Descriptive Complexity: Parallelism and Circuit Complexity

Thomas Pipilikas

INTER-INSTITUTIONAL GRADUATE PROGRAM "ALGORITHMS, LOGIC AND DISCRETE MATHEMATICS"

Overview

- The real world is inherently parallel
- Descriptive complexity is inherently parallel in nature.
- Quantification is a parallel operation.

- The real world is inherently parallel
- Descriptive complexity is inherently parallel in nature.
- Quantification is a parallel operation.

- Some problems are very easy to parallelize.
- We have increased the ability to produce small, fast, inexpensive proccessors.
- A proccessors speed is bounded.
- Memory requirements

A Motivation example

Find all prime numbers in the interval $[1, n]$

Algorithm **Sieve of Eratosthenes**

A Motivation example

Find all prime numbers in the interval $[1, n]$

Algorithm **Sieve of Eratosthenes**

Start with the list of numbers $1, 2, \ldots, n$ represented as a "mark" bit-vector initialized to 1000...00.

A Motivation example

Find all prime numbers in the interval $[1, n]$

Algorithm **Sieve of Eratosthenes**

Start with the list of numbers $1, 2, ..., n$ represented as a "mark" bit-vector initialized to 1000...00.

In each step, the next unmarked number m (associated with a 0 in element m of the mark bit-vector) is a prime.

A Motivation example

Find all prime numbers in the interval $[1, n]$

Algorithm **Sieve of Eratosthenes**

Start with the list of numbers $1, 2, ..., n$ represented as a "mark" bit-vector initialized to 1000...00.

In each step, the next unmarked number m (associated with a 0 in element m of the mark bit-vector) is a prime.

Find this element m and mark all multiples of m beginning with m^2.

A Motivation example

Find all prime numbers in the interval $[1, n]$

Algorithm **Sieve of Eratosthenes**

Start with the list of numbers $1, 2, ..., n$ represented as a "mark" bit-vector initialized to 1000...00.

In each step, the next unmarked number m (associated with a 0 in element m of the mark bit-vector) is a prime.

Find this element m and mark all multiples of m beginning with m^2.

When $m^2 > n$, the computation stops and all unmarked elements are prime numbers.

A Motivation example

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
$m=2$																												
2	3		5		7		9		11		13		15		17		19		21		23		25		27		29	
	$m=3$																											
2	3		5		7				11		13				17		19				23		25				29	
			$m=5$																									
2	3		5		7				11		13				17		19				23						29	
					$m=7$																							

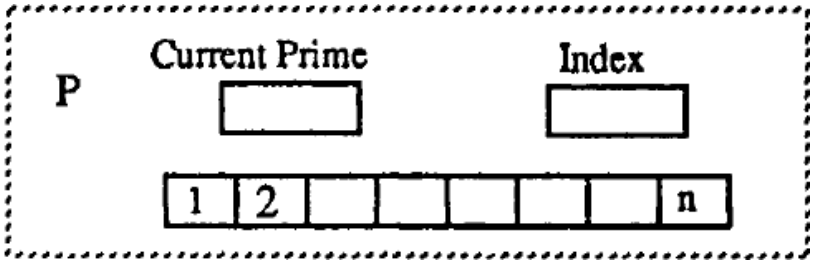
Sieve of Eratosthenes for $n = 30$

A Motivation example

A **Single-proccessor** for the algorithm

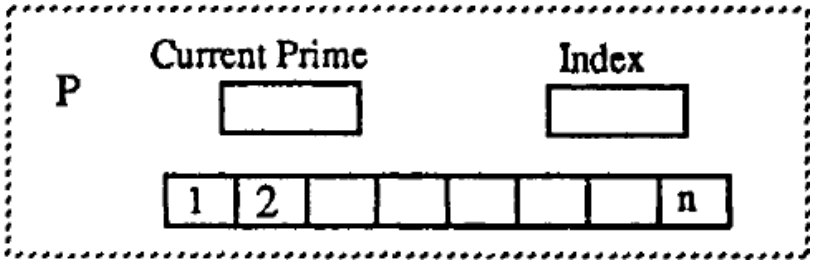
A Motivation example

A **Single-proccessor** for the algorithm



A Motivation example

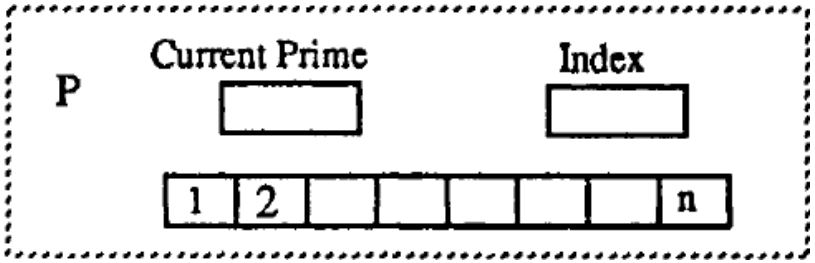
A **Single-proccessor** for the algorithm



- *Current Prime* contains the latest prime number found (initialized to 2).

