

# Parallelism and Machine Models

Andrew D Smith

University of New Brunswick, Fredericton  
Faculty of Computer Science

# Overview

**Part 1:** The Parallel Computation Thesis

**Part 2:** Parallelism of Arithmetic RAMs

# Part 1

# The Parallel Computation Thesis

# The Random Access Machine (RAM)

## Cook & Reckhow (1974)

More realistic model of existing computers

Loses the sequential access of Turing machines

Keeps certain properties important to complexity theory

Memory consists of an infinite sequence of registers  
and each register is capable of holding an arbitrary integer

Associated with the machine is a cost function  
which assigns a cost (time required) to each operation

# The Random Access Machine (RAM)

**Cook & Reckhow (1974)**

More realistic model of existing computers

Loses the sequential access of Turing machines

Keeps certain properties important to complexity theory

Memory consists of an infinite sequence of registers  
and each register is capable of holding an arbitrary integer

Associated with the machine is a cost function  
which assigns a cost (time required) to each operation

Unless otherwise stated  
all operations are assumed uniform cost

## The original instruction set

$$X_i \leftarrow c, c \in \mathbb{Z}$$

Literal assignment (always unit cost)

$$X_i \leftarrow X_{X_j}$$

$$X_{X_i} \leftarrow X_j$$

Indirect addressing (central to Random Access)

IF  $X_i > 0$  GOTO  $m$

Conditional transfer (required for Turing equivalence)

READ  $X_i$

WRITE  $X_i$

Input and Output operations

$$X_i \leftarrow X_j + X_k$$

$$X_i \leftarrow X_j - X_k$$

Addition and Subtraction

## Benefits of Random Access

- Random Access allows indirect addressing, essential for abstract data types such as list and trees
- The RAM is also a cognitive aid, and permits an algorithm designer to visualize the manipulation of abstract structures

## Relating Turing Machines and RAMs

If some **Turing machine** recognizes  $A$  within time  $T(n) \geq n$ ,  
then some **RAM** (with arbitrary cost  $\ell$ ), recognizes  $A$  in time  $T(n)\ell(n)$

If a set  $A$  is recognized by a **logarithmic cost RAM** within time  $T(n) > n$ ,  
then some multitape Turing Machine recognizes  $A$  within time  $T(n)^2$

If a set  $A$  is recognized by **uniform cost RAM** within time  $T(n) > n$ ,  
then some multitape Turing Machine recognizes  $A$  within time  $T(n)^3$

In a nutshell: RAMs are safe to use in computational complexity

# The Parallel Random Access Machine

## Fortune & Wyllie (1978)

Introduced to model computation by multiple RAMs operating simultaneously on the same data with the goal of solving the same problem

A PRAM has an unbounded set of processors  $P_0, P_1, \dots$

- Each is a RAM as defined by Cook & Reckhow

Slight Modification:

$P_j$  has an accumulator  $A_j$   
all ops except new STORE  $X_i$   
operate on accumulators

A PRAM has an unbounded global memory  $X_0, X_1, \dots$

- Each  $P_j$  has a local memory also (details not important)

## Start of computation on a PRAM

- Input placed in input registers; all memory cleared
- Input length placed in  $A_0$  and  $P_0$  is started

## Start of computation on a PRAM

- Input placed in input registers; all memory cleared
- Input length placed in  $A_0$  and  $P_0$  is started

### If $P_i$ executes **FORK**( $x$ ), then:

1. Local memory for first inactive processor  $P_j$  is cleared
2. Accumulator of  $P_j$  is given value in the accumulator of  $P_i$
3.  $P_j$  starts running at label  $x$  of the finite program

Another new instruction

## Start of computation on a PRAM

- Input placed in input registers; all memory cleared
- Input length placed in  $A_0$  and  $P_0$  is started

**If  $P_i$  executes FORK( $x$ ), then:**

1. Local memory for first inactive processor  $P_j$  is cleared
2. Accumulator of  $P_j$  is given value in the accumulator of  $P_i$
3.  $P_j$  starts running at label  $x$  of the finite program

Another new instruction

## Number of Processors

PRAM algorithms often assume some number of processors  
True PRAM model only requires processors be countable  
Their number restricted by what can be started

## Start of computation on a PRAM

- Input placed in input registers; all memory cleared
- Input length placed in  $A_0$  and  $P_0$  is started

If  $P_i$  executes **FORK**( $x$ ), then:

1. Local memory for first inactive processor  $P_j$  is cleared
2. Accumulator of  $P_j$  is given value in the accumulator of  $P_i$
3.  $P_j$  starts running at label  $x$  of the finite program

Another new instruction

## Number of Processors

PRAM algorithms often assume some number of processors  
True PRAM model only requires processors be countable  
Their number restricted by what can be started

## Conflicts

Concurrent reads allowed

Concurrent read/write  $\rightarrow$  write goes first

Concurrent write causes immediate halt in rejecting state

{CREW, EREW, CREW}  
Irrelevant

## Start of computation on a PRAM

- Input placed in input registers; all memory cleared
- Input length placed in  $A_0$  and  $P_0$  is started

If  $P_i$  executes **FORK**( $x$ ), then:

1. Local memory for first inactive processor  $P_j$  is cleared
2. Accumulator of  $P_j$  is given value in the accumulator of  $P_i$
3.  $P_j$  starts running at label  $x$  of the finite program

Another new instruction

## Number of Processors

PRAM algorithms often assume some number of processors  
True PRAM model only requires processors be countable  
Their number restricted by what can be started

## Conflicts

Concurrent reads allowed

Concurrent read/write  $\rightarrow$  write goes first

Concurrent write causes immediate halt in rejecting state

{CREW, EREW, CREW}  
Irrelevant

No consideration of implementation details (e.g. communication cost)

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) = \bigcup_k \text{TM-SPACE}(T^k(n))$$

The most fundamental result in parallel complexity theory  
Connects parallel complexity to the world of  $P \stackrel{?}{=} NP$

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) = \bigcup_k \text{TM-SPACE}(T^k(n))$$

The most fundamental result in parallel complexity theory  
Connects parallel complexity to the world of  $P \stackrel{?}{=} NP$

