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

\[ X_i \leftarrow c, \ c \in \mathbb{Z} \]  \hspace{3cm} \text{Literal assignment (always unit cost)}

\[ X_i \leftarrow X_{X_j} \]
\[ X_{X_i} \leftarrow X_j \]  \hspace{3cm} \text{Indirect addressing (central to Random Access)}

\[ \text{IF } X_i > 0 \text{ GOTO } m \]  \hspace{3cm} \text{Conditional transfer (required for Turing equivalence)}

\text{READ } X_i
\text{WRITE } X_i  \hspace{3cm} \text{Input and Output operations}

\[ X_i \leftarrow X_j + X_k \]
\[ X_i \leftarrow X_j - X_k \]  \hspace{3cm} \text{Addition and Subtraction}

\begin{itemize}
  \item Random Access allows indirect addressing, essential for abstract data types such as list and trees
  \item The RAM is also a cognitive aid, and permits an algorithm designer to visualize the manipulation of abstract structures
\end{itemize}
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 $P_0, P_1, \ldots$
- 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 $X_0, X_1, \ldots$
- 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 $\text{FORK}(x)$, 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
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 $\text{FORK}(x)$, 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

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 \textsc{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

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 → write goes first
Concurrent write causes immediate halt in rejecting state

\{\text{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 $\text{FORK}(x)$, 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

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 → write goes first
Concurrent write causes immediate halt in rejecting state

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

**Two Consequences**

<table>
<thead>
<tr>
<th>PRAM-LOGTIME = LOGSPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Related to AC and NC hierarchies</td>
</tr>
<tr>
<td>• Well known result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRAM-PTIME = PSPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Write conflict irrelevant mod P</td>
</tr>
<tr>
<td>• Of Interest Here</td>
</tr>
</tbody>
</table>
The most fundamental result in parallel complexity theory
Connects parallel complexity to the world of P =? NP

Two Consequences

<table>
<thead>
<tr>
<th>PRAM-LOGTIME = LOGSPACE</th>
<th>PRAM-PTIME = PSPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Related to AC and NC hierarchies</td>
<td></td>
</tr>
<tr>
<td>• Well known result</td>
<td></td>
</tr>
<tr>
<td>• Write conflict irrelevant mod P</td>
<td></td>
</tr>
<tr>
<td>• Of Interest Here</td>
<td></td>
</tr>
</tbody>
</table>

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
PSPACE

Simulates in polynomial time

PRAM

Proved with a generic transformation from PSPACE bounded (TM, input) pair to a graph problem easily solved on a PRAM

Parallelism and Machine Models

Andrew D Smith, October 25 2003
\[ \bigcup_k \text{PRAM-TIME}(T^k(n)) \subseteq \bigcup_k \text{TM-SPACE}(T^k(n)) \]

How it works: Transform Turing Machine Acceptance into Graph Reachability

\[ M \text{ is a Turing Machine, } x \text{ is the input, } S \text{ is the tape used} \]

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
\[ G(x, M, S) \text{ has out-degree } \leq 1 \]
(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 \( x \)
How it works: Transform Turing Machine Acceptance into Graph Reachability

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

\( M \) is a Turing Machine, \( x \) is the input, \( S \) is the tape used

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

Counting States:
\[ S \in O(n^c) \Rightarrow M \text{ has } 2^{O(n^c)} \text{ states} \]

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

\( (\text{node } s \equiv \text{ initial configuration}) \), \( (\text{node } t \equiv \text{ final configuration}) \)

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 \( \Rightarrow \text{NC is really polynomial} \)
**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 $t$. 

**Illustration:**

- Initial graph with edges from 1 to 2, 3, and 4, and from 4 to 5.
- Transitive closure graph.
- Possible first use of pointer jumping!
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 $(I + A)$

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

$(I + A)^2$

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

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

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

Possibly the first use of pointer jumping!
Now for the other direction...

Must design a PSPACE bounded TM that can simulate an arbitrary PRAM program

Obtained by reducing PSPACE TM acceptance to graph reachability
\[ \cup_k \text{PRAM-TIME}(T^k(n)) \supseteq \cup_k \text{TM-SPACE}(T^k(n)) \]

**Mutually Recursive Procedures**
Verify the instruction executed, and the contents of memory and accumulators at each time

**VERIFY_ACCUMULATOR**
Guess: time \( t' < t \), instruction \( i \)
Verify \( A_j \) last modified at time \( t' \)
\( P_j \) executed \( i \) at time \( t' \)
\( i \) executed at time \( t' \) produces \( c \)

