L_1 = \{I_1;....;I_k\&v=w \mid I_1;...;I_k$ encodes a straight-line SMM program such that after performing it $p^*(v) = p^*(w)$ }

It is evident that an SMM can recognize this language in space O(n) where n denotes the number of **new** instructions in the straight-line program $I_1;...;I_k$. On the other hand a Turing Machine on-line recognizer must have written a full description of the $\Delta$-structure generated by program $I_1;...;I_k$ when it reads the &-symbol, and therefore by an information theoretical argument it must have consumed by that time space $\Omega(n.\log(n))$.

This example leaves open the situation for the case of off-line computations but it is not difficult to see that also there O(n) nodes on an SMM are capable of storing as much information as O(n.log(n)) tape squares on a Turing Machine. The basic technique for such a simulation is the construction of a $\Delta$-structure which represents a cycle of O(n) nodes with in each node a pointer to some other node in the cycle. The latter pointer stores in this way O(log(n)) bits of information, and it is a matter of simple programming to show that this respresentation can be used and updated, using no more than O(log(n)) additional nodes. A similar technique works also for the KUM. For more details see [119].

Schönhage [104] has compared his model both with (multi-dimensional) Turing Machines and with some RAM models in the uniform time measure. The results are:

**a) SMM $\approx$ SRAM-Utime ( real-time )**
**b) T $\leq$ SMM ( real-time )**

As a consequence, the SMM can be simulated on a standard RAM in the logarithmic time measure with n.log(n) time-overhead. The simulations for a) are straightforward. In the proof Schönhage introduces another simplified RAM model, the main advantage of which is that the instruction code is address free (instead the model uses a special purpose address register). This reduced RAM model again is real-time equivalent to the SRAM in the uniform time measure.

Simulation b) is easy for one-dimensional Turing Machines. In the higher dimensional case a tape-segmentation trick is used.

A final advantage claimed for the SMM model is that it supports an implementation of integer multiplication which runs in linear time. So if everybody in the area of analysis of algebraic and numerical algorithms would embrace this model all occurrences of the multiplication complexity function M(n) could be replaced by simple O(n) expressions. Attractive it might look like, it seems that this proposal so far has few followers in the literature. For a discussion on the connection between theoretical fast multiplication algorithms and practical machines I refer to the recent note by Schönhage in the EATCS bulletin [105].

There are a few theoretical results known about the SMM model. In [33] Halpern et. al. prove a version of the Hopcroft, Paul, Valiant result that time T(n) can be simulated in space O(T(n)/log(T(n))). For this purpose they use the space measure determined by the number of nodes, which, as we indicated above, differs from Turing Machine space by a logarithmic factor. Since the time measure seems to differ from Turing Machine time by the same logarithm one still can maintain that it is a similar result.

More recent is the result presented by Schnitger [101] who seems to indicate that there exists a non-linear gap in time between the SMM and the KUM. Since the main distinction between these two models is that Schönhage uses a directed graph as storage structure, where Kolmogorov & Uspenskii use an undirected graph, the implicit bound of $\#\Delta$ on the out-degree for the SMM becomes a bound on the indegree for the KUM as well. Therefore there can't exist too many short paths leading to some particular node. Schnitger introduces a well designed on-line real-time recognition problem which exploites this feature, and establishes a non-linear lower bound for this problem on the KUM, depending however on some unproven conjecture about communication complexity.

The SMM represents an interesting theoretical model, but, in the context of the above observations, its attractiveness as a fundamental model for complexity theory is questionable. Its time measure is based on uniform time in a context where this measure is known to underestimate the true time complexity. The same observation holds for the space measure for the machine; under the plausible definition of the space measure the SMM does not belong to the first machine class as defined in section 1.2.4. This problem can be solved by the introduction of a logarithmic space measure where an n-node $\Delta$-structure is charged for space n.log(n) , but such an unnatural definition for the space measure would make the model even less attractive.

## 2.4 Networks and non-uniform models.

All machine models discussed so far have the property that a single machine, or a single program operates on inputs of arbitrary length. Occasionally the machine, in performing some simulation, will first have to decide the length of the input, and first then, after having set up the proper simulation parameters, the "real" simulation will start.

Networks are a typical representative of an alternative approach to the complexity of decision problems and/or function evaluation. In this approach the complexity investigated is of a more structural kind. Given some problem, say deciding membership in the language $L$, one first reduces this problem to a family of finite problems $L_n := L \cap \Sigma^n$. These finite problems are easy with respect to machine computations since their solution can be programmed using table-look-up. Instead one investigates the size $s_n$ of the devices which solve these finite problems. The quantity $s_n$ as a function of the input size $n$ now becomes a measure of the complexity of the problem $L$.

From the perspective of computation on arbitrary length input $x$ the process of computation now clearly is decomposed in two steps: from the input length $n = |x|$ determine the n-th device $M_n$; next let $x$ be processed by $M_n$ in order to produce the required answer.

Intuitively the time complexity of $L$ is bounded by the complexity of these two stages together. But note that neither of these two stages has a time complexity which is measured by $s_n$ as a function of $n$; for the first stage it is at best a lower bound: retrieving an object of size $s_n$ requires time $\Omega(s_n)$. For the second stage $s_n$ does not even shed any light on the time required to operate $M_n$ on $x$.

At this point the restrictions which make the theory meaningful come to our rescue. The first restriction is that the mapping $n \mapsto M_n$ is *Uniform* (it should be computable in Polynomial time and/or Logspace). The second restriction is that the devices $M_n$, are extremely easy to evaluate, as is the case with formulas and networks.

Without the first condition it is possible to have undecidable problems with trivial network complexity: take any undecidable problem $U \subset \mathbf{N}$, and consider the language $L_U$ defined by $x \in L_U$ iff $|x| \in U$. For every length $n$ there exist very simple devices which accept or reject all inputs of length $n$; it is the problem of deciding which of the two to use which is undecidable.

There exists a meaningful alternative for making the mapping $n \mapsto M_n$ uniform; one can also consider Turing Machine computability relative to some suitable oracle. See Schnorr [102] or Karp & Lipton [49] for more details.

Without the second restriction the theory cannot provide any complexity bounds above $\log(n)$: take a Universal Turing Machine with a program for $L$; this is an object of constant size. If one adds a description of the length $n$ in $\log(n)$ bits one can obtain a finite device prepared for dealing with inputs of length $n$. Asymptotically, the $\log(n)$ bits of $n$ dominate in the description of the size of this object.

There exists an extensive literature on the subject of network-like models and non-uniform complexity. A detailed treatment falls outside the scope of this chapter. I will restrict myself therefore to a few results which connect this theory to the machine models introduced elsewhere in this section. For more details I refer to the source references. An example of a textbook on complexity theory which is based on network-like models is the 1976 text by Savage [92].

### 2.4.1 The Network model.

A *Logical Network* or *Boolean Circuit* is a finite labeled directed acyclic graph. *Input nodes* (*Output nodes*) are nodes without ancestors (successors) in the graph. Input nodes are labeled with the names of input variables $x_1,...,x_n$ . The internal nodes in the graph are labeled with functions from a finite collection, called the *basis* of the Network. Here the condition is enforced that the number of ancestors of an internal node is equal to the number of arguments of its label; moreover a suitable edge labeling establishes a 1-1 correspondence between these ancestors and the arguments of this label.

By induction one defines for every node in the network a *function represented by this node* : for an input variable labeled by $x_j$ this is the projection function $(x_1,...,x_n) \rightarrow x_j$ ; an internal node with label $g(y_1,...,y_k)$ for which the k ancestors represent the functions $h_1(x_1,...,x_n)$, ....., $h_k(x_1,...,x_n)$ represents the function $g( h_1(x_1,...,x_n), ....., h_k(x_1,...,x_n) )$ . A function $f(x_1,....,x_n)$ is computed by the network iff it is represented by an output node.

The most important size functions for networks are:

-- depth: the length of a longest path in the graph between an input node and an output node
-- size : the total number of nodes in the graph

It is common to restrict oneself to networks over a basis of logical functions consisting of monadic and binary functions only: There exists four monadic functions, among which only the negation function is important. There exist 16 binary logical functions, among which the and, the or, and the xor are the best known ones in the network theory. Implication is another function but its role in Network theory is incomparably small compared to its role in logic and mathematics. It is a known result that there exist binary functions like the nand function, which form a complete basis by themselves. Still the most common network basis consists of three functions: and, or , and negation. If the negation is omitted the network only represents *monotone* functions and will be called a *monotone network* .

Every node in the network can be ancestor of an arbitrary number of successors. If the number of successors is restricted to 1 the network becomes a tree, and one talks about formulas rather than a circuit.