## Two Consequences

### PRAM-LOGTIME = LOGSPACE

- Related to AC and NC hierarchies
- Well known result

### PRAM-PTIME = PSPACE

- Write conflict irrelevant mod P
- **Of Interest Here**

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) = \bigcup_k \text{TM-SPACE}(T^k(n))$$

The most fundamental result in parallel complexity theory  
Connects parallel complexity to the world of  $P \stackrel{?}{=} NP$

## Two Consequences

### PRAM-LOGTIME = LOGSPACE

- Related to AC and NC hierarchies
- Well known result

### PRAM-PTIME = PSPACE

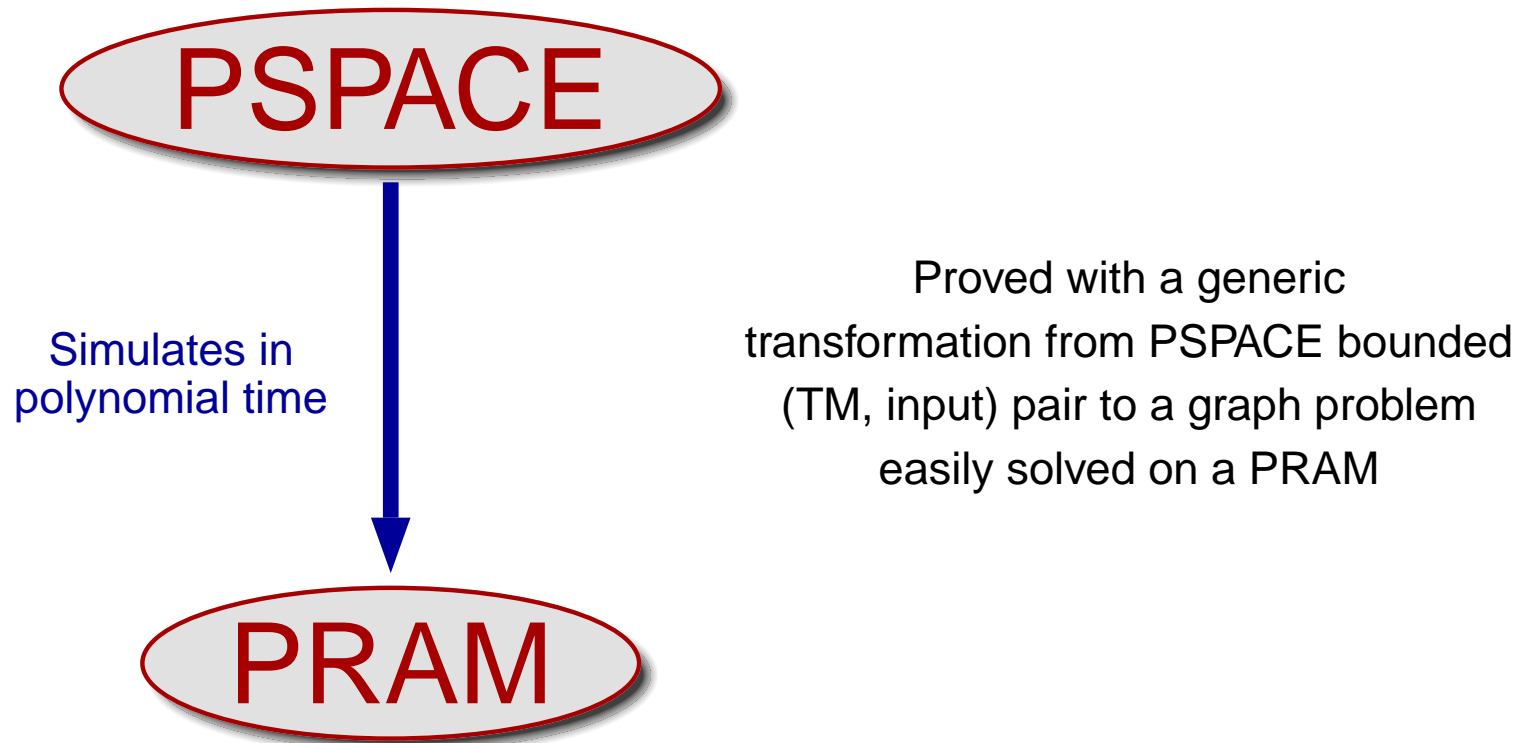
- Write conflict irrelevant mod P
- **Of Interest Here**

## Some Remarks

The result seems to rely on communicating exponential amounts of information through global storage. Two that don't:

The class of sets accepted by **nondeterministic** polynomial time bounded, polynomial global storage bounded PRAMs is identically **PSPACE**

The class of sets accepted by **deterministic** polynomial time, polynomial space PRAMs contains the class **co-NP**



$$\bigcup_k \text{PRAM-TIME}(T^k(n)) \subseteq \bigcup_k \text{TM-SPACE}(T^k(n))$$

How it works: Transform Turing Machine Acceptance into Graph Reachability

$M$  is a Turing Machine,  $x$  is the input,  $S$  is the tape used

Identify **states** of  $M$  with **nodes** of digraph  $G(x, M, S)$

Identify **transitions** of  $M$  with **arcs** of  $G(x, M, S)$

Counting States:

$S \in O(n^c) \Rightarrow M$  has  $2^{O(n^c)}$  states

If  $M$  is deterministic then  $G(x, M, S)$  has out-degree  $\leq 1$   
(a rooted forest, arcs toward roots)

(node  $s \equiv$  initial configuration), (node  $t \equiv$  final configuration)

If  $t$  is reachable from  $s$ , then the TM halts in an accepting state on  $x$

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) \subseteq \bigcup_k \text{TM-SPACE}(T^k(n))$$

How it works: Transform Turing Machine Acceptance into Graph Reachability

$M$  is a Turing Machine,  $x$  is the input,  $S$  is the tape used

Identify **states** of  $M$  with **nodes** of digraph  $G(x, M, S)$

Identify **transitions** of  $M$  with **arcs** of  $G(x, M, S)$

Counting States:

$$S \in O(n^c) \Rightarrow M \text{ has } 2^{O(n^c)} \text{ states}$$

If  $M$  is deterministic then  $G(x, M, S)$  has out-degree  $\leq 1$   
(a rooted forest, arcs toward roots)