**VERIFY_MEMORY**
Guess: processor \( P_j \), time \( t' < t \)
Verify \[ P_j \text{ stored to } X \text{ at time } t' \]
\[ A_j \text{ contained } c \text{ at time } t' \]
no store to \( X \) between \( t' \) and \( t \)

**VERIFY_INSTRUCTION**
Guess: processor \( P_j \), instruction \( i' \)
Verify \[ P_j \text{ executed } i' \text{ at time } t - 1 \]
if \( i' \neq i - 1 \) then \( i' \) jumps to \( i \)

**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 \} \text{ are more complex than } \{ +, - \}

\begin{align*}
+,- & \in \mathsf{AC}^0 \text{ but } \times, \div \notin \mathsf{AC}^0 \\
\text{(Furst, Saxe & Sipser, 1981)}
\end{align*}
Multiplication is controversial!

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

original evidence: circuit complexity

\[ +, - \in \mathsf{AC}^0 \text{ but } \times, \div \notin \mathsf{AC}^0 \]

(Furst, Saxe & Sipser, 1981)

**Hartmanis & Simon (1974)**

Proved RAM with \{ +, -, \times, \div, \land, \lor, \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

\[+, - \in AC^0 \text{ but } \times, \div \notin AC^0\]  
(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)

Part 1

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 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 \{+, -, \times, \div\}
Quantified Boolean Formulas

Instance: A formula of the form $Q_1 x_1 \ldots Q_n x_n F(x_1, \ldots, x_n)$ 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 $Q_1 x_1 \ldots Q_n x_n F(x_1, \ldots, x_n)$ 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:

$$F_d(x, y) = \exists z (F_{d/2}(x, z) \land F_{d/2}(z, y))$$
Quantified Boolean Formulas

Instance: A formula of the form \( Q_1x_1 \ldots Q_nx_nF(x_1, \ldots, x_n) \) 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:

\[
F_d(x, y) = \exists z (F_{d/2}(x, z) \land F_{d/2}(z, y))
\]

A logarithmic sized recursive QBF for reachability:

\[
F_d(x, y) = \exists z \forall u \forall v \left( ((u = x \land u = z) \lor (u = z \land u = y)) \Rightarrow F_{d/2}(u, v) \right)
\]
The Reachability Problem in digraphs can be expressed as a QBF of size logarithmic in the size of the graph.

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

\[ (\sum_{k=0}^{n} x_k z^k = x_0 + x_1 z + x_2 z^2 + x_3 z^3 + \ldots ) \]

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 \( \big\{ (f \otimes g)_k = f_k \times g_k \)

Define the structure \( \mathcal{P} = \langle \mathbb{P}, +, \cdot, \otimes, \downarrow, [h] \rangle \),
the set of polynomials and the operations defined above
Straight Line Programs over $P$

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

The $1^{\text{st}}$ instruction is of the form:

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

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

$$(k) \quad p_k \leftarrow p_i + p_j \mid p_i \cdot p_j \mid p_i \otimes p_j \mid \downarrow p_i \mid \lfloor 2 \rfloor p_i \mid 1$$

If $\Pi$ is an SLP on $P$ of length $n$ then $\Pi$ generates the polynomial $p_n$
\( \alpha(x_1, \ldots, x_n) \) is a Boolean formula

\( \chi : \{x_1, \ldots, x_n\} \mapsto \{0, 1\} \) is an interpretation

\( k \) = number with binary representation \( \chi(x_1)\chi(x_2) \cdots \chi(x_n) \)

Define \( \alpha_k = \alpha(\chi(x_1), \ldots, \chi(x_n)) \)

Generating Polynomial associated with \( \alpha \)

\[
p_{\alpha}(z) = \sum_{k=0}^{2^n-1} \alpha_k z^k
\]
\[ \alpha(x_1, \ldots, x_n) \] is a Boolean formula
\[ \chi : \{x_1, \ldots, x_n\} \mapsto \{0, 1\} \] is an interpretation
\( k \) = number with binary representation \( \chi(x_1)\chi(x_2)\cdots\chi(x_n) \)

Define \( \alpha_k = \alpha(\chi(x_1), \ldots, \chi(x_n)) \)

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 \( \{\land, \lor, \neg\} \), then \( p_\alpha \) can be generated by a SLP of length \( O(n + m) \).