The complexity measures introduced below depend on the logical basis, and on the fact whether one deals with networks or formulas. In the latter situation one speaks about *formula complexity* rather than *network complexity* . The logical basis is implied by the context; default is the standard three element basis, and it is moreover not hard to see that for network complexity the effect of the choice of a basis is at most a constant factor in depth and size; for formulas the story is different, see [54,83].

For a given finite collection of logical functions $f_1(x_1,...,x_n)$, ....., $f_k(x_1,...,x_n)$ the network complexities $size(f_1,...,f_k)$ and $depth(f_1,...,f_k)$ are defined to be the minimal size or depth of a network such that the entire set is computed by this network. If $\mathbf{F}_n$ for $n \in \mathbf{N}$ is a family of n-argument functions it is possible to investigate the asymptotic behavior of $size(\mathbf{F}_n)$ and $depth(\mathbf{F}_n)$ as a function in n . If L is a language over $\{0,1\}^*$ then the network

complexities of L are defined as the network complexities of the family $\mathbf{F}_n$ where $\mathbf{F}_n$ for every n consists of the characteristic function $L_n$ of the set $L \cap \{0,1\}^n$ .

The following results are known :

| | | |
|---|---|---|
| **a)** | size( $\mathbf{F}_n$ ) $\leq$ # $\mathbf{F}_n$ .$2^n$ | (easy) |
| **b)** | size( $\mathbf{F}_n$ ) $\leq$ $2^n/n(1+\varepsilon)$ for # $\mathbf{F}_n$ = 1, n $\geq$ N($\varepsilon$) | (Lupanov [62]) |
| **c)** | depth( $\mathbf{F}_n$ ) $\leq$ n + 1 | (McColl & Patterson [66]) |
| **d)** | depth( $\mathbf{F}_n$ ) $\leq$ size( $\mathbf{F}_n$ ) | (trivial) |
| **e)** | size( $\mathbf{F}_n$ ) $\leq$ # $\mathbf{F}_n$ . $2^{\text{depth}(\mathbf{F}_n)}$ | (trivial) |

It is known that the bound in b) is asymptotically tight; by enumeration of all possible networks one shows that there for every $\delta > 0$ only a fraction of all functions can be computed by a circuit of size $\leq 2^n(1-\delta)/n$ since there are not enough of those small circuits. It is a much harder problem to prove lower bounds in network complexity for explicit functions. There the largest lower bound for network complexity shown for an individual explicit function equals 3n + o(n) [5]. For formula complexity one has an $\Omega(n^2/\log(n))$ lower bound [72] . For monotone networks exponential lower bounds for specific functions have been established since 1985 : [2,85].

It is generally assumed that establishing lower bounds for network complexities is about the strongest statement one can make about the intractability of some problem. It is also the most difficult task faced by the complexity theorist, and so far progress has been much smaller than one might hope for.

### 2.4.2 Relation with Turing Machine complexity.

The finite control of a Turing machine can be considered to be some finite automaton which can be implemented using a Boolean Network. The size of this network tells something of the complexity of the program of the machine, whereas the depth of this network indicates how fast the machine can perform a single step of its computation, assuming that the machine has to be implemented using logical gates. Still these connections between Networks and Turing Machines are not the ones investigated in the literature; there one considers the relation between time and space measures for languages recognized by Turing Machines and the Network complexity for these languages as defined in the previous section.

Consider the time-space diagram which encodes the complete computation of some Turing Machine on some input of length n . Without loss of generality we can assume that the machine consumes both maximal time and space on every input of length n, so the size of this diagram does not depend on the precise input. The structure of the diagram is such that it is filled with characters, such that each character is determined completely by the characters located at a small local set of neighbors. These dependencies are directed forward in time. It is not hard to design a Boolean Network of size proportional to the size of the time-space diagram which evaluates (an encoding of) the character at position (s,t) in the space-time diagram by an amount of circuitry for this location which depends of the program only. In this way one can establish a result:

**f)** size $(\{L_n\}) \leq P.T(n).S(n)$ (Savage [92])

here $T(n)$ and $S(n)$ denote time- and space complexity of the Turing Machine whereas $P$ denotes its size.

In the above simulation a large amount of circuitry is wasted for calculations the purpose of which is to copy the contents of an unvisited tape cell from one configuration to the next. These local circuits could be eliminated, provided one would know in advance where the heads are residing on the tapes. But that is the problem which has been solved by the Hennie-Stearns simulation where a k-tape Turing machine was obliviously simulated on a two-tape machine. Exploiting that idea one obtains:

**g)** size $(\{L_n\}) \leq P.T(n).\log(T(n))$ (Pippenger & Fischer [82], Schnorr [102])

It follows from the construction that both results are proved using a uniform family of circuits. Conversely, it is not hard to see that for logspace-uniform Networks, a Turing Machine can construct the Network and subsequently evaluate in time polynomial in the size. A tighter bound has been established by Pippenger [81]. There exists also a tight relation between the depth measure and the space complexity of Turing Machines. For spacebounds $s(n) \geq \log(n)$ which are constructable in space $O(\mathbf{max}\ \log(n),\log(s(n)))$ Borodin [8] has shown:

**h)** depth $(\{L_n\}) \leq O(s(n)^2)$ , when $L \in$ **NSPACE**$(s(n))$

Moreover for these functions $s(n)$ circuits of depth $s(n)$ can be evaluated in space $O(s(n))$.

More recently Networks have become used as a tool for characterizing complexity of parallel computation as well. This connection has lead to the introduction of two hierarchies of classes:

$\mathbf{NC}_k$ = the class of languages L recognized by logspace uniform networks of polynomial size and depth $O(\log(n)^k)$

$\mathbf{SC}_k$ = **T-TIME&SPACE(** **POL**,$O(\log(n)^k)$ **)**

Both $\mathbf{NC}_k$ and $\mathbf{SC}_k$ form a hierarchy within **P** . These hierarchies have been named after Nick Pippenger and Steve Cook respectively. See Cook [12,13] for further details.

## 3. THE SECOND MACHINE CLASS.

This section is dedicated to a survey of the second machine class which consists of those models which satisfy the Parallel Computation Thesis. This thesis states that parallel time and sequential space are polynomially related. As will be seen in the sequel models don't have to use parallellism in order to become a second machine class member. In fact the Alternating models in section 3.2 are sequential devices with a modified mode of acceptance, whereas the models in section 3.3 are sequential devices which operate on huge objects in unit time. The really parallel models first appear in section 3.4 . As will be seen in section 4 the boundaries of the second machine class in the realm of parallel machines become rather vague. Slight modification of the rules of the game leads to even more powerful models. There are also weaker models of parallel devices, one of which is introduced in section 4.1 .

The current literature on parallel models is growing explosively and consequently in a chapter like the present one only a fraction can be covered. I am quite aware of the fact that even some main stream developments in the theory of parallel computation are not covered at all. I have not dealt with parallel models consisting of RAM's or finite state devices in an arbitrary interconnection pattern [31], models for systolic computation [32,67], cellular automata [115], or the subject of relative quality of one interconnection pattern with respect to another [28,70]. I have not looked into the relation between parallel models and network resource bounds [12,13]. The subject of write-conflict resolution methods is only mentioned [18,20,21,59]. I have also paid no attention to what happens if additional resource bounds are enforced on parallel devices, like polynomial bounds on the number of processors used or other hardware bounds [17]. The reader interested in such subjects is referred to the indicated literature.

For the class of models covered in the survey I establish a tight connection between simulations of sequential devices and transitive closure algorithms which perform in essence a brute force try-out everything simulation. This connection will be explained in section 3.1. Since all efficient simulations presented in sections 3 and 4 of this chapter are based on this connection the "unreasonableness" of the parallel models presented is clearly exposed. If parallel models are efficient by simulating sequential devices in such a crude manner something must be wrong. Still the very same connection enables me to place these models in a common perspective.

## 3.1 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. There exists a generic translation between space bounded computations and paths in a suitably constructed configuration graph. For reasons of simplification we will assume from now onwards that all spacebounds are $\Omega(\log(n))$. For a given input $x$, a given Turing Machine $M$ and a given space bound $S$ we can form the graph $G(x,M,S)$ of all configurations $c$ of machine $M$ which use space $\leq S$; 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 space bound $S$ there exist an unique node corresponding to the initial configuration on input $x$.
2. Assuming that a suitable notion of acceptance 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 $2^{c.S}$, 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 $\leq 1$; if $M$ is nondeterministic then the outdegree of some nodes can be $\geq 2$, but for a suitable restriction of the Turing Machine model the outdegree can be assumed to be $\leq 2$ 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 $\leq 2^{c.S}$.