(node  $s \equiv$  initial configuration), (node  $t \equiv$  final configuration)

If  $t$  is reachable from  $s$ , then the TM halts in an accepting state on  $x$

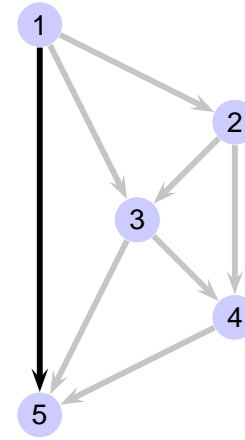
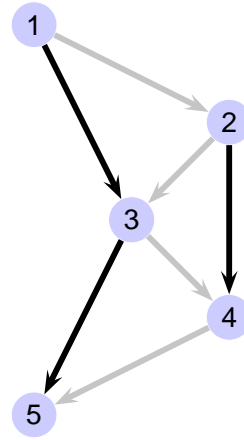
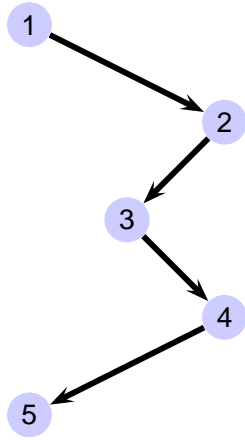
**Transitive Closure: solved with Boolean matrix multiplication of adjacency matrix**

**Boolean Matrix Multiplication  $\in \text{AC}^0$ , Transitive Closure  $\in \text{AC}^1 \subseteq \text{NC}^2$**

**Graph has exponential number of nodes  $\Rightarrow$  NC is really polynomial**

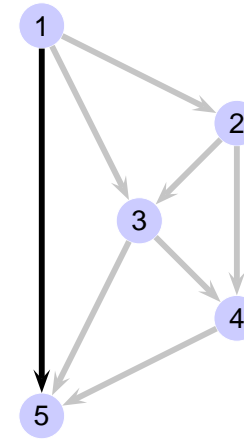
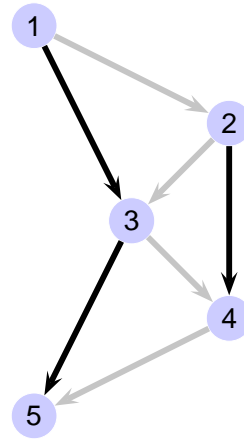
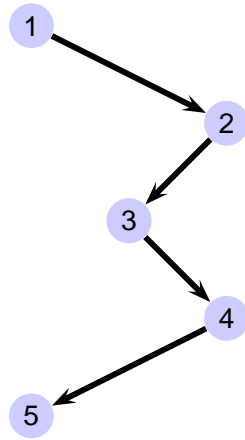
## Transitive Closure

Given a directed graph  $G$  and two nodes  $s$  and  $t$ , determine whether there is a path in  $G$  that starts at  $s$  and ends at  $t$ .



## Transitive Closure

Given a directed graph  $G$  and two nodes  $s$  and  $t$ , determine whether there is a path in  $G$  that starts at  $s$  and ends at  $t$ .



## Boolean Matrix Multiplication

Adjacency Matrix  $(I + A)$

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$(I + A)^2$

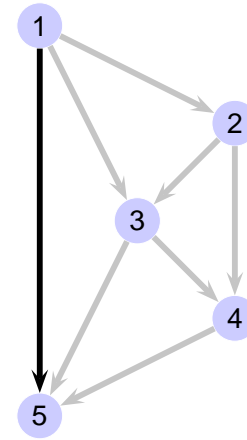
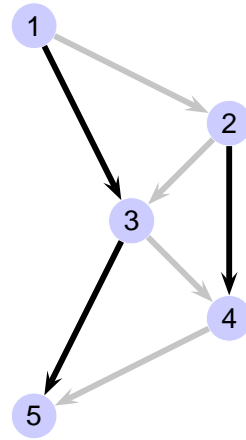
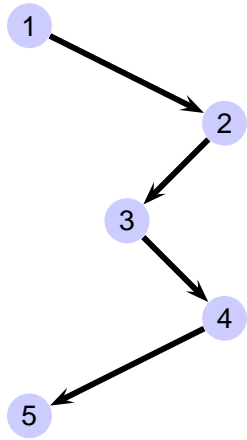
$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$(I + (I + A)^2)^2$

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

## Transitive Closure

Given a directed graph  $G$  and two nodes  $s$  and  $t$ , determine whether there is a path in  $G$  that starts at  $s$  and ends at  $t$ .