A Motivation example

A **Single-proccessor** for the algorithm



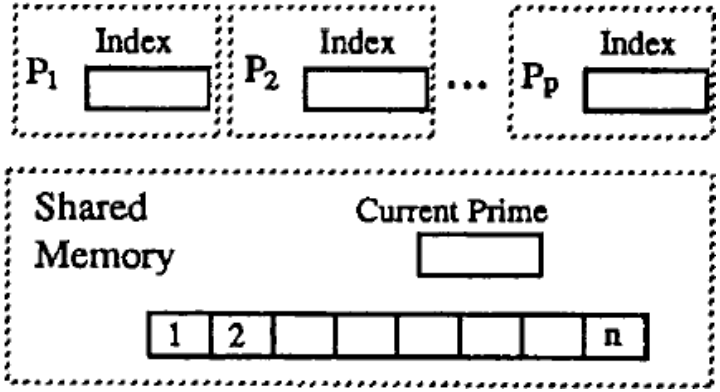
- *Current Prime* contains the latest prime number found (initialized to 2).
- *Index* is initialized to the square of Current Prime. Then incriments by Current Prime in order to mark all of its multiples.

A Motivation example

A **Parallel p-proccessors machine** for the algorithm

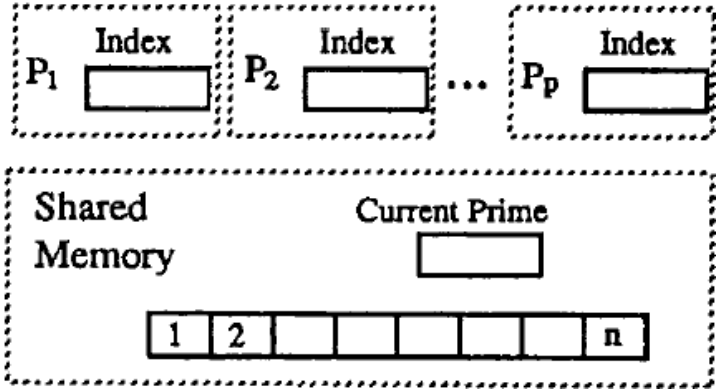
A Motivation example

A **Parallel p-proccessors machine** for the algorithm



A Motivation example

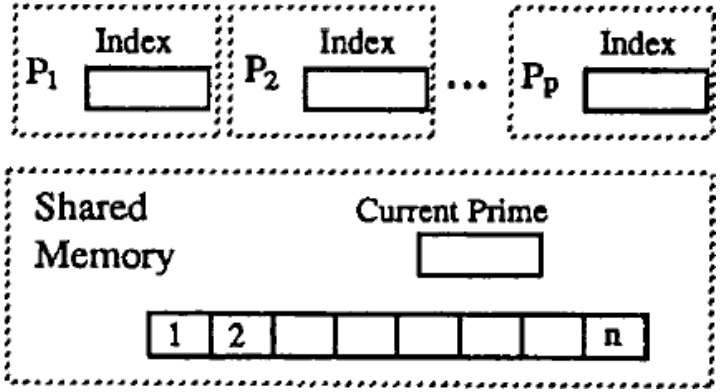
A **Parallel p-proccessors machine** for the algorithm



- *Shared Memory* contains Curent Prime and the list of numbers.

A Motivation example

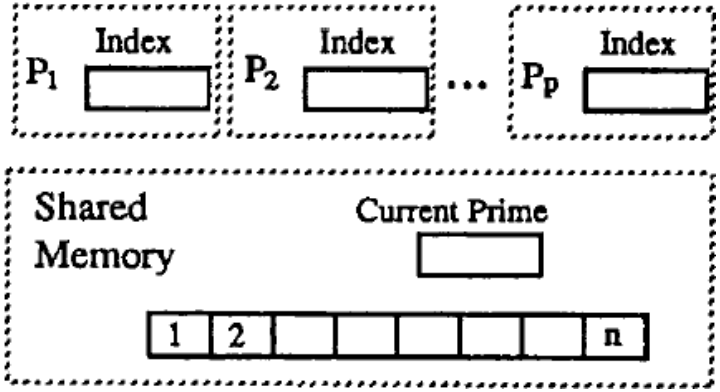
A **Parallel p-proccessors machine** for the algorithm



- *Shared Memory* contains Curent Prime and the list of numbers.
- Each *Proccessor* refers to the shared memory:

A Motivation example

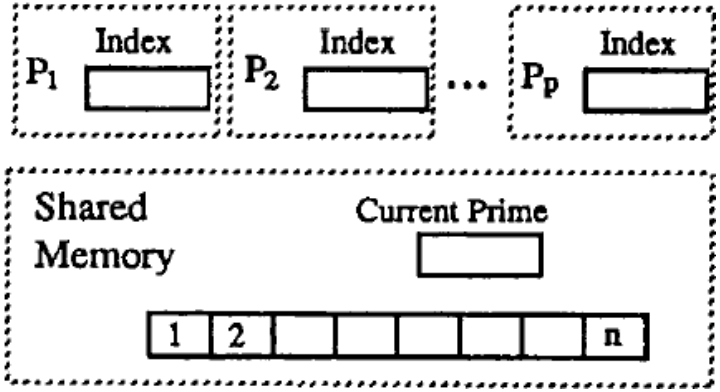
A **Parallel p-proccessors machine** for the algorithm



- *Shared Memory* contains Curent Prime and the list of numbers.
- Each *Proccessor* refers to the shared memory:
  - Updates Current Prime

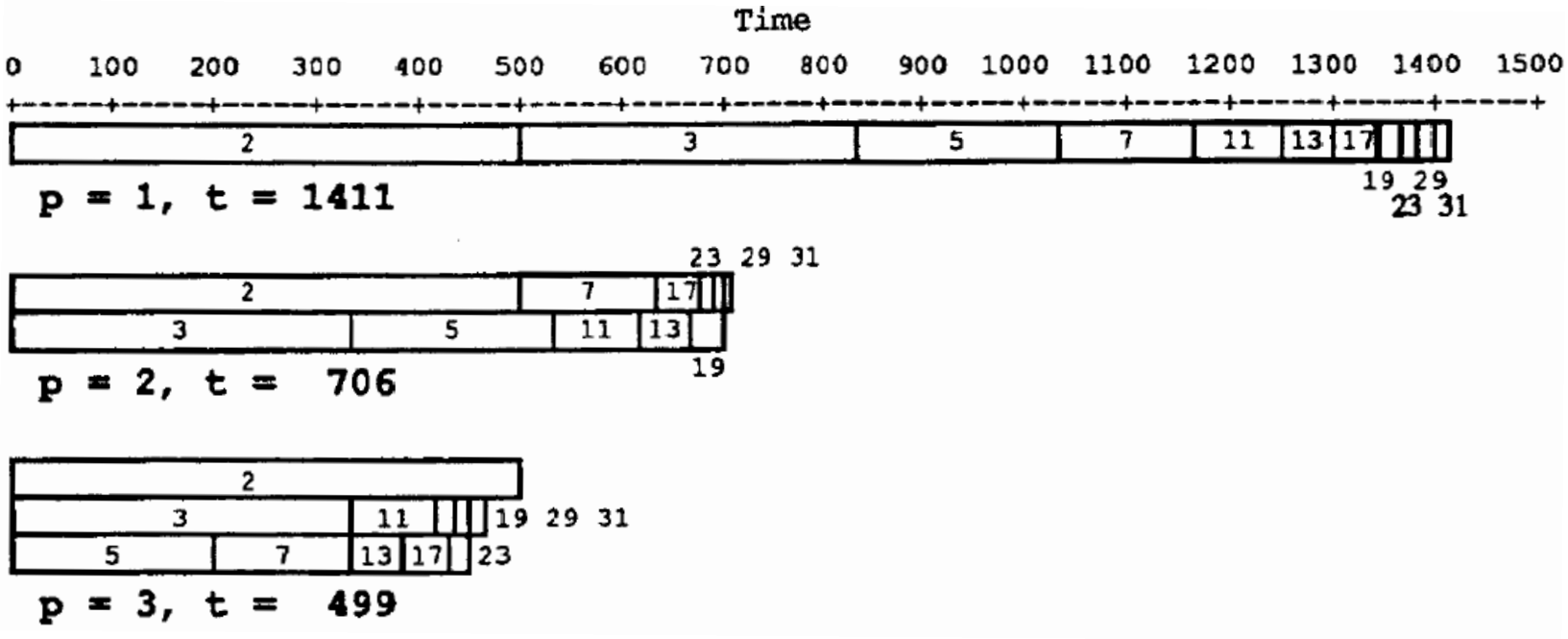
A Motivation example

A **Parallel p-proccessors machine** for the algorithm



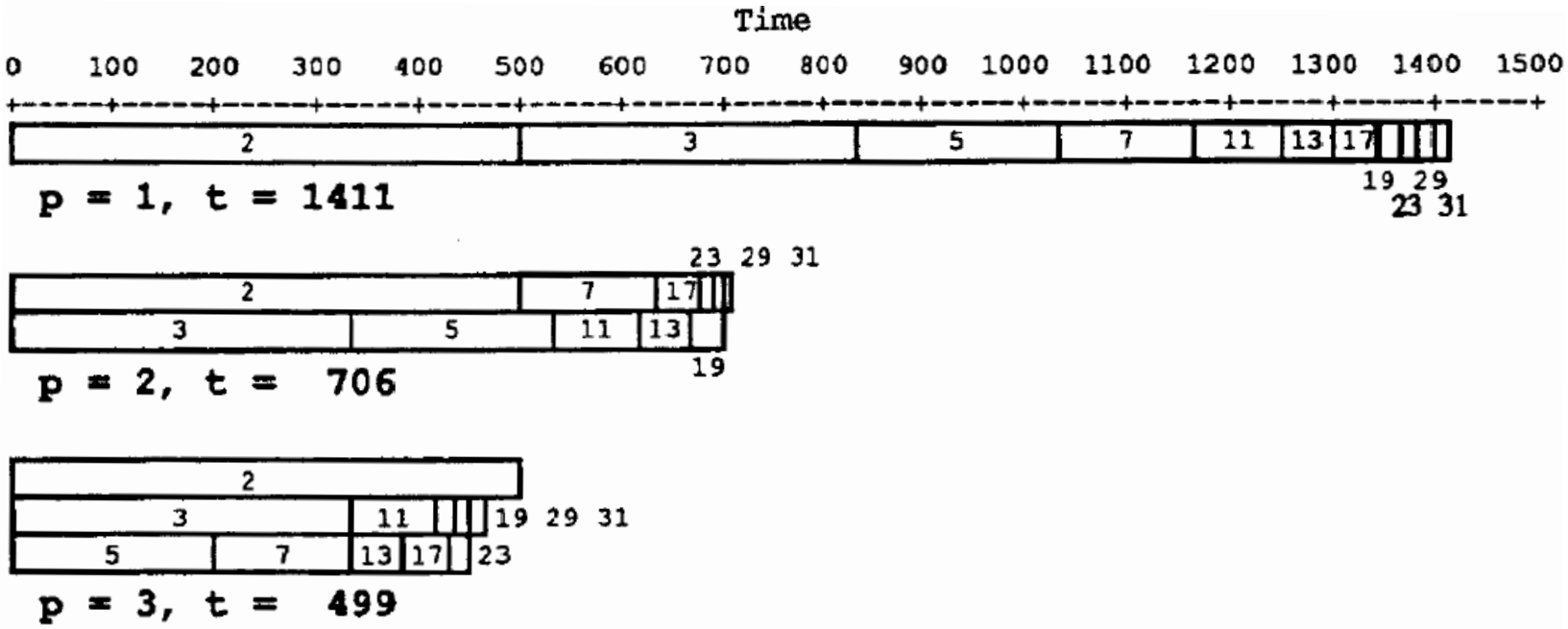
- *Shared Memory* contains Curent Prime and the list of numbers.