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 expensive method for simulating a single computation, it is exactly this transitive closure algorithm which is hidden in almost all proofs that some particular parallel machine obeys the Parallel Computation Thesis.

### 3.1.1 Transitive closure algorithms and PSPACE-complete problems.

Various designs of a 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 ; $N = 2^{c.S}$ . If we let M(x,M,S)[i,i] = 1 for all $i \leq N$ , then the transitive closure of M can be computed by c.S squarings of M(x,M,S) using 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 time bound T on the computation is given this time is reduced to O(S.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.

Next we 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 $\leq 2^{order}$ :

```
proc path = (int order, node i,j) bool:
    if order = 0 then i = j or edge(i,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 path(c.S,init,final), where init and final are the unique initial and final configurations in G(x,M,S) ; given a time bound T the initial parameter order can also be chosen to be log(T) . With recursion depth c.S (respectively log(T) ) and parameter size O(S) this recursive procedure can be evaluated in space $O(S^2)$ (respectively O(S.log(T)) ) ; this is the main ingredient of the proof of Savitch's theorem [94] which implies that **PSPACE = NPSPACE** :

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

A third formulation of the transitive closure algorithm yields the **PSPACE**-completeness of the problem **QUANTIFIED BOOLEAN FORMULAS (QBF)** [112]:

**QUANTIFIED BOOLEAN FORMULAS:**

**INSTANCE:** A formula of the form $Q_1x_1....Q_nx_n[\ P(x_1,....,x_n)\ ]$ , where each $Q_i$ equals $\forall$ or $\exists$ , and where $P(x_1,....,x_n)$ is a propositional formula in the boolean variables $x_1,....,x_n$ .

**QUESTION:** is this formula true ?

The idea is to encode nodes in G(x,M,S) by a boolean valuation to a sequence of $k := c.S$ boolean variables. From the proof of Cook's theorem which establishes the **NP**-completeness of **SATISFIABILITY** ( see [11] or [29] ) one obtains the existence of a propositional formula $P_0$ in 2.k variables : $P_0(x_1,....,x_k,y_1,....,y_k)$ , where the variables $x_1,....,x_k$ encode node i , the variables $y_1,....,y_k$ encode node j and $P_0$ expresses that i=j or there exists an edge from i to j . One now can define by induction a sequence of quantified propositional formulas $P_d$ such that $P_d(x_1,....,x_k,y_1,....,y_k)$ expresses the presence of a path of length $\leq 2^d$ between node i and node j . In a naive approach the formula $P_d$ would include only existential quantifiers, and $P_d$ 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:

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

Substituting for $x_1,....,x_k$ and $y_1,....,y_k$ the codes of the initial and final node in G(x,M,S) in $P_k(x_1,....,x_k,y_1,....,y_k)$ we obtain a closed quantified boolean formula, the truth of which 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-theoretical interpretation of **QBF** several authors have investigated the end-game analysis of games inspired by real life games. A useful intermediate game is

**GENERALIZED GEOGRAPHY** [99] ; from there one can reach **HEX** on arbitrary graphs and even **HEX** on the traditional hexagonal board ; see [19,87] . For people interested in the position of **CHECKERS, CHESS** and **GO** : these games are not on the list because, after earlier **PSPACE**-hardness results [24,60], they turn out to be even more difficult than **PSPACE** [25,88,89] .

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** [30] , one has nowadays examples of group theoretical problems which have been shown to be **PSPACE**-hard [46,47] . Together with problems which encode **PSPACE**-bounded computations in a more direct way (like Reif's **GENERALIZED MOVER'S PROBLEM** [86] ) this has led 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.

### 3.1.2 Establishing membership in the second machine class.

Suppose that we are faced with some parallel or otherwise powerful machine model X and that we want to prove that X is a member of the second machine class: **X-PTIME = X-NPTIME = PSPACE**. The above observations provide us with some tools for proving this result.

The inclusion **PSPACE** $\subseteq$ **X-PTIME** can be shown by inventing a polynomial time algorithm on X which solves a **PSPACE**-complete problem like **QBF** . An alternative is to show that on X one can implement one of the transitive closure algorithms from the previous section in polynomial time.

For the inclusion **X-NPTIME** $\subseteq$ **PSPACE** one uses in most cases a guess and verify method. A nondeterministic machine is used to guess a trace of an accepting X-computation on the given input. Such a trace consists of instructions executed and memory values stored during this computation but in general not all information stored in memory during the X-computation can be written down in polynomial space. Therefore one writes down enough information from which the remaining memory contents can be reconstructed. The reconstruction is done by a recursive procedure which reflects the machine architecture of model X ; the recursive procedure tells how the memory contents at time t depend on those at time t-1 . Care has to be taken that all arguments for this procedure and also its values can be written down in polynomial space.

Using the trace and the recursive procedure it becomes possible to certify the trace as being consistent: for every conditional instruction the correct branch has been selected.

If one has established for two models X and X' that they are both members of the second machine class, one has indirectly shown that X and X' simulate each other with polynomial overhead in time. In the literature only a few instances of direct simulations establishing such polynomial time overheads are given. One example is the refined analysis of the power of various models of vector machines by Ruzzo in the unpublished report [90], where explicit overheads for simulation of Alternating Turing Machines and vector machines are given. Another example can be found in the paper by van Leeuwen & Wiedermann on Array Processing Machines [120] where time overheads for simulation on the SIMDAG and a reverse simulation are determined.

## 3.2. The Alternation model.

### 3.2.1. The concept of Alternation.

The concept of *Alternation* [9] leads to machine models which obey the Parallel Computation Thesis without providing any intrinsic parallellism at all. As a computational device an Alternating Turing Machine is very similar to a standard Nondeterministic Sequential Turing Machine; only its mode of acceptance 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 a 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 acceptance 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 the quality of which 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 [9] 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 computation makes sense for virtually every machine model and is not restricted to Turing Machines. For example, it makes sense to consider alternating finite automata, alternating PDA's

etc. It is clear that for every device some power may be gained by proceeding to the level of alternation but whether the gain is real depends on the specific model considered. For example, in the case of finite automata, the languages recognized by Alternating Finite Automata are still Regular languages; the gain is a potentially doubly exponential reduction of the size of the automaton.

Time ( Space ) consumed by an alternating computation is measured to be the maximal Time (Space) consumption along any branch in the computation tree. Another complexity measure which was considered by Ruzzo [91] for the Alternating Turing Machine is the minimal size of an accepting computation tree. An accepting tree is a subtree of the full computation tree which yields a certificate that the input is accepted; it includes all successors of a Universal node but for an Existential node only one successor with quality **accept** is included in the accepting computation tree. This measure is related both to nondeterministic time in the sequential model and parallel time and leads to various interesting classes with simultaneous resource bounds. I refer to the paper for more details.

### 3.2.2 Relation with sequential models.

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:

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

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 :

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

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

In the sequel I will indicate why the above equalities are true. First consider the inclusion $\mathbf{APTIME} \subsetneqq \mathbf{PSPACE}$ . 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 guess the valuation for the quantified variables where the universally (existentially) quantified variables are guessed in a universal (existential) state ; these values 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 logspace-bounded 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, and since the time needed for a single sweep is bounded by the square of the size of the list of configurations the running time for this procedure is polynomial.

The reverse inclusion **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 K by K table of 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 occurrence 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. This last observation breaks down for nondeterministic devices M and therefore we cannot obtain the inclusion **NP** $\subseteq$ **ALOGSPACE** in this way.