## Boolean Matrix Multiplication

Adjacency Matrix  $(I + A)$

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$(I + A)^2$

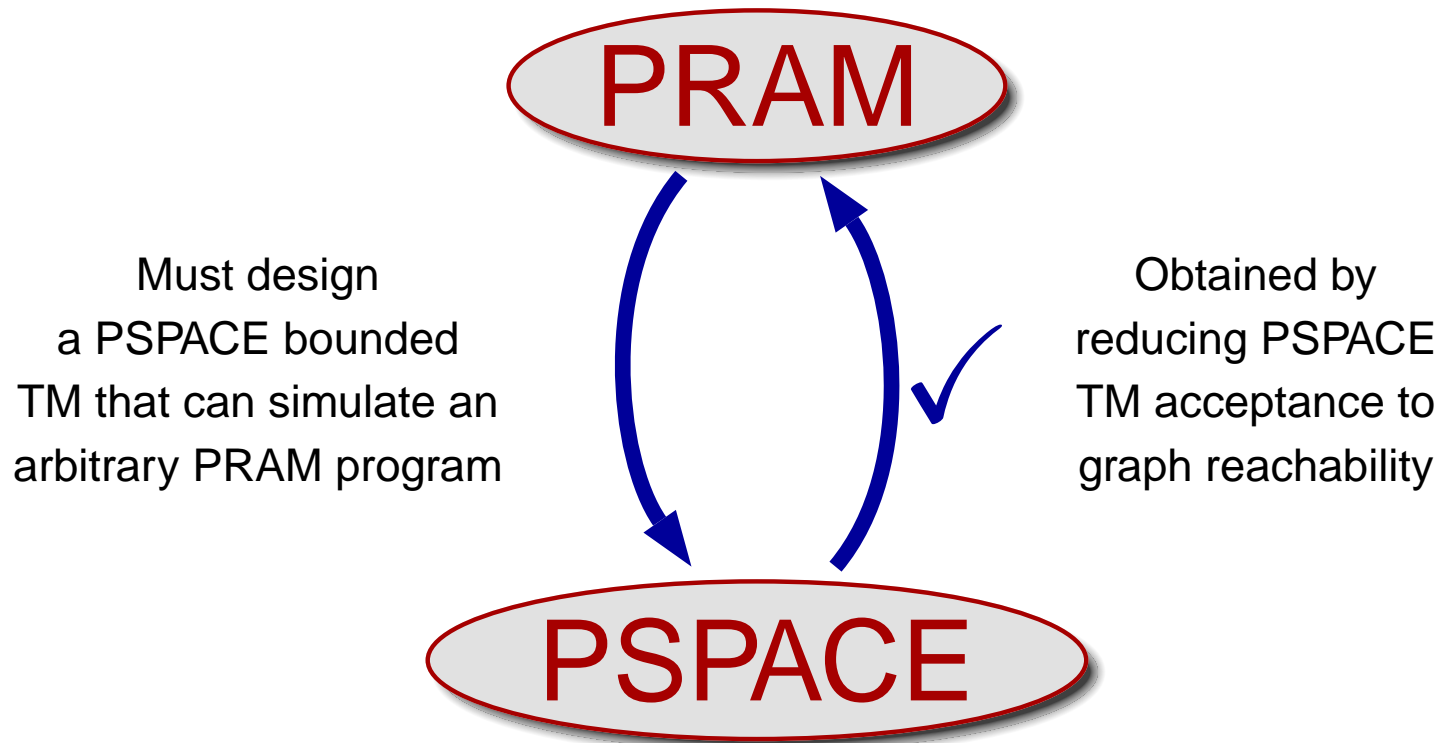
$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$(I + (I + A)^2)^2$

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

**Possibly the first use of pointer jumping!**

Now for the other direction...



$$\bigcup_k \text{PRAM-TIME}(T^k(n)) \supseteq \bigcup_k \text{TM-SPACE}(T^k(n))$$

### Mutually Recursive Procedures

Verify the instruction executed, and the contents of memory and accumulators at each time

**VERIFY\_ACCUMULATOR**( $P_j, t, c$ )

Guess: time  $t' < t$ , instruction  $i$

Verify  $\left\{ \begin{array}{l} A_j \text{ last modified at time } t' \\ P_j \text{ executed } i \text{ at time } t' \\ i \text{ executed at time } t' \text{ produces } c \end{array} \right.$

**VERIFY\_MEMORY**( $X, t, c$ )

Guess: processor  $P_j$ , time  $t' < t$

Verify  $\left\{ \begin{array}{l} P_j \text{ stored to } X \text{ at time } t' \\ A_j \text{ contained } c \text{ at time } t' \\ \text{no store to } X \text{ between } t' \text{ and } t \end{array} \right.$

**VERIFY\_INSTRUCTION**( $P_j, t, i$ )

Guess: processor  $P_j$ , instruction  $i'$

Verify  $\left\{ \begin{array}{l} P_j \text{ executed } i' \text{ at time } t - 1 \\ \text{if } i' \neq i - 1 \text{ then } i' \text{ jumps to } i \end{array} \right.$

**Think:**

Tracing adjacency list represented digraph backward

Must guess origin of each arc!

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) \supseteq \bigcup_k \text{TM-SPACE}(T^k(n))$$

## 1. Construct non-deterministic PSPACE bounded TM:

To determine if the PRAM accepts, the TM must **VERIFY** 2 things:

1.  $P_0$  halts at time  $t$  with  $A_0 = 1$
2. No concurrent writes occurred

**VERIFY** using mutually recursive procedures from last slide:

**{VERIFY\_INSTRUCTION, VERIFY\_ACCUMULATOR, VERIFY\_MEMORY}**

Since the PRAM takes polynomial time, the stack will contain a poly number of calls  
And since the space is at most exponential, values pushed require polynomial bits

$$\bigcup_k \text{PRAM-TIME}(T^k(n)) \supseteq \bigcup_k \text{TM-SPACE}(T^k(n))$$

## 1. Construct non-deterministic PSPACE bounded TM:

To determine if the PRAM accepts, the TM must **VERIFY** 2 things:

1.  $P_0$  halts at time  $t$  with  $A_0 = 1$
2. No concurrent writes occurred

**VERIFY** using mutually recursive procedures from last slide:

{**VERIFY\_INSTRUCTION**, **VERIFY\_ACCUMULATOR**, **VERIFY\_MEMORY**}

Since the PRAM takes polynomial time, the stack will contain a poly number of calls  
And since the space is at most exponential, values pushed require polynomial bits

## 2. Eliminate all non-deterministic choices:

### **Savitch's Theorem (1970)**

For any "space constructible" function  $T(n) \geq \log n$ ,  
 $\text{NSPACE}(T(n)) \subseteq \text{SPACE}(T(n)^2)$

# The Parallel Computation Thesis

**Goldschlager (1982)**

Introduced concept to define parallelism for complexity theory

Time-bounded parallel machines are polynomially related to space-bounded computers. That is, for any function  $T(n)$ ,

$$\bigcup_k \text{PARALLEL-TIME}(T^k(n)) = \bigcup_k \text{SPACE}(T^k(n))$$

# The Parallel Computation Thesis

**Goldschlager (1982)**

Introduced concept to define parallelism for complexity theory

Time-bounded parallel machines are polynomially related to space-bounded computers. That is, for any function  $T(n)$ ,

$$\bigcup_k \text{PARALLEL-TIME}(T^k(n)) = \bigcup_k \text{SPACE}(T^k(n))$$

**In theoretical computer science, a thesis is definitive**

- ⇒⇒ Church-Turing Thesis defines **Effective** computation
- ⇒⇒ Edmonds' (Efficiency) Thesis defines **Efficient** computation
- ⇒⇒ Parallel Computing Thesis defines **Parallel** computation