- Each *Proccessor* refers to the shared memory:
  - Updates Current Prime
  - Uses its private index to step through the list and mark the multiples of the prime that updated.

A Motivation example



Implementation of the algorithm for $p \in [3]$ proccessors and $n = 1000$

A Motivation example

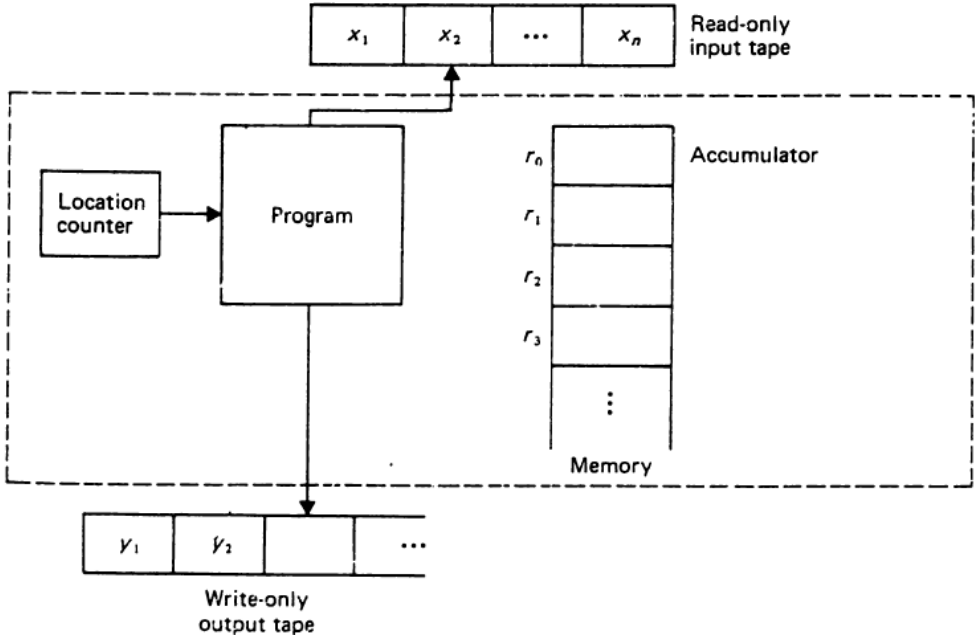


Implementation of the algorithm for $p \in [3]$ proccessors and $n = 1000$

Note that by using more than three proccessors would not reduce the computation time. (Why?)