Again by a simple padding argument one obtains the equality **APSPACE** = **EXPTIME** , as an easy consequence of the equality **ALOGSPACE** = **P** .

### 3.3. Sequential machines operating on huge objects in unit time.

The first machine model 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 & Stockmeyer [84] , shortly later succeeded by the MRAM of Hartmanis & Simon [36,37] . 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 of 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 [36]. More recently it has been established that the combination of multiplication and division, in absence of the bitwise Boolean instructions suffices as well; see [3,103]. This result shows the power of a purely arithmetical model.

In this section I will illustrate the power of these models by a model which has been obtained by moving in the other direction: stressing the pure symbol manipulation instructions and dropping the powerful arithmetic. This is the EDITRAM model proposed in [110] as a model of the text editor you may have in mind while editing texts behind your terminal. I will present an outline of the proof of the validity of the Parallel Computation Thesis for the EDITRAM and indicate the connection with the proofs for the earlier models.

#### 3.3.1. The EDITRAM model.

In the EDITRAM we extend the standard RAM with a fixed finite set of text-files. Standard arithmetic registers can be used as cursors in a text-file. 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 string1 by string2 in a text-file, 6) concatenation of text-files, 7) copying of segments of text-files as indicated by cursor positions, and 8) deletion of segments of text-files as indicated by cursor positions.

In the systematic string replacement instruction 5) the arguments string1 and string2 are to be presented by literals in the program; substitution of the contents of an entire text-file for a single

character would allow a doubly exponential growth of the size of text-files, which is more than we are aiming for.

The time complexity of the model is defined by using the uniform time measure for the arithmetic registers. So an edit instruction is charged one unit of time. A reasonable alternative would be to use the logarithmic time measure with respect to the arithmetic registers. Then an edit instruction would be 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 text-file. Since in our model the growth of a text-file is at most exponential the two measures will be polynomially related. This observation no longer would be valid if we would allow that entire text-files are substituted for strings by the replace operation; the doubly exponential growth of the text-files will require exponential time in the logarithmic measure so the uniform and the logarithmic measure no longer are polynomially related.

### 3.3.2 The EDITRAM is a Second Machine Class device.

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 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 worktape. 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 the length of the values of the arithmetic registers is linearly bounded by the time, 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 worktape indeed represents an accepting computation.

From the meaning of the individual instructions one can write down a recursive procedure which expresses 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, the entire textfile, up to the given position must be re-computed by recursive calls. This is the most complicated case in the description of this recursive procedure. For details I refer to [110].

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 Vectormachine 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 a 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 remains bounded by the standard trick enabling the machine to use consecutive registers in its memory.

In general these simulations achieve the required space-bound 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 $Q_1x_1....Q_nx_n[\ P(x_1,....,x_n)\ ]$ 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:

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

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

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:

$$(0 \vee 0) \Rightarrow 0\ ,(0 \vee 1) \Rightarrow 1\ ,(1 \vee 0) \Rightarrow 1\ ,(1 \vee 1) \Rightarrow 1\ ,$$

$$(0 \wedge 0) \Rightarrow 0\ ,(0 \wedge 1) \Rightarrow 0\ ,(1 \wedge 0) \Rightarrow 0\ ,(1 \wedge 1) \Rightarrow 1\ ,$$

$$(\neg 0) \Rightarrow 1\ ,\quad (\neg 1) \Rightarrow 0\ ,\quad (0) \Rightarrow 0\ ,\quad (1) \Rightarrow 1\ .$$

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 text-file (this 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 [110]. This completes the proof of the second inclusion.

The proof of the corresponding inclusions in the original papers on vector machines and MRAM's 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 bit-string consisting of the $2^K$ bit-strings representing the first $2^K$ integers, separated by markers, and of bit-strings to be used as masks for extracting a given bit-position 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 [3].

### 3.4. Machines with true parallellism.

In this section we consider models which provide visible parallelism by having multiple processors operate on shared data and/or shared channels. In these models computation can proceed either synchronously (all processors perform a step of the computation at the same time, driven by a local clock) or asynchronously (each processor computes at its own speed).

There exists a number of possible strategies for resolving the write conflicts which arise when several processors attempt to write in the same location of shared memory. In the Priority Write strategy the processor with the lowest index will succeed in writing in such a shared location and the other values will be lost. 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 becomes the winner but it is nondeterministically determined which one). For yet another approach to multiple writes see [18]. The computational power of the parallel RAM models based on these resolution strategies has been compared in [20,21,59] under the unrealistic assumption that the individual processors have arbitrary computing power. In these investigations the priority model has established itself as the most powerful one.

There are several methods of controlling the creation of parallel processors. Some models have a large or even infinite collection of identical processors which operate 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.

All together there are a large number of possible models and variations thereof. In this section I will discuss only a few of these models which indeed can be shown to obey the Parallel Computation Thesis. As will become clear in section 4 some minor modifications of the models suffice to increase the power of these machines.

#### 3.4.1. The SIMDAG model.

In the SIMDAG model [31] (Single Instruction, Multiple Data AGregate) 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 any local processor with a signature less than the threshold value transmitted performs 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 sub-processors activated is bounded by an exponential of the running time of the SIMDAG 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 becomes the winner in case of a write conflict.

For a device like the SIMDAG it is necessary to restrict the power of the arithmetic instructions involved. Otherwise, as we will see in the sequel, the machine may become too 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 in a single write instruction by letting each local processor write a 1 in a fixed register in global memory if its bit equals 1 whereas the value 0 is not transmitted to the global memory.

Based on the above incomplete description I can sketch 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 re-computation of intermediate results.

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

This algorithm first loads the K by K matrix M(x,M,S) which was introduced in section 3.1 in global memory. For convenience assume that K is a power of 2 . The matrix is constructed by letting processor i + K.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 as a shift operator) can determine first the values of i and j and next unravel these values as bit-patterns 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.j+K.K.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 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.

### 3.4.2 The Array Processing Machine.

Our next model, the Array Processing Machine (APM) was proposed by van Leeuwen and Wiedermann [120] . It has been inspired by the contemporary vectorized supercomputers. 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 upper bounds of the vector argument and the mask respectively.

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

In their paper [120] 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 non trivial 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 can be easily obtained by the techniques used for other devices in this section. 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 will 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.

### 3.4.3. Models with recursive parallelism.

As an example of a parallel machine of a different character I mention the Recursive Turing Machine introduced by Savitch [96]. 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.