# The Parallel Computation Thesis

**Goldschlager (1982)**

Introduced concept to define parallelism for complexity theory

Time-bounded parallel machines are polynomially related to space-bounded computers. That is, for any function  $T(n)$ ,

$$\bigcup_k \text{PARALLEL-TIME}(T^k(n)) = \bigcup_k \text{SPACE}(T^k(n))$$

**In theoretical computer science, a thesis is definitive**

- ⇒ Church-Turing Thesis defines **Effective** computation
- ⇒ Edmonds' (Efficiency) Thesis defines **Efficient** computation
- ⇒ Parallel Computing Thesis defines **Parallel** computation

## The Second Machine Class

Consists of those devices that satisfy the Parallel Computation Thesis with respect to the traditional, sequential Turing machine model

**Not required to have more than one processor!**

## **Part 2**

# **Parallelism of Arithmetic RAMs**

## Multiplication is controversial!

$\{\times, \div\}$  are more complex than  $\{+, -\}$

original evidence:  
circuit complexity

$+, - \in AC^0$  but  $\times, \div \notin AC^0$   
(Furst, Saxe & Sipser, 1981)

# Multiplication is controversial!

$\{\times, \div\}$  are more complex than  $\{+, -\}$

original evidence:  
circuit complexity

$+, - \in AC^0$  but  $\times, \div \notin AC^0$   
(Furst, Saxe & Sipser, 1981)

## Hartmanis & Simon (1974)

Proved RAM with  $\{+, -, \times, \div, \wedge, \vee, \neg\}$  is in Second Machine Class

**Open Problem:** relationship between RAMs with and without multiplication

# Multiplication is controversial!

$\{\times, \div\}$  are more complex than  $\{+, -\}$

original evidence:  
circuit complexity

$+, - \in AC^0$  but  $\times, \div \notin AC^0$   
(Furst, Saxe & Sipser, 1981)

## Hartmanis & Simon (1974)

Proved RAM with  $\{+, -, \times, \div, \wedge, \vee, \neg\}$  is in Second Machine Class

**Open Problem:** relationship between RAMs with and without multiplication

## Bertoni, Mauri & Sabadini (1981)

What is the complexity of counting solutions to QBF problems?

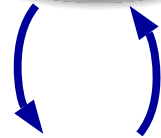
- ➡ Motivated by Counting Complexity of Valiant (#P results)
- ➡ Discovered that generating functions for # of solutions to a QBF can be evaluated by manipulating polynomials in a particular way
- ➡ Proved that Arithmetic RAMs (those with  $+, -, \times, \div$ ) can do the required manipulations in polynomial time

# How multiplication simulates parallelism

(Blue lines indicate P-time reductions)

PRAM

The usual (familiar) model of parallel computation  
A measure of data independence



PSPACE

Classical complexity class  
Equal to PTIME on a PRAM (as we have seen)

Part 1

Part 2



QBF

Satisfiability of Quantified Boolean Formulas  
Most well known PSPACE complete problem



straight line  
program

Model associated with an algebraic structure  
Sequence of operations on elements of a set



Arithmetical  
RAM

A Random Access Machine with  
operation set  $\{+, \cdot, \times, \div\}$

## Quantified Boolean Formulas

- Instance: A formula of the form  $Q_1x_1 \dots Q_nx_nF(x_1, \dots, x_n)$  where each  $Q_i \in \{\forall, \exists\}$  and  $F$  is a propositional formula in  $n$  variables
- Question: Does this formula evaluate to **true**?

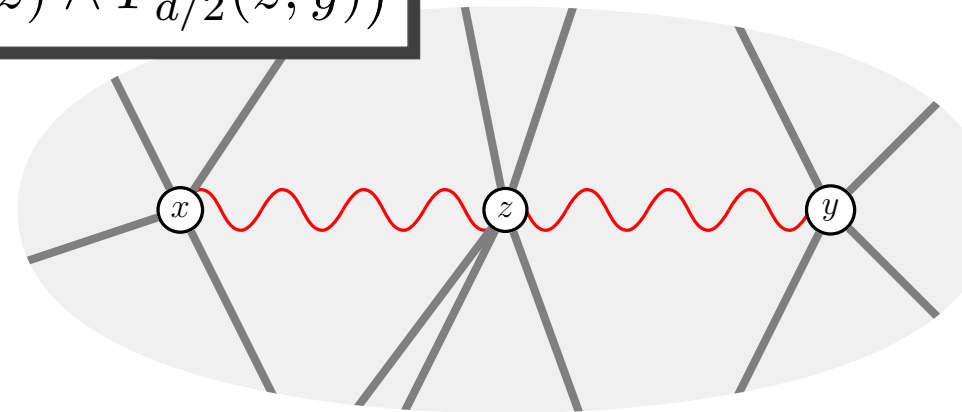
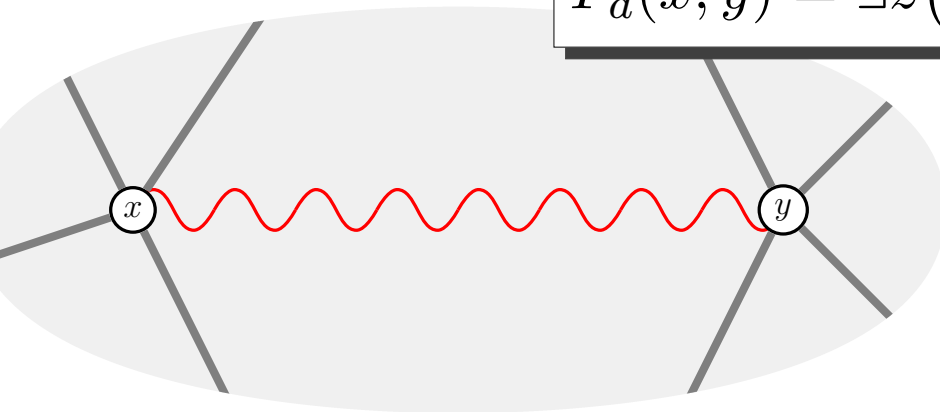
## Quantified Boolean Formulas

Instance: A formula of the form  $Q_1x_1 \dots Q_nx_nF(x_1, \dots, x_n)$  where each  $Q_i \in \{\forall, \exists\}$  and  $F$  is a propositional formula in  $n$  variables

Question: Does this formula evaluate to **true**?