Random Access Machine

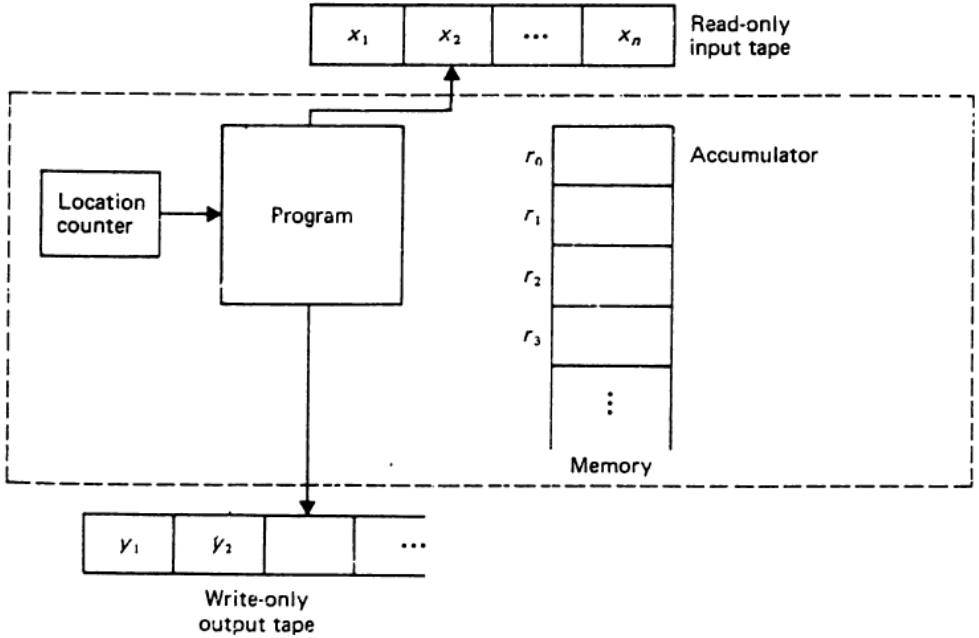
A *random access machine* (RAM) is one-accumulator computer that consists:



Random Access Machine

A *random access machine* (RAM) is one-accumulator computer that consists:

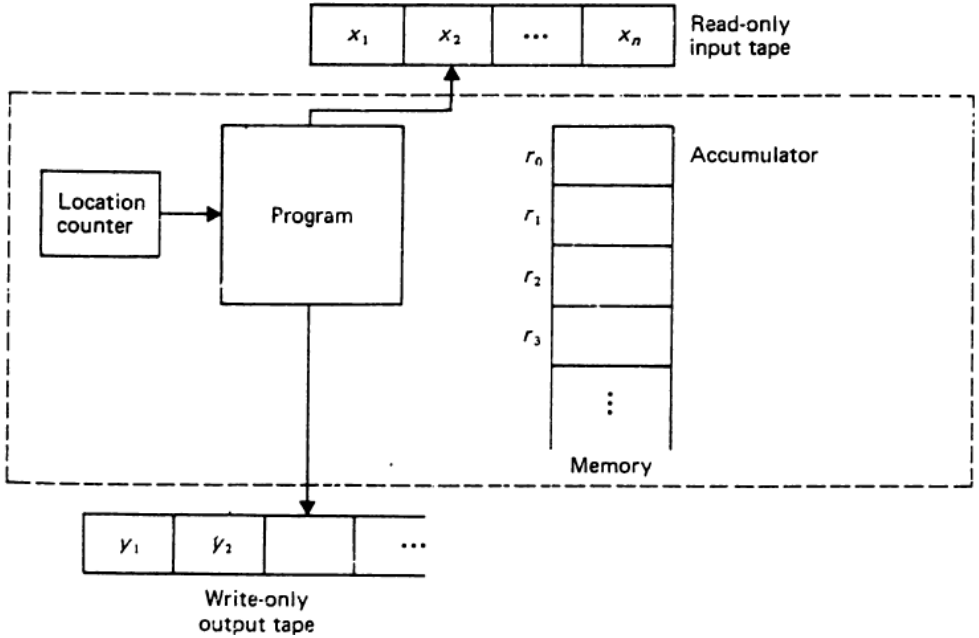
- A *read-only input tape*



Random Access Machine

A *random access machine* (RAM) is one-accumulator computer that consists:

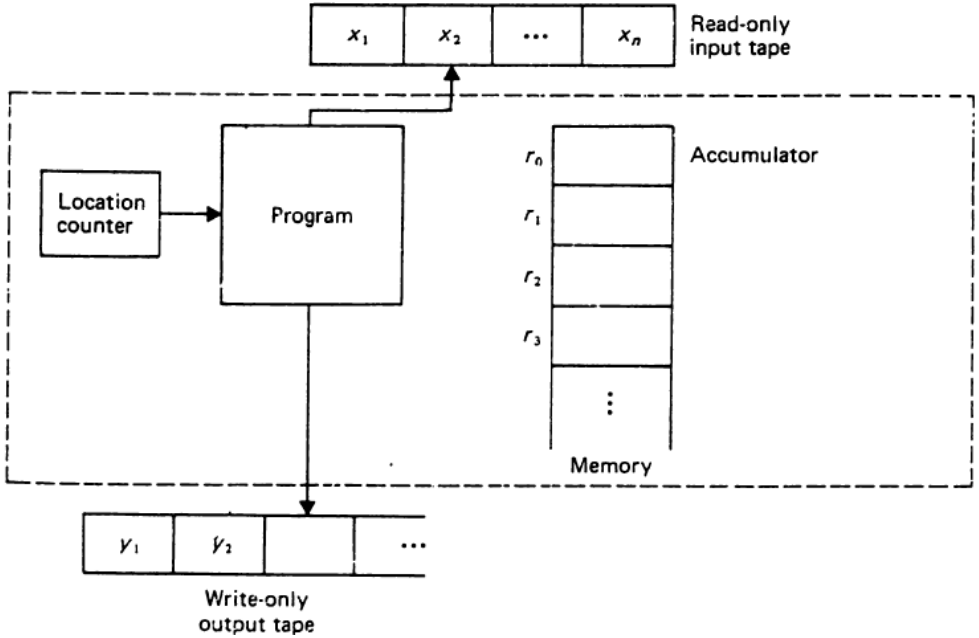
- A *read-only input tape*
- An infinite *wright-only output tape*



Random Access Machine

A *random access machine* (RAM) is one-accumulator computer that consists:

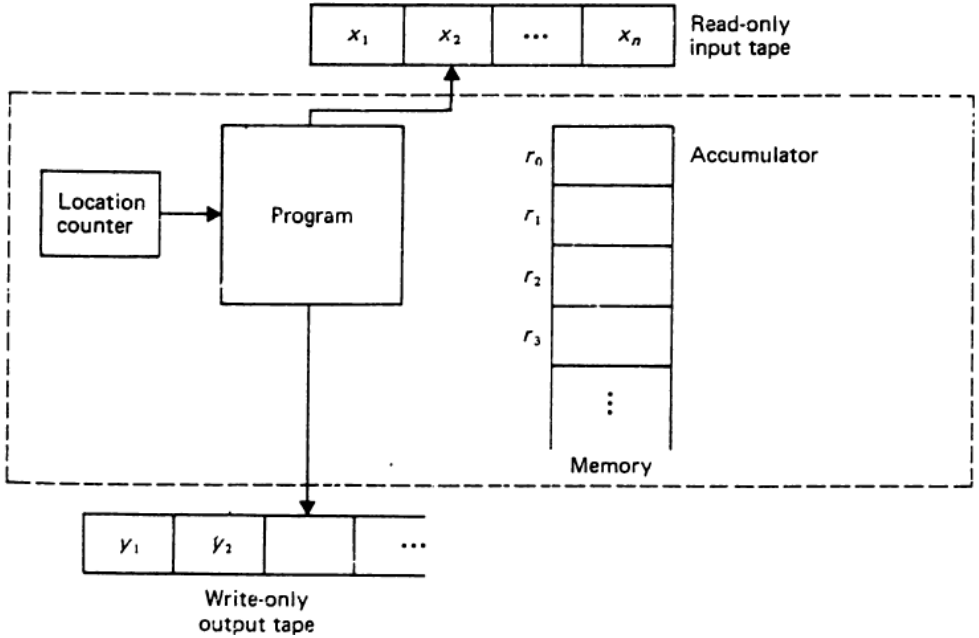
- A *read-only input tape*
- An infinite *wright-only output tape*
- A *program*, which contains a sequence of *instrutions*



Random Access Machine

A *random access machine* (RAM) is one-accumulator computer that consists:

- A *read-only input tape*
- An infinite *wright-only output tape*
- A *program*, which contains a sequence of *instrutions*
- A *memory*, which contains a sequence of *registers*



Read-only input tape:

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).
- Whenever a symbol is read from the input tape, the tape head moves one square to the right.

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).
- Whenever a symbol is read from the input tape, the tape head moves one square to the right.

Wright-only output tape:

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).
- Whenever a symbol is read from the input tape, the tape head moves one square to the right.

Wright-only output tape:

- Is a sequence of squares, each of which is initially blank.

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).
- Whenever a symbol is read from the input tape, the tape head moves one square to the right.