A similar model, based on the RAM is the k-PRAM described by Savitch & Stimson [98]. In this model a RAM-like device can create up to k copies of itself, which start computing in their local environment, while their creator is computing onwards. These copies themselves can create new offspring as well. Data are transmitted from parent to child at generation time by loading parameters in the registers of the offspring. Upon termination a child can return a result to its parent by writing a value in a special register of the parent which is read-only for the parent. The parent can inquire about this register whether his child has already written into it; if he reads the register before the child has assigned a value to it, the parent's computation is suspended. Using these features a parent can activate a number of children and consume the first value which is returned, aborting the computations of the remaining children which have not yet terminated.

Clearly models based on such local communication channels are much slower in broadcasting information to a collection of sub-processors and in gathering the answers. However, 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 sub-processors polynomial time suffices in case a complete binary tree is formed; 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 sub-processors.

It is not difficult to see that the above models support an implementation of the recursive version of the transitive closure algorithm from section 3.1 which runs in polynomial time. This explains why **PSPACE** is contained in the **//-PTIME** class for these models. The time overhead which used to be $n^2$ before becomes now $n^3$ due to the fact that the recursive algorithm contains also a loop over all intermediate nodes. This loop would require exponential time if performed sequentially, and therefore it is replaced by a recursive expansion where a call at order p produces an exponential number of calls at order p-1. The recursion depth of the procedure now becomes $\log(N)^2$ in stead of $\log(N)$.

The same recursive expansion trick is used in order to show that for these models **//-NPTIME** $\subseteq$ **//-PTIME**. A single nondeterministic computation of length T is replaced by $2^T$ deterministic computations relative to a choice string which is different for every copy. Transforming this idea into a complete proof exposes a number of complications. The final proof for this inclusion yields an $O(T^6)$ time overhead for elimination of nondeterminism. For the details I refer to the papers [96,98].

A crucial difference between the SIMDAG model and the Recursive models 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 models each processor, once being activated, performs its own program except for the impact of

communication with its parent 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.

This has consequences for the proof of the inclusion **//-PTIME $\subseteq$ PSPACE** for these models. Rather than writing down the entire trace of the computation and certifying it by a recursive procedure, the entire geneaological tree of machine incarnations is searched by a recursive procedure. Recursion depth is bounded by the polynomial bound on the running time of the parallel machine. A polynomial bound on the size of the recursive stack frames is obtained from the fact that the machine has no powerful arithmetic. It is crucial at this place that the communication between parent and offspring is entirely local; if offspring can write in global memory the proof breaks down.

A further complication results from the fact that a parent can abort some of its children before they have terminated; evaluating the result produced by such a child may lead to a divergent computation. Therefore all devices are extended with a counter keeping track of global synchronous time. Such counters are needed anyhow in order to let a parent know which one of its two children was the first to terminate. Again I refer for more details to the full papers [96,98].

One obtains the feeling that the models in this subsection are less "stable" than the powerful sequential models or the synchronous SIMDAG. This is also illustrated by the results in the final section of this chapter.

# 4. PARALLEL MACHINE MODELS OUTSIDE THE SECOND MACHINE CLASS

In this section I discuss several models which can not be classified as being either first or second class devices. First I discuss a parallel model which is much weaker than the models in the previous section. Next I will discuss the models which are more powerful than the second machine class members, including a model claimed to perform all computations in constant time.

## 4.1. A weak parallel machine.

The *Parallel Turing Machine*, (abbreviated PTM), introduced by Wiedermann in [125] should not be confused with the devices introduced by Savitch under the name *Recursive Turing Machines* in [96]. 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 Turing program for a machine with $k$ d-dimensional single-head tapes; however, instead of choosing 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 have met is related to the achievable degree of parallellism. 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 $q.S^k$ 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 Turing Machine 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 simulation 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 $O(S^k.\log(S))$ .

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 simulation 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 [125].

## 4.2. Powerful Parallel models.

There exist in the literature several models which seem to be more powerful than a second class machine. Fortune & Wyllie [23] have described a hybrid of the SIMDAG with a parallel machine based on branching, called P-RAM in [23] and called MIMD-RAM in my earlier survey [117] . For this model it has been established that **MIMD-RAM-PTIME = PSPACE** and **MIMD-RAM-NPTIME = NEXPTIME.**

Savitch [97] 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.**

N. Blum has written a short note argueing against the Parallel Computation Thesis [6]. His claim is based on a pair of models called the PRAM and the WRAM respectively. These models resemble the SIMDAG but instead of the priority strategy for write conflicts he uses the exclusive write strategy in the PRAM and common write strategy in the WRAM. For these models he claims the results **NEXPTIME** $\subseteq$ **WRAM-PTIME** and **DEXPTIME** $\subseteq$ **PRAM-PTIME** . The correctness of his claims is questionable since his proof requires the presence of powerful arithmetic as well.

Below I give a sketch of these models and the simulations involved.

### 4.2.1 The MIMD-RAM.

The MIMD-RAM which was introduced by Fortune & Wyllie [23] as a hybrid between the SIMDAG and the k-PRAM described in section 3.4. In the MIMD-RAM a machine can create offspring by forking, where the offspring will perform the instructions of its own program. Upon creation the offspring machine will start executing at the first instruction in its program. The subsequenct course of the computation is influenced by the initial value of the accumulator which has been set by its creator. The machine has standard additive RAM arithmetic.

Each offspring processor has its own local memory. The machines communicate furthermore by global memory. Global and local memories are standard RAM memories. Each device can both read and write in its local memory using the standard RAM instructions. For passing information upwards processors write in global memory. Simultaneous reads from global memory are allowed, but simultaneous writes are prohibited. The machine accepts if the oldest copy accepts by halting with 1 in its accumulator.

In the model a special input convention is used which supports the reading of an input of length n in time O(log(n)), so sublinear running times become meaningful.

The MIMD-RAM has the property that its deterministic version is a true second machine class device whereas its nondeterministic version provides us with a full exponential speed-up in time. This is expressed by the equalities **MIMD-RAM-PTIME = PSPACE** and **MIMD-RAM-NPTIME = NEXPTIME.** The proofs for these equalities can be sketched as follows:

**PSPACE $\subseteq$ MIMD-RAM-PTIME** : one proof consists of simulation of a 2-PRAM where the channels have been replaced by global registers. Another proof consists of a version of the transitive closure algorithm for the computation graph. Simultaneous writes are prevented by designing a fan-in algorithm for evaluation of the and or or of O(S) bits. In both cases the resulting program is a deterministic MIMD-RAM program.

**MIMD-RAM-PTIME $\subseteq$ PSPACE** : Due to the restrictions on the arithmetic in time T at most $2^T$ machines are created, each operating on at most $2^T$ registers and operate on values bounded by $2^T$. This implies that all arguments for a recursive procedure val(i,j,t) , which yields the value of register j in device i at time t , can be written down in polynomial space. A similar procedure can be obtained for the contents of the global memory. However, it is no longer possible to write down in polynomial space the entire trace of the computation since each device performs its own instructions.

Instead the trace of the computation is built together with the tree of recursive calls; a certain value is present in some register as a result of the execution of previous instructions which have been executed because of the presence of certain values in other registers at some earlier time etc. It is not hard to see that an Alternating RAM can evaluate this recursion guessing and certifying the trace at the same time. From the fact that the MIMD-RAM involved is deterministic it follows that only the true values and instructions performed can be certified. It also follows that this alternating RAM consumes time polynomially bounded by T . Hence the above inclusion follows.