**QBF and PSPACE - a recursive QBF for reachability:**

$$F_d(x, y) = \exists z (F_{d/2}(x, z) \wedge F_{d/2}(z, y))$$



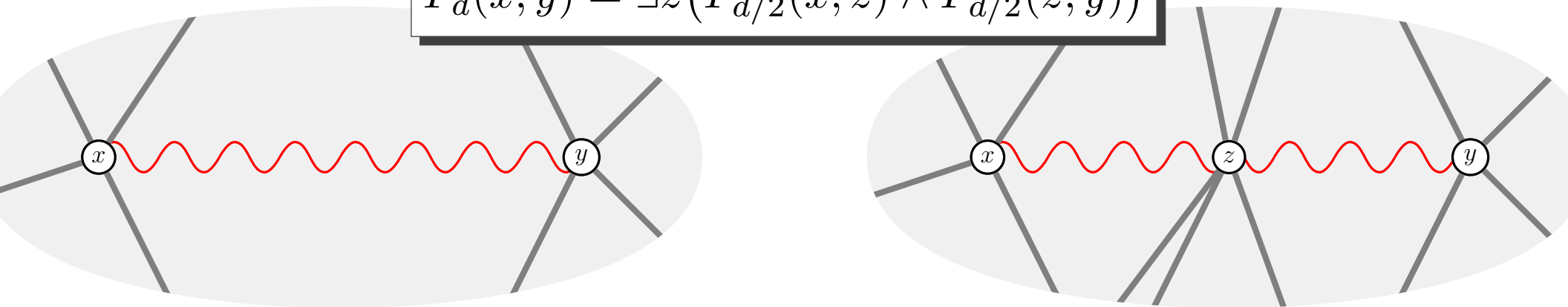
## Quantified Boolean Formulas

Instance: A formula of the form  $Q_1x_1 \dots Q_nx_nF(x_1, \dots, x_n)$  where each  $Q_i \in \{\forall, \exists\}$  and  $F$  is a propositional formula in  $n$  variables

Question: Does this formula evaluate to **true**?

**QBF and PSPACE - a recursive QBF for reachability:**

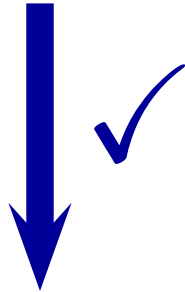
$$F_d(x, y) = \exists z (F_{d/2}(x, z) \wedge F_{d/2}(z, y))$$



**A logarithmic sized recursive QBF for reachability:**

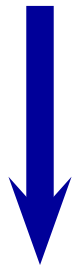
$$F_d(x, y) = \exists z \forall u \forall v \left( ((u = x \wedge v = z) \vee (u = z \wedge v = y)) \Rightarrow F_{d/2}(u, v) \right)$$

PSPACE



The Reachability Problem in digraphs  
can be expressed as a QBF of size logarithmic  
in the size of the graph

QBF



**Next**

Show that QBFs can be solved by  
evaluating a straight line program

straight line  
program

# Polynomials

$$\left( \sum_{k=0}^n x_k z^k = x_0 + x_1 z + x_2 z^2 + x_3 z^3 + \dots \right)$$

**Let  $f = f(z)$  and  $g = g(z)$  be polynomials in  $z$**

Addition:  $(f + g)_k = f_k + g_k$

Multiplication:  $(f \cdot g)_k = \sum_{j=0}^k f_j g_{k-j}$

Right Shift:  $(\downarrow f)_k = f_{k+1}$

Useful operations on GFs  $\left\{ \begin{array}{l} (f \otimes g)_k = f_k \times g_k \\ (\boxed{h}f)_k = f_{hk} \end{array} \right.$

Define the structure  $\mathcal{P} = \langle \mathbb{P}, +, \cdot, \otimes, \downarrow, \boxed{h} \rangle$ ,  
the set of polynomials and the operations defined above

## Straight Line Programs over $\mathcal{P}$

A straight-line program (SLP) of length  $n$  on the structure  $\mathcal{P}$  is sequence of  $n$  instructions such that:

The 1<sup>st</sup> instruction is of the form:

$$(1) \quad p_1 \leftarrow z \quad (\text{i.e. the elementary polynomial})$$

The  $k^{\text{th}}$  instruction is of the form:

$$(k) \quad p_k \leftarrow p_i + p_j \mid p_i \cdot p_j \mid p_i \otimes p_j \mid \downarrow p_i \mid \boxed{2} p_i \mid 1$$

If  $\Pi$  is an SLP on  $\mathcal{P}$  of length  $n$   
then  $\Pi$  **generates** the polynomial  $p_n$

- $\alpha(x_1, \dots, x_n)$  is a Boolean formula
- $\chi : \{x_1, \dots, x_n\} \mapsto \{0, 1\}$  is an interpretation
- $k =$  number with binary representation  $\chi(x_1)\chi(x_2) \cdots \chi(x_n)$

Define  $\alpha_k = \alpha(\chi(x_1), \dots, \chi(x_n))$

Generating Polynomial  
associated with  $\alpha$

$$p_\alpha(z) = \sum_{k=0}^{2^n-1} \alpha_k z^k$$

- $\alpha(x_1, \dots, x_n)$  is a Boolean formula
- $\chi : \{x_1, \dots, x_n\} \mapsto \{0, 1\}$  is an interpretation
- $k =$  number with binary representation  $\chi(x_1)\chi(x_2) \cdots \chi(x_n)$

Define  $\alpha_k = \alpha(\chi(x_1), \dots, \chi(x_n))$

Generating Polynomial  
associated with  $\alpha$

$$p_\alpha(z) = \sum_{k=0}^{2^n-1} \alpha_k z^k$$

If  $\alpha$  has  $n$  variables and  $m$  operations from  $\{\wedge, \vee, \neg\}$ , then  $p_\alpha$  can be generated by a SLP of length  $O(n + m)$ .