Proof by structural induction on \( \alpha \):
\[ p_{x_j} = (1 + x) \cdots (1 + x^{2^{j-1}})x^{2^j} (1 + x^{2^j+1}) \cdots (1 + x^{2^n-1}) \]
\[ p_{\beta \lor \gamma} = p_\beta + p_\gamma - (p_\beta \otimes p_\gamma) \]
\[ p_{\beta \land \gamma} = p_\beta \otimes p_\gamma \]
\[ \psi = Q_1 x_1 \ldots Q_n x_n \alpha(x_1, \ldots, x_n), \quad Q_j \in \{\exists, \forall\}, \text{ is a QBF} \]

\[ Z_j = \prod_{x_j \in \{0,1\}} \text{ if } Q_j = \forall, \text{ and } Z_j = \sum_{x_j \in \{0,1\}} \text{ otherwise} \]

The number of satisfying interpretations for \( \psi \)

\[ \#\psi = Z_1 \ldots Z_n \alpha(x_1, \ldots, x_n) \]
ψ = Q_1 x_1 \ldots Q_n x_n \alpha(x_1, \ldots, 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 ψ

\#ψ = Z_1 \ldots Z_n \alpha(x_1, \ldots, x_n)

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

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

\[ Q_{n-j+1} = \forall \Rightarrow p(j) = \overline{2}(p(j - 1) \otimes (\downarrow p(j - 1))) \]

\[ Q_{n-j+1} = \exists \Rightarrow p(j) = \overline{2}(p(j - 1) + (\downarrow p(j - 1))) \]
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

Addition: $(f + g)(x) = f(x) + g(x)$
Multiplication: $(f \cdot g)(x) = f(x) \times g(x)$
Right Shift: $(\downarrow f)(x) = f(x) \div x$

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

Addition: \[(f + g)(x) = f(x) + g(x)\]
Multiplication: \[(f \cdot g)(x) = f(x) \times g(x)\]
Right Shift: \[(\downarrow f)(x) = f(x) \div x\]

\[
(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \mod (z^{n+2} - x) \right) \mod z
\]
\[
(hf)(x) = \left( f(z) \mod z^h - x \right) \mod z
\]
Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

Addition: \[(f + g)(x) = f(x) + g(x)\]
Multiplication: \[(f \cdot g)(x) = f(x) \times g(x)\]
Right Shift: \[(\downarrow f)(x) = f(x) \div x\]

\[
(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \mod (z^{n+2} - x) \right) \mod z
\]
\[
(h \hat{f})(x) = \left( f(z) \mod z^h - x \right) \mod z
\]

clear variable: carries don’t propagate
high order terms act as workspace
$z$ must be sufficiently large

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

**Addition:**

$$(f + g)(x) = f(x) + g(x)$$

**Multiplication:**

$$(f \cdot g)(x) = f(x) \times g(x)$$

**Right Shift:**

$$(\downarrow f)(x) = f(x) \div x$$

**Easy operations**

- Change variable: carries don’t propagate
- High order terms act as workspace
- Mod function cleans things up

\[
(f \otimes g)(x) = \left( f(z)g(z^{n+1}) \mod (z^{n+2} - x) \right) \mod z
\]

\[
(hf)(x) = \left( f(z) \mod z^h - x \right) \mod z
\]

$z$ must be sufficiently large

Mod function easily computed

Parallelism and Machine Models

Andrew D Smith, October 25 2003
Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

Addition: $$(f + g)(x) = f(x) + g(x)$$

Multiplication: $$(f \cdot g)(x) = f(x) \times g(x)$$

Right Shift: $$(\downarrow f)(x) = f(x) \div x$$

\[
(f \otimes g)(x) = \left( f(z) g(z^{n+1}) \mod (z^{n+2} - x) \right) \mod z
\]

\[
(h \lfloor f \rfloor)(x) = \left( f(z) \mod z^h - x \right) \mod z
\]

- Change variable: carries don’t propagate
- Mod function cleans things up
- High order terms act as workspace
- \(z\) must be sufficiently large
- Mod function easily computed
- Correctness proofs: divisibility results

Parallelism and Machine Models
Andrew D Smith, October 25 2003
Simulating SLPs over $\mathcal{P}$ with Arithmetic RAMs

Addition: $(f + g)(x) = f(x) + g(x)$

Multiplication: $(f \cdot g)(x) = f(x) \times g(x)$

Right Shift: $(\downarrow f)(x) = f(x) \div x$

easy operations

change variable: carries don’t propagate

high order terms act as workspace

mod function cleans things up

$z$ must be sufficiently large

correctness proofs: divisibility results

mod function easily computed

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

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

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  

Usual way of thinking about space: it must be initialized. Initializing exponential space requires exponential time.

Lookup tables can be constructed usually in linear time. Example: Threshold Functions, Bit Counting.
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
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: $\{\land,\lor,\neg\}$
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: \(\{\land, \lor, \neg\}\)

**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