**MIMD-RAM-NPTIME $\subseteq$ NEXPTIME** and **MIMD-RAM-NLOGTIME $\subseteq$ NP** : These inclusions can be shown by brute force simulations: the overhead in simulating a nondeterministic MIMD-RAM computation by writing down an entire record of its computation is at most exponential in time.

**NP $\subseteq$ MIMD-RAM-NLOGTIME** and **NEXPTIME $\subseteq$ MIMD-RAM-NPTIME** : These inclusions are shown by designing a nondeterministic MIMD-RAM algorithm which recognizes an **NP**-complete problem in logarithmic time. This yields the first inclusion and the second will follow by a similar argument or by a padding argument. For the **NP**-complete problem we select the problem **BOUNDED TILING** [93], called **SQUARE TILING** (GP13) by Garey & Johnson [29]:

**BOUNDED TILING:**

**INSTANCE:** a finite set W of tiles (squares with colors given on their edges) , and an N by N square V with a given coloring on the 4N unit edge segments on the border of V .

**QUESTION:** is it possible to tile the square V with copies of the tiles in W (without rotations or reflections) such that each pair of adjacent tiles have matching colors on their common edge, and such that the tiles adjacent to the border of the square V have colors matching the given coloring of the border on their exterior edges ?

An instance of **BOUNDED TILING** can be solved by a nondeterministic MIMD-RAM which operates as follows: first it creates $N^2$ copies, each covering a single unit square inside V . This requires time O(log(N)). Next in a nondeterministic move each processor guesses the tile to be placed on its square. This is the unique nondeterministic move in the entire program. Subsequently, using the global memory, each processor exchanges its tile with the processors representing its neighbors, and certifies whether the choices match. The result is obtained by a standard fan-in communication of the bits computed in this way. Evaluation of the match requires constant time; the fan-in procedure requires O(log(N)) steps.

### 4.2.2 The LPRAM.

The LPRAM is a model introduced by Savitch [97]. It is a hybrid between the k-PRAM and the MRAM, since it combines the recursive parallellism of the k-PRAM with the powerful arithmetic of the MRAM. Communication is by channels only as in the k-PRAM.

The LPRAM satisfies the same equations as the MIMD-RAM discussed above, and the proofs for these equations are similar to the ones given before.

**PSPACE $\subseteq$ LPRAM-PTIME** : trivial; don't use the vector instructions.

**LPRAM-PTIME $\subseteq$ PSPACE** : by a standard guess and verify method. The same alternating verifier can be used as in section 4.2.1 , but since the vector instructions allow for exponential

growth of values the certification of values should proceed at the bit level rather than at the register level. This is the same trick as is used for the inclusion **MRAM-NPTIME $\subsetneq$ PSPACE** in the papers by Pratt & Stockmeyer [84] and Hartmanis & Simon [36,37].

**LPRAM-NPTIME $\subsetneq$ NEXPTIME** and **LPRAM-NLOGTIME $\subsetneq$ NP** : These inclusions again are obtained by a brute force simulation.

**NP $\subsetneq$ LPRAM-NLOGTIME** and **NEXPTIME $\subsetneq$ LPRAM-NPTIME** : These inclusions are again proved by implementing a nondeterministic LPRAM-algorithm for **BOUNDED TILING** which runs in logarithmic time: the machine first creates a tree of $N^2$ offspring processors which guess a tile for some unit square inside V. These guesses are communicated to the oldest processor by a fan-in procedure, where along the way the exponential growing information is stored using the vector instructions. Next the same information is distributed once more over $N^2$ processors in such a way that each processor obtains a cell with its four neighbors. These $N^2$ processors then check whether the tiles match and finally the results are communicated to the oldest processor by a bounded fan-in procedure.

### 4.2.3 Extending the SIMDAG with powerful arithmetic.

In this section we consider a hybrid of a SIMDAG with the MRAM. To my knowledge such a model has never been formally introduced in the literature, but implicitly it seems to have been considered.

Starting point here is a more refined analysis of the running time of the transitive closure algorithm in section 3.3.1. Such an analysis 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 unravelling 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 explains why the power of the arithmetics in the model is a crucial factor in establishing whether the model is a second machine device 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 more powerful parallel Boolean instructions the unravelling can be distributed 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 $\subsetneq$ PSIMDAG-PTIME.** Hence it is unlikely that such a model obeys the Parallel Computation Thesis, unless **PSPACE = NEXPTIME.**

The above ideas form the basis for the objections against the Parallel Computation Thesis as put forward by N. Blum [6]. However, in his paper he only considers the third phase of the above algorithm: the computation of the transitive closure itself in $\log(T)$ steps. It should be clear that in a model where addition is the only arithmetic available stages 1 and 2 still will require time $O(S)$.

But given a suitable set of powerful arithmetic instructions the required speed-up of stages 1 and 2 from O(S) to O(log(S)) become possible. It seems therefore that his model requires such strong arithmetic instructions, but this makes his claims much weaker, given the fact that models showing this kind of behavior had been proposed before by Fortune & Wyllie and by Savitch.

Taking the computation of the transition matrix for granted, the first inclusion claimed by Blum: **NEXPTIME** $\subseteq$ **WRAM-PTIME** , follows from the fact that in the transitive closure algorithm only 0's and 1's need to be written, and the writing of a 0 can be suppressed. So in the case of a multiple write the writes will be consistent.

The second inclusion: **DEXPTIME** $\subseteq$ **PRAM-PTIME** , now follows by realizing that instead of the transitive closure the t-th power of the transition matrix can be computed. If the machine is deterministic all powers of the transition matrix have the property that every row contains at most a single non-zero entry. From this it follows that during a squaring every matrix element M(x,M,S)[i,j] will be written by at most one processor. Note that also stage 2 of the algorithm must be modified in order to have the required fan-in of O(S) bits of local information into a single matrix element: this can be obtained at the price of an additional time of O(log(S)). So Blum's construction is valid on the PRAM for deterministic computations.

### 4.3. Arbitrary computations in constant time.

Intuition in computation theory states that, whatever model of a computing device one selects, it should always remain true that running time on this model behaves like a complexity measure. Hence in particular it should never be possible to perform arbitrary complex computations in constant time.

A model which violates this very mild restriction has been described in the literature. We find the model in the context of an analysis of the correct interpretation of the Parallel Computation Thesis by Ian Parberry. These ideas, which were originally presented in his Ph.D. Thesis [73], have been published recently in SIGACT News [74] ; for similar results with slightly improved resource bounds see [75].

In his discussion Parberry considers a parallel RAM model which runs fully synchronously. The number of active processors is determined at the start of the computation. Each processor has beside the standard additive arithmetic some shifts or multiplication like instructions which enable processors to extract bits from bit-strings at arbitrary locations. The processors communicate via a global memory. Each processor has a read-only register initialized at its index.

Resources considered are time T, storage S (counted in RAM words), word-size W (both in local and global storage), and number of processors P . These quantities depend on the size of the input n : T(n), S(n), W(n) and P(n) .

The model resembles the SIMDAG with two major differences: in the first place, rather than having all processors being activated by a single central processor, all processors with index $\leq$ P(n) are activated at time t=0 and remain so till all have halted. The result of the computation can be found by inspecting register 0 in global memory. The second difference is that the model is non-uniform: some quantities like P(n) and derived quantities are given to all processors in advance.

Having done so we have given the machine sufficient power to perform the transitive closure algorithm from section 3.3.1 in constant time. Remember the three contributions of the running time analysis introduced in section 4.2.3 above; here $T'(n)$ and $S'(n)$ denote the time and space bounds on the Turing Machine computations on input $x$ with $|x| = n$, and $K(n)$, the size of the transition matrix $M(x,M,S'(n))$ equals $2^{c'.S'(n)}$ for a suitable constant $c'$.

1. $O(\log(K(n))) = O(S'(n))$ for evaluation of the matrix size $K(n)$
2. $O(S'(n))$ for unravelling the configurations and computing $M(x,M,S'(n))$
3. $O(\log(T'(n))) = O(S'(n))$ for computing the transitive closure of $M(x,M,S'(n))$

Step 1 becomes $O(1)$ due to the non-uniformity involved. Quantities like the space $S'(n)$ and time $T'(n)$ for the Turing Machine to be simulated are given in advance, as are the number of processors $P(n)$ which will be invoked.