Proof by structural induction on  $\alpha$ :

$$p_{x_j} = (1 + x) \cdots (1 + x^{2^j-1}) x^{2^j} (1 + x^{2^j+1}) \cdots (1 + x^{2^n-1})$$

$$p_{\beta \vee \gamma} = p_\beta + p_\gamma - (p_\beta \otimes p_\gamma)$$

$$p_{\beta \wedge \gamma} = p_\beta \otimes p_\gamma$$

- $\psi = Q_1x_1 \dots Q_nx_n\alpha(x_1, \dots, x_n)$ ,  $Q_j \in \{\exists, \forall\}$ , is a QBF
- $Z_j = \prod_{x_j \in \{0,1\}}$  if  $Q_j = \forall$ , and  $Z_j = \sum_{x_j \in \{0,1\}}$  otherwise

The number of satisfying interpretations for  $\psi$

$$\#\psi = Z_1 \dots Z_n\alpha(x_1, \dots, x_n)$$

- $\psi = Q_1x_1 \dots Q_nx_n\alpha(x_1, \dots, x_n)$ ,  $Q_j \in \{\exists, \forall\}$ , is a QBF
- $Z_j = \prod_{x_j \in \{0,1\}}$  if  $Q_j = \forall$ , and  $Z_j = \sum_{x_j \in \{0,1\}}$  otherwise

The number of satisfying interpretations for  $\psi$

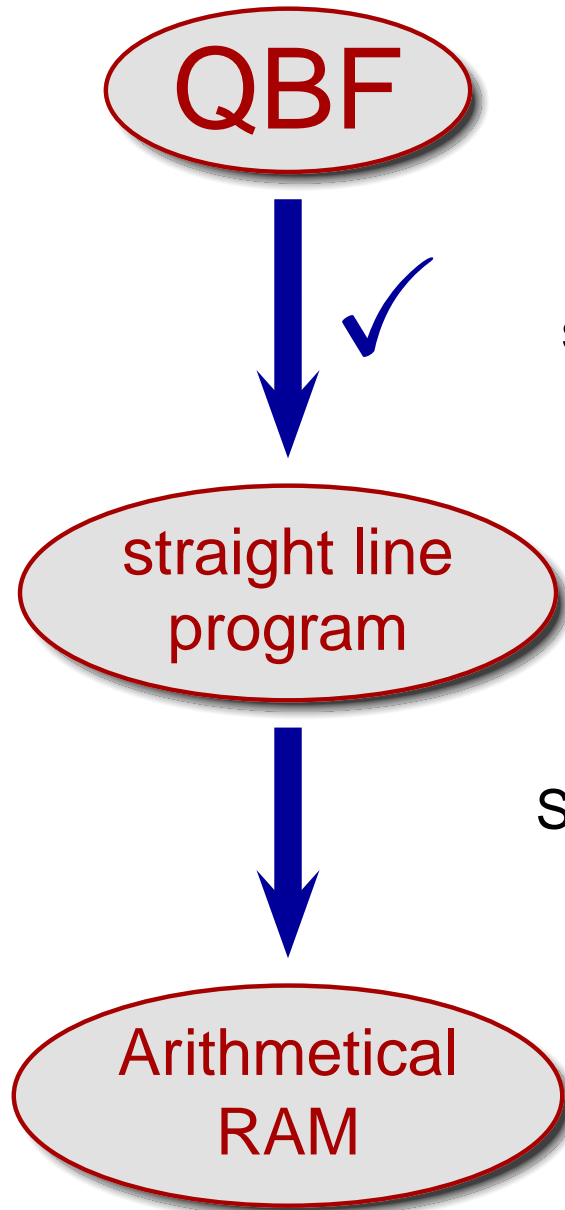
$$\#\psi = Z_1 \dots Z_n \alpha(x_1, \dots, x_n)$$

If  $\psi$  is a QBF of size  $n$  then  $\#\psi$  can be computed by means of a length  $O(n)$  straight-line program over  $\mathcal{P}$

$$p(0) = p_\alpha = \sum_k \alpha_k z^k$$

$$Q_{n-j+1} = \forall \quad \Rightarrow \quad p(j) = \boxed{2}(p(j-1) \otimes (\downarrow p(j-1))),$$

$$Q_{n-j+1} = \exists \quad \Rightarrow \quad p(j) = \boxed{2}(p(j-1) + (\downarrow p(j-1)))$$



QBFs can be solved by a polynomial sized SLP over polynomials, which evaluates to the number of models for the QBF

**Next**

Show that the operations of an Arithmetic RAM are sufficient for evaluating an SLP over  $\mathcal{P}$  in polynomial time

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\begin{array}{l} \text{Addition:} \\ \text{Multiplication:} \\ \text{Right Shift:} \end{array} \begin{array}{l} (f + g)(x) = f(x) + g(x) \\ (f \cdot g)(x) = f(x) \times g(x) \\ (\downarrow f)(x) = f(x) \div x \end{array} \left. \vphantom{\begin{array}{l} \text{Addition:} \\ \text{Multiplication:} \\ \text{Right Shift:} \end{array}} \right\} \begin{array}{c} \text{easy} \\ \text{operations} \end{array}$$

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\left. \begin{array}{l} \text{Addition: } (f + g)(x) = f(x) + g(x) \\ \text{Multiplication: } (f \cdot g)(x) = f(x) \times g(x) \\ \text{Right Shift: } (\downarrow f)(x) = f(x) \div x \end{array} \right\} \text{easy operations}$$

$$(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \bmod (z^{n+2} - x) \right) \bmod z$$
$$(\boxed{h}f)(x) = \left( f(z) \bmod z^h - x \right) \bmod z$$