Wright-only output tape:

- Is a sequence of squares, each of which is initially blank.
- When a write instruction is executed, an integer is printed in the square of the output tape that is currently under the output tape head and the tape head is moved one square to the right.

Read-only input tape:

- Is a sequence of squares, each of which holds an integer (possibly negative).
- Whenever a symbol is read from the input tape, the tape head moves one square to the right.

Wright-only output tape:

- Is a sequence of squares, each of which is initially blank.
- When a write instruction is executed, an integer is printed in the square of the output tape that is currently under the output tape head and the tape head is moved one square to the right.
- Once an output symbol has been written. it cannot be changed.

Memory:

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).
- Each register is capable of holding an integer of arbitrary size.

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).
- Each register is capable of holding an integer of arbitrary size.
- Register R_0 is called *accumulator* and all computation takes place in it.

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).
- Each register is capable of holding an integer of arbitrary size.
- Register R_0 is called *accumulator* and all computation takes place in it.

Program:

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).
- Each register is capable of holding an integer of arbitrary size.
- Register R_0 is called *accumulator* and all computation takes place in it.

Program:

- Is a sequence of labeled instructions.

Memory:

- Consists of a sequence of registers, $R_0, R_1, ..., R_i, ...$ (we place no upper bound on the number of registers that can be used).
- We have random access to each register (indirect addresing).
- Each register is capable of holding an integer of arbitrary size.
- Register R_0 is called *accumulator* and all computation takes place in it.

Program:

- Is a sequence of labeled instructions.
- Does not modify itself.

Instructions:

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.
- Each instruction consists of two parts:

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.
- Each instruction consists of two parts:
  - The *operation code*

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.
- Each instruction consists of two parts:
  - The *operation code*
  - The *address*:

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.
- Each instruction consists of two parts:
  - The *operation code*
  - The *address*: operand / label

Instructions:

- arithmetic, input-output, indirect addressing, branching instructions e.t.c.
- Each instruction consists of two parts:
  - The *operation code*
  - The *address*: operand / label

Operation code	Address
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JUMP	label
10. JGTZ	label
11. JZERO	label
12. HALT	

Example of basic instructions

We can define the meaning of a program P at each step with the help of two quantities:

We can define the meaning of a program P at each step with the help of two quantities:

- The *memory map* $c : \mathbb{N} \to \mathbb{Z}$, where $c(i)$ is the contents of the register R_i.

We can define the meaning of a program P at each step with the help of two quantities:

- The *memory map* $c : \mathbb{N} \to \mathbb{Z}$, where $c(i)$ is the contents of the register R_i.

  Initially, $\forall i \in \mathbb{N} \quad c(i) = 0$.

We can define the meaning of a program P at each step with the help of two quantities:

- The *memory map* $c : \mathbb{N} \to \mathbb{Z}$, where $c(i)$ is the contents of the register R_i.

  Initially, $\forall i \in \mathbb{N} \quad c(i) = 0$.
- The *location counter*, which determines the next instruction to execute.

We can define the meaning of a program P at each step with the help of two quantities:

- The *memory map* $c : \mathbb{N} \to \mathbb{Z}$, where $c(i)$ is the contents of the register R_i.

  Initially, $\forall i \in \mathbb{N} \quad c(i) = 0$.

- The *location counter*, which determines the next instruction to execute.

  Initially, the location counter is set to the first instruction in P.

We can define the meaning of a program P at each step with the help of two quantities:

- The *memory map* $c : \mathbb{N} \to \mathbb{Z}$, where $c(i)$ is the contents of the register R_i.

  Initially, $\forall i \in \mathbb{N} \quad c(i) = 0$.

- The *location counter*, which determines the next instruction to execute.

  Initially, the location counter is set to the first instruction in P.

  After execution of the kth instruction in P, the location counter is automatically set to $k + 1$ (i.e. the next instruction), unless the kth instruction is **JUMP**, **HALT**, **JGTZ**. or **JZERO**.

Operand:

Operand:

1 $= i$: The integer i itself.

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$
3. $*i$: The contents of register R_j, where j is the content of register R_i ($i \in \mathbb{N}$) (*indirect addressing*). If $j < 0$ the machine halts.

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$
3. $*i$: The contents of register R_j, where j is the content of register R_i ($i \in \mathbb{N}$) (*indirect addressing*). If $j < 0$ the machine halts.

To specify the meaning of an instruction we define $v(a)$, the *the value of operand* a, as follows:

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$
3. $*i$: The contents of register R_j, where j is the content of register R_i ($i \in \mathbb{N}$) (*indirect addressing*). If $j < 0$ the machine halts.

To specify the meaning of an instruction we define $v(a)$, the *the value of operand* a, as follows:

$$v\,(= i) = i$$

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$
3. $*i$: The contents of register R_j, where j is the content of register R_i ($i \in \mathbb{N}$) (*indirect addressing*). If $j < 0$ the machine halts.