Step 2 and step 3 are combined. In the simulation by Blum a team of $O(S'(n))$ processors is called upon in order to decompose the index of processor $i.K(n) + j$ into bits and pieces in order to determinate whether $M(x,M,S'(n))[i,j]$ equals 0 or 1. So the index of a processor is analyzed whether it represents a legal transition between two successive transitions of the machine which is simulated. In Parberry's simulation this idea is carried to the extreme: the index of a processor is analyzed whether it represents the coding of an entire accepting computation. The length of a string coding of such a computation becomes $O(S'(n).T'(n))$, hence the total number of indices analyzed in this way becomes $2^{c.T'(n).S'(n)}$ for some constant $c$. The analysis of one such index can be performed by breaking it into bits and pieces: given a coding like the one used in time/space diagrams of Turing Machines, a team of $T'(n).S'(n)$ processors can perform the comparisons of the coded configurations at time $t$ and time $t+1$ in time $O(1)$; as Parberry indicates $T$ processors in a single team already suffice. One must check furthermore that the initial configutration at time $t=0$ represents a proper configuration on the real input but that can be done in a similar way.

The result obtained in this way is that the simulation time becomes $O(1)$ at the price of using $P(n) = 2^{c.T'(n).S'(n)}.T'(n)$ processors and word-size $W(n) = O(T'(n).S'(n))$.

By a similar proof Parberry shows that if you can simulate with word-size $W(n)$ an $B(n)$ time bounded deterministic Turing Machine in time $B'(n)$ on his model one can use this simulation as a tool for speeding-up a $T'(n)$ time-bounded deterministic Turing Machine computation with word-size $O(W(n) + B(n) + \log(T(n)))$ to time $O(T(n)/B(n) + B'(n))$; the second term in the time analysis of the simulation is the time required to set up a transition matrix structure for computing $B(n)$ steps in time $O(1)$ by table look-up, whereas the first term measures the time needed for simulation of $T(n)$ steps in blocks of $B(n)$ steps each.

This brings Parberry to his analysis of how the above simulations support in fact the Parallel Computation Thesis: a "reasonable" parallel device should satisfy the constraint that $W(n)$ is polynomially bounded by $T(n)$ and that (as a consequence) $P(n)$ is bounded by some expression

$\exp( T(n)^k )$ for some constant $k$. The result above shows that within these bounds arbitrarily large polynomial speed-ups can be obtained with "reasonable" devices, but the speed-up to constant time is possible only at the price of violating this "reasonability" criterion.

To my opinion Parberry's result shows once more the hideous power of shifts and multiplicative instructions. Also the non-uniformity of the model makes it suspect. If quantities like $T'(n)$, $S'(n)$ or $P(n)$ must be computed from the input the $O(1)$ time bound is invalidated.

## REFERENCES.


1 Aho, A.V., Hopcroft, J.E. & Ullman, J.D., *The design and analysis of computer algorithms*, Addison Wesley, 1974.

2 Andreev, A.E., *On a method for obtaining lower bounds for the complexity of individual monotone functions* , Soviet Math. Dokl. **31** (1985) 530-534.

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

4 Blum, M., *A machine-independent theory of the complexity of recursive functions* , J. Assoc. Comput. Mach. **14** (1967) 322-336.

5 Blum, N., *A Boolean function requiring 3n network size* , Theor. Comp. Sci. **28** (1984) 337-345.

6 Blum, N., *A note on the "Parallel Computation Thesis"* , Inf. Proc. Letters **17** (1983) 203-205.

7 Böhling, K.H. & von Braunmühl, B., *Komplexität bei Turingmaschinen* , B.I. Wissenschaftsverlag, Reihe Informatik **14**, 1974.

8 Borodin, A., *On relating time and space to size and depth* , SIAM J. Comput. **6** (1977) 733-744.

9 Chandra, A.K., Kozen, D.C. & Stockmeyer, L.J., *Alternation,* J. Assoc. Comput. Mach. **28** (1981) 114-133.

10 Chazelle, B. & L. Monier, *Unbounded hardware is equivalent to deterministic Turing machines* , Theor. Comp. Sci. **24** (1983) 123-120.

11 Cook, S.A., *The complexity of theorem proving procedures* , Proc. ACM STOC **3** (1971), 151-158.

12 Cook, S.A., *Towards a complexity theory of synchronous parallel computation* , Enseign. Math. **27** (1980) 99-124.

13 Cook, S.A., *The classification of problems which have fast parallel algorithms* , **in** M. Karpinski ed., Proc. Fundamentals of Computation Theory 1983, Borgholm Sweden , Springer LCS **158** (1983), 78-93 .

14 Cook, S.A. & Reckhow, R.A., *Time bounded random access machines,* J. Comput. Syst. Sci. **7** (1973) 354-375.

15 Davis, M., *Computability and unsolvability,* Mc Graw Hill, 1958.

16 Davis, M., *Why Gödel didn't have Church's thesis,* Inf. and Control **54** (1982) 3-24.

17 Dymond, P.W. & Cook, S.A., *Hardware complexity and parallel computation* , Proc. IEEE FOCS **21** (1980), 360-372.

18 Dymond, P.W. & Ruzzo, W.L., *Parallel RAMs with owned global memory and deterministic context-free language recognition* , Proc. ICALP **13** , Springer LCS **226** (1986), 95-104.

19 Even, S., & R.E. Tarjan, *A combinatorial problem which is complete in polynomial space* , J. Assoc. Comput. Mach. **23** (1976) 710-719.

20 Fich, F.E., P.L. Radge & A. Widgerson, *Relations between concurrent-write models of parallel computation* , SIAM J. Comput. **17** (1988) 606-627.

21 Fich, F.E., F. Meyer auf der Heide, P.L. Radge & A. Widgerson, *One, Two, Three ....Infinity: Lower bounds for parallel computation* , Proc. ACM STOC **17** (1985), 48-58.

22 Fischer, P.C., Meyer, A.R. & Rosenberg, A.L., *Real-time simulation of multihead tape units*, J. Assoc. Comput. Mach. **19** (1972) 590-607.

23 Fortune, S. & J. Wyllie, *Parallellism in random access machines*, Proc. ACM STOC **10** (1978), 114-118.

24 Fraenkel, A.S. , M.R. Garey, D.S. Johnson, T. Schaefer & Y. Yesha, *The complexity of Checkers on an $N \times N$ board, preliminary report* , Proc. IEEE FOCS **19** (1978), 55-64.

25 Fraenkel, A.S. & D. Lichtenstein, *Computing a perfect strategy for $n$ by $n$ Chess requires time exponential in $n$* , J. Combin. Theory **31** (1981) 199-214.

26 Fredman, F.L., Komlós, J. & Szemerédi E., *Storing a sparse table with $O(1)$ worst case access time,* Proc. IEEE FOCS **23** (1982), 165-169.

27 Fredman, F.L., Komlós, J. & Szemerédi E., *Storing a sparse table with $O(1)$ worst case access time,* J. Assoc. Comput. Mach. **31** (1984) 538-544.

28 Galil, Z. & Paul, W., *An efficient general-purpose parallel computer* , J. Assoc. Comput. Mach. **30** (1983) 360-387.

29 Garey, M.S. & D.S. Johnson, *Computers and intractability, a guide to the theory of NP-completeness* , Freeman, 1979.

30 Gilbert, J.R., T. Lengauer & R.E. Tarjan, *The pebbling problem is complete in polynomial space* , SIAM J. Comput. **9** (1980) 513-524.

31 Goldschlager, L.M., *A universal interconnection pattern for parallel computers,* J. Assoc. Comput. Mach. **29** (1982) 1073-1086.

32 Gruska, J., *Systolic Automata - Power, Characterizations, Nonhomogeneity*, Proc. MFCS'84, Springer LCS **176** (1984), 32-49.

33 Halpern, J.Y., Loui, M.C., Meyer, A.R. & Weise, D., *On Time Versus Space III* , Math. Syst. Theory **19** (1986) 13-28.

34 Harel, D., *Recurring dominos: Making the highly undecidable highly understandable* , Ann. Disc. Math. **24** (1985) 51-72.

35 Hartmanis, J., *Observations about the development of theoretical computer science,* Ann. Hist. Comp. **3** (1981) 42-51.

36 Hartmanis, J. & J. Simon, *On the power of multiplication in random access machines* , Proc IEEE SWAT **15** (1974), 13-23.

37 Hartmanis, J. & Simon, J., *On the structure of feasible computations,* **in** Rubinoff, M. & Yovits, M.C. eds., *Advances in computers* **14** , Acad. Press 1976, pp. 1-43.

38 Hartmanis, J. & Stearns, R.E., *On the computational complexity of algorithms* , Trans. Amer. Math. Soc. **117** (1966) 285-306.

39 Hemmerling, A., *On the space complexity of multidimensional Turing automata,* E.-M. Arndt Univ. Greifswald, Preprint **2** , 1979.

40 Hennie, F.C. & Stearns, R.E., *Two-way simulation of multi-tape Turing machines* , J. Assoc. Comput. Mach. **13** (1966) 533-546.

41 Hopcroft, J., Paul, W. & Valiant, L., *On time versus space and related problems* , Proc. IEEE FOCS **16** (1975), 57-64.

42 Hopcroft, J., Paul, W. & Valiant, L., *On time versus space* , J. Assoc. Comput. Mach. **24** (1977) 332-337.

43 Hopcroft, J.E. & J.D. Ullman, *Introduction to automata theory, languages and computation* , Addison Wesley 1979 .

44 Ibarra, O.H., & Moran, S., *Some time-space trade-off results concerning single-tape and offline TM's* , SIAM J. Comput. **12** (1983) 388-394.

45 Jacobs, C.T.M. & van Emde Boas, P., *Two results on tables,* Inf. Proc. Letters **22** (1986) 43-48.

46 Jerrum, M., *The complexity of finding minimum-length generator sequences (extended abstract),* Proc. ICALP **11**, Springer LCS **172** (1984), 270-280.

47 Jerrum, M., *The complexity of finding minimum-length generator sequences* , Theor. Comp. Sci. **36** (1985) 265-290.

48 Johnson, D.S., *A child's garden of complexity classes,* This volume.

49 Karp, R.M. & Lipton, R.J., *Some connections between nonuniform and uniform complexity classes* , Proc. ACM STOC **12** (1980), 302-309.

50 Katajainen, J., Penttonen, M. & van Leeuwen, J., *Fast simulation of Turing machines by random access machines,* SIAM J. Comput. **17** (1988), 77-88.

51 Kleene, S.C., *Origins of recursive function theory,* Ann. Hist. Comp. **3** (1981) 52-67.

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

53 Kosaraju, S.R., *Real-time simulation of concatenable double-ended queues* , Proc. ACM STOC **11** (1979), 346-351.

54 Krapchenko, V.M., *Complexity of the realization of a linear function in the class of Π-circuits* , Math. Notes Acad. Sci. USSR (1971) 21-23.