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\begin{array}{l}
 \text{Addition: } (f + g)(x) = f(x) + g(x) \\
 \text{Multiplication: } (f \cdot g)(x) = f(x) \times g(x) \\
 \text{Right Shift: } (\downarrow f)(x) = f(x) \div x
 \end{array}
 \left. \vphantom{\begin{array}{l} \text{Addition: } \\ \text{Multiplication: } \\ \text{Right Shift: } \end{array}} \right\} \text{easy operations}$$

change variable:  
carries don't propagate

high order terms  
act as workspace

$$(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \bmod (z^{n+2} - x) \right) \bmod z$$

$z$  must be  
sufficiently large

$$(\boxed{h}f)(x) = \left( f(z) \bmod z^h - x \right) \bmod z$$

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\begin{array}{l}
 \text{Addition: } (f + g)(x) = f(x) + g(x) \\
 \text{Multiplication: } (f \cdot g)(x) = f(x) \times g(x) \\
 \text{Right Shift: } (\downarrow f)(x) = f(x) \div x
 \end{array}
 \left. \vphantom{\begin{array}{l} \text{Addition: } \\ \text{Multiplication: } \\ \text{Right Shift: } \end{array}} \right\} \text{easy operations}$$

change variable:  
carries don't propagate

high order terms  
act as workspace

mod function  
cleans things up

$$(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \bmod (z^{n+2} - x) \right) \bmod z$$

$z$  must be  
sufficiently large

$$(\overline{h}f)(x) = \left( f(z) \bmod z^h - x \right) \bmod z$$

mod function  
easily computed

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\begin{array}{l}
 \text{Addition: } (f + g)(x) = f(x) + g(x) \\
 \text{Multiplication: } (f \cdot g)(x) = f(x) \times g(x) \\
 \text{Right Shift: } (\downarrow f)(x) = f(x) \div x
 \end{array}
 \left. \vphantom{\begin{array}{l} \text{Addition: } \\ \text{Multiplication: } \\ \text{Right Shift: } \end{array}} \right\} \text{easy operations}$$

change variable:  
carries don't propagate

high order terms  
act as workspace

mod function  
cleans things up

$$(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \bmod (z^{n+2} - x) \right) \bmod z$$

$z$  must be  
sufficiently large

$$(\overline{h}f)(x) = \left( f(z) \bmod z^h - x \right) \bmod z$$

mod function  
easily computed

correctness proofs:  
divisibility results

# Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

$$\begin{array}{l}
 \text{Addition: } (f + g)(x) = f(x) + g(x) \\
 \text{Multiplication: } (f \cdot g)(x) = f(x) \times g(x) \\
 \text{Right Shift: } (\downarrow f)(x) = f(x) \div x
 \end{array}
 \left. \vphantom{\begin{array}{l} \text{Addition: } \\ \text{Multiplication: } \\ \text{Right Shift: } \end{array}} \right\} \text{easy operations}$$

change variable:  
carries don't propagate

high order terms  
act as workspace

mod function  
cleans things up

$$(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \bmod (z^{n+2} - x) \right) \bmod z$$

$z$  must be  
sufficiently large

$$(\overline{h}f)(x) = \left( f(z) \bmod z^h - x \right) \bmod z$$

mod function  
easily computed

correctness proofs:  
divisibility results

**Conclusion:**  
Arithmetic RAMs belong to the Second Machine Class

# The Standard Model

For a problem instance  $x$ , the standard model is a unit cost RAM with  $O(|x|^{O(1)})$  registers each having size  $O(\log |x|)$ . The operations include any arithmetic or Boolean operations on numbers stored in registers.

# The Standard Model

For a problem instance  $x$ , the standard model is a unit cost RAM with  $O(|x|^{O(1)})$  registers each having size  $O(\log |x|)$ . The operations include any arithmetic or Boolean operations on numbers stored in registers.

**Polynomial space**  $\Rightarrow$  Usual way of thinking about space: it must be initialized  
Initializing exponential space requires exponential time

# The Standard Model

For a problem instance  $x$ , the standard model is a unit cost RAM with  $O(|x|^{O(1)})$  registers each having size  $O(\log |x|)$ . The operations include any arithmetic or Boolean operations on numbers stored in registers.

**Polynomial space**  $\Rightarrow$  Usual way of thinking about space: it must be initialized  
Initializing exponential space requires exponential time

**Log sized registers**  $\Rightarrow$  Allows indexing a polynomial number of registers  
Also allows counting a polynomial number of elements

# The Standard Model

For a problem instance  $x$ , the standard model is a unit cost RAM with  $O(|x|^{O(1)})$  registers each having size  $O(\log |x|)$ . The operations include any arithmetic or Boolean operations on numbers stored in registers.

**Polynomial space**  $\Rightarrow$  Usual way of thinking about space: it must be initialized  
Initializing exponential space requires exponential time

**Log sized registers**  $\Rightarrow$  Allows indexing a polynomial number of registers  
Also allows counting a polynomial number of elements

**Usual Operations**  $\Rightarrow$  Functions with operators in the C programming language  
Arithmetic operations:  $\{+, -, \times, \div\}$   
Boolean operations:  $\{\wedge, \vee, \neg\}$

# The Standard Model

For a problem instance  $x$ , the standard model is a unit cost RAM with  $O(|x|^{O(1)})$  registers each having size  $O(\log |x|)$ . The operations include any arithmetic or Boolean operations on numbers stored in registers.

**Polynomial space** ⇒ Usual way of thinking about space: it must be initialized  
Initializing exponential space requires exponential time

**Log sized registers** ⇒ Allows indexing a polynomial number of registers  
Also allows counting a polynomial number of elements

**Usual Operations** ⇒ Functions with operators in the C programming language  
Arithmetic operations:  $\{+, -, \times, \div\}$   
Boolean operations:  $\{\wedge, \vee, \neg\}$

**Acceptable Operations** ⇒ Many operations of the form  $\{0, 1\}^{\log n} \mapsto \{0, 1\}^{\log n}$   
Lookup tables can be constructed usually in linear time  
Example: Threshold Functions, Bit Counting