55 Leong, B.L., & Seiferas, J.I., *New real-time simulations of multihead tape units* , J. Assoc. Comput. Mach. **28** (1981) 166-180.

56 Lewis, H.R. & Papadimitriou, C.H., *Elements of the theory of computation* , Prentice Hall, 1981.

57 Li, M., Longpré, L. & Vitányi, P.M.B., *The power of the queue* , Proc. Structure in Compl. Theory **1** (1986), Springer LCS **223** (1986), 219-233.

58 Li, M. & Vitányi, P.M.B., *Tape versus Queue and Stacs: the Lower Bounds*, Inf. and Comp. **78** (1988) 56-85.

59 Li, M. & Yesha, Y., *Separation and lower bounds for ROM and Nondeterministic Models of Computation*, Inf. and Comp. **73** (1987), 102-128.

60 Lichtenstein, D. & M. Sipser, *Go is polynomial-space hard* , J. Assoc. Comput. Mach. **27** (1980) 393-401.

61 Lorys, K. & Liskiewicz, M., *Two applications of Fürer's counter to one-tape nondeteriministic TMs*, proc. MFCS'88, Springer LCS, to appear 1988.

62 Lupanov, O.B., *On the asymptotic bounds of complexities of formulas which realize logic algebra functions*, Dokl. Akad. Nauk SSSR **128** (1959) 464-467; Engl. transl. in Autom. Expr. **2 nr. 6** (1960) 12-14.

63 Maass, W., *Combinatorial lower bound arguments for deterministic and nondeterministic Turing machines* , Trans. Amer. Math. Soc. **292** (1985) 675-693.

64 Maass, W. & Schnitger, G., *An optimal lower bound for Turing machines with one work tape and a two-way input tape* , Proc. Structure in Compl. Theory **1**, Springer LCS **223** (1986), 249-264.

65 Machtey, M. & P. Young, *An Introduction to the general theory of algorithms*, Theory of Computation Series, North Holland, 1978.

66 McColl, W.F. & Patterson, M.S., *The depth of all Boolean functions* , SIAM J. Comput **6** (1977) 373-380.

67 Mead, C. & Conway, L., *Introduction to VLSI systems*, Addison Wesley, 1980.

68 Mehlhorn, K., *On the program size of perfect universal hash functions,* Proc. IEEE FOCS **23** (1982), 170-175

69 Mehlhorn, K., *Data structures and algorithms 1, Sorting and searching,* EATCS Monographs on Theoretical Computer Science **1**, Springer Verlag, 1984.

70 Meyer auf der Heide, F., *Efficiency of universal parallel computers* , Acta Informatica **19** (1983) 269-296.

71 Minsky, M., *Computation, finite and infinite machines,* Prentice Hall, 1972.

72 Neciporuk, È.I., *A Boolean function* , Soviet Math. Dokl. **7** (1966) 999-1000.

73 Parberry, I., *A complexity theory for parallel computation* , Ph.D. Thesis, Dept. Comp. Sci., Univ. of Warwick, May 1984.

74 Parberry, I., *Parallel speedup of sequential machines: a defense of the Parallel Computation Thesis* , SIGACT News **18.1** (1986) 54-67.

75 Parberry, I. & Schnitger, G., *Parallel computation with treshold functions (preliminary version)* , Proc. Structure in Complexity Theory **1** , Springer LCS **223** (1986), 272-290.

76 Paterson, M.S., *Tape bounds for time bounded Turing machines* , J. Comput. Syst. Sci. **6** (1972) 116-124.

77 Paul, W.J., *On-line simulation of* k+1 *tapes by* k *tapes requires nonlinear time* , Inf. & Control **53** (1982) 1-8.

78 Paul, W.J., Pippenger, N., Szemerédi, E & Trotter, W.T., *On determinism versus non-determinism and related problems* , Proc. IEEE FOCS **24** (1983), 429-438.

79 Paul, W.J. & Reischuk, R., *On time versus space II* , J. Comput. Syst. Sci. **23** (1981) 108-126.

80 Pippenger, N., *Probabilistic simulations* , Proc. ACM STOC **14** (1982), 17-26.

81 Pippenger, N., *On simultaneous resource bounds*, Proc. IEEE FOCS **20** (1979), 307-311.

82 Pippenger, N. & Fischer, M.J., *Relations among complexity measures* , J. Assoc. Comput. Mach. **26** (1979) 361-381.

83 Pratt, V.R., *The effect of basis on size of Boolean expressions* , Proc. IEEE FOCS **16** (1975), 119-121; Also: Kibern. Sb. Nov. Ser. **17** (1980) 114-123 (Russian).

84 Pratt, V.R. & Stockmeyer, L.J., *A characterization of the power of vector machines,* J. Comput. Syst. Sci. **12** (1976) 198-221.

85 Razborov, A.A., *Lower bounds for the monotone complexity of some Boolean functions* , Soviet Mat. Dokl. **31** (1985) 354-357.

86 Reif, J.H., *Complexity of the mover's problem and generalizations , extended abstract* , Proc. IEEE FOCS **20** (1979), 421-427.

87 Reisch, S., *HEX ist **PSPACE**-volständig* , Acta Informatica **15** (1981) 167-191.

88 Robson, J.M., *N by N checkers is exptime complete*, SIAM J. Comput. **13** (1984) 252-267.

89 Robson, J.M., *The complexity of Go*, **in** R.E.A. Mason, ed., Information Processing 83 (proceedings ninth IFIP world computer congress, Paris 1983), North Holland, Amsterdam (1983), pp. 413-418.

90 Ruzzo, W.L., *An improved characterization of the power of vector machines* , Rep. U. Washington DCS-TR-78-10-01.

91 Ruzzo, W.L., *Tree-size bounded alternation* , J. Comput. Syst. Sci. **21** (1980) 218-235.

92 Savage, J.E., *The complexity of computing* , John Wiley , 1976.

93 Savelsberg, M. & van Emde Boas, P., *BOUNDED TILING, an alternative to SATISFIABILITY?* , **in** Wechsung, G., (ed.), Proc. 2nd Frege Conference, Akademie Verlag, Berlin GDR, (1984), pp. 354-365.

94 Savitch, W.J., *Relations between deterministic and nondeterministic tape complexities,*. J. Comput. Syst. Sci. **4** (1970) 177-192.

95 Savitch, W.J., *The influence of the machine model on computational complexity,* **in** Lenstra, J.K., Rinnooy Kan, A.H.G. & van Emde Boas, P. Interfaces between computer science and operations research, Math. Centre Tracts **99** (1978), pp. 1-32.

96 Savitch, W.J., *Recursive Turing machines* , Inter. J. Comput. Math. **6** (1977) 3-31.

97 Savitch, W.J., *Parallel random access machines with powerful instruction sets* , Math. Systems Theory **15** (1982) 191-210.

98 Savitch, W.J., & Stimson, M.J., *Time bounded random access machines with parallel processing,* J. Assoc. Comput. Mach. **26** (1979) 103-118.

99 Schaefer, T.J., *Complexity of some two-person perfect-information games* , J. Comput. Systems Sci. **16** (1978) 185-225.

100 Schmidt, J.P. & Siegel, A., *The spatial complexity of oblivious k-probe hash functions*, manuscript Courant Institute, New York, May 1988.

101 Schnitger, G., *Storage Modification Machines versus Kolmogorov - Uspenskii Machines (an information flow analysis )*, manuscript Penn State University, Oct . 1987.

102 Schnorr, C.P., *The network complexity and the Turing machine complexity of finite functions*, Acta Informatica **7** (1976) 95-107.

103 Schönhage, A., *On the power of random access machines* , Proc. ICALP **6** , Springer LCS **71** (1979), 520-529.

104 Schönhage, A., *Storage modification machines* , SIAM J. Comput. **9** (1980) 490-508.

105 Schönhage, A., *Tapes versus pointers, a study in implementing fast algorithms* , EATCS Bulletin, **30** (Oct. 1986) 23-32.

106 Schorr, A., *Physical parallel devices are not much faster than sequential ones* , Inf. Proc. Letters **17** (1983) 103-106.

107 Sipser, N., *Halting space-bounded computations,* note. Theor. Comp. Sci. **10** (1980) 335-337.

108 Slot, C. & van Emde Boas, P., *On tape versus core; an application of space efficient hash functions to the invariance of space,* Proc. ACM STOC **16** (1984), 391-400.

109 Slot, C. & van Emde Boas, P., *The problem of space invariance for sequential machines* , Inf. and Comp. **77** (1988) 93-122.

110 Stegwee, R.A., Torenvliet, L. & van Emde Boas, P., *The power of your editor,* Rep. IBM Research, RJ 4711 (50179) (May 1985) .

111 Stockmeyer, *The complexity of decision problems in automata theory and logic* , rep. MAC TR-133, MIT 1984.

112 Stockmeyer, L., *The polynomial time hierarchy*, Theor. Comp. Sci. **3** (1977) 1-22.

113 Stockmeyer, L., *Classifying the computational complexity of problems,* J. Symb. Logic **52** (1987) 1-43.

114 Stoss, H.J., *Zwei-Band-Simulation von Turingmaschinen* , Computing **7** (1971) 222-235.

115 Toffoli, T. & Margolus, N., *Cellular Automata Machines, a new environment for modelling* , the MIT Press, 1987.

116 Torenvliet, L. & P. van Emde Boas, *A Note on Time and Space* , Proc. Computing Science in the Netherlands CSN87 (1987), pp.225-234.

117 van Emde Boas, P. *The second machine class, models of parallelism,* **in** Lenstra, J.K. & van Leeuwen, J., eds., Parallel computers and computations, CWI syllabus **9** (1985), pp. 133-161.

118 van Emde Boas, P. *The second machine class 2, an encyclopedic view on the parallel computation thesis,* **in** H. Rasiowa, ed., Mathematical problems in computation theory, Banach Center Publications, **21** (1987), pp. 235-256.

119 P. van Emde Boas, *Space Measures for Storage Modification Machines* , Rep. FVI-UvA-87-16, Nov 1987 (to appear in Inf. Proc. Letters).

120 van Leeuwen, J. & J. Wiedermann, *Array processing machines* , **in** L. Budach ed., Fundamentals of Computation Theory 1985, Cottbus GDR, Springer LCS **199** (1985), 257-268. (A more extended version is available as Rep. Rijksuniversiteit Utrecht, RUU-CS-84-13, Dec. 1984.)

121 Vitányi, P.M.B., *An optimal simulation of counter machines* , SIAM J. Comput. **14** (1985) 1-33.

122 Vitányi, P.M.B., *Non-sequential computation and laws of nature* , Proc. AWOC 86, Springer LCS **227** (1986), 108-120 .

123 Wagner, K. & Wechsung, G., *Computational complexity,* Mathematische Monographien **19** , VEB Deutscher Verlag der Wissenschaften , 1986. (Published in the West by Reidel 1986).

124 Wiedermann, J. *Deterministic and nondeterministic simulation of the RAM by the Turing Machine,* **in** R.E.A. Mason, ed., Information Processing 83 (proceedings ninth IFIP world computer congress, Paris), North Holland, Amsterdam (1983), pp. 163-168.

125 Wiedermann, J., *Parallel Turing machines* , report Rijksuniversiteit Utrecht, RUU-CS-84-11, Nov. 1984.

126 Wiedermann, J., *Fast simulation of nondeterministic Turing machines with application to the knapsack problem* , Rep. VUSEI-AR 4/1986, Bratsilava, Oct. 1986.