To specify the meaning of an instruction we define $v(a)$, the *the value of operand* a, as follows:

$$v\,(= i) = i$$

$$v\,(i) = c\,(i)$$

Operand:

1. $= i$: The integer i itself.
2. i : The contents of register R_i, where $i \in \mathbb{N}$
3. $*i$: The contents of register R_j, where j is the content of register R_i ($i \in \mathbb{N}$) (*indirect addressing*). If $j < 0$ the machine halts.

To specify the meaning of an instruction we define $v(a)$, the *the value of operand* a, as follows:

$$v(= i) = i$$

$$v(i) = c(i)$$

$$v(*i) = c(c(i))$$

Instruction	Meaning
1. LOAD a	$c(0) \leftarrow v(a)$
2. STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
3. ADD a	$c(0) \leftarrow c(0) + v(a)$
4. SUB a	$c(0) \leftarrow c(0) - v(a)$
5. MULT a	$c(0) \leftarrow c(0) \times v(a)$
6. DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$†
7. READ i	$c(i) \leftarrow$ current input symbol.
READ $*i$	$c(c(i)) \leftarrow$ current input symbol. The input tape head moves one square right in either case.
8. WRITE a	$v(a)$ is printed on the square of the output tape currently under the output tape head. Then the tape head is moved one square right.
9. JUMP b	The location counter is set to the instruction labeled b.
10. JGTZ b	The location counter is set to the instruction labeled b if $c(0) > 0$; otherwise, the location counter is set to the next instruction.
11. JZERO b	The location counter is set to the instruction labeled b if $c(0) = 0$; otherwise, the location counter is set to the next instruction.
12. HALT	Execution ceases.

The meaning of basic instructions

What does a RAM do?

- A RAM computes functions:
  - A RAM can compute exactly the partial recursive functions.

What does a RAM do?

- A RAM computes functions:
  - A RAM can compute exactly the partial recursive functions.
- A RAM accepts languages:
  - A RAM accepts exactly the recursively enumerable languages.

What does a RAM do?

- A RAM computes functions:
  - A RAM can compute exactly the partial recursive functions.
- A RAM accepts languages:
  - A RAM accepts exactly the recursively enumerable languages.

Thus a RAM is a reasonable model of a computer.

Example A RAM program computing the function $f : \mathbb{Z} \to \mathbb{Z}$

$$f(n) = \begin{cases} n^n & , n \in \mathbb{N}_{>0} \\ 0 & , \text{otherwise} \end{cases}$$

Example A RAM program computing the function $f : \mathbb{Z} \to \mathbb{Z}$

$$f(n) = \begin{cases} n^n & , n \in \mathbb{N}_{>0} \\ 0 & , \text{otherwise} \end{cases}$$

```
begin
    read r1;
    if r1 ≤ 0 then write 0
    else
        begin
            r2 ← r1;
            r3 ← r1 − 1;
            while r3 > 0 do
                begin
                    r2 ← r2 * r1;
                    r3 ← r3 − 1
                end;
            write r2
        end
end
```

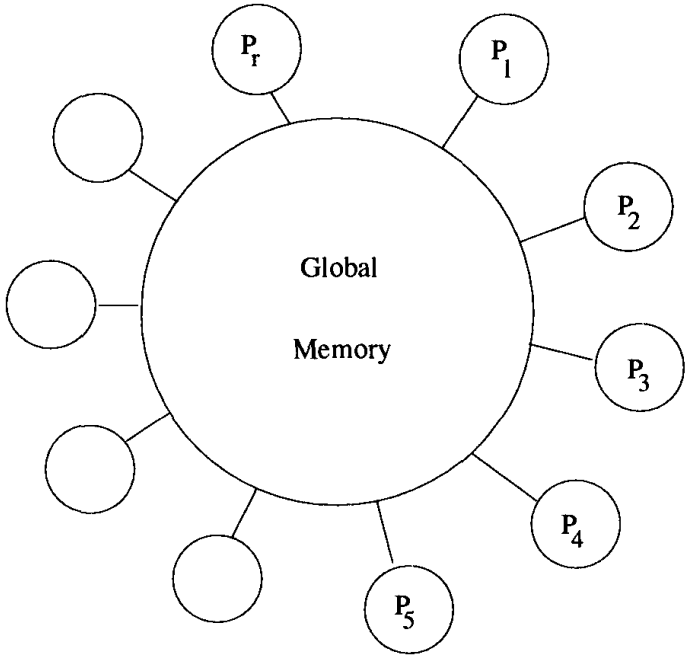
An algorithm for f

Solution

	RAM program		Corresponding Pidgin ALGOL statements
	READ	1	**read** $r1$
	LOAD	1	
	JGTZ	pos	**if** $r1 \leq 0$ **then write** 0
	WRITE	=0	
	JUMP	endif	
pos:	LOAD	1	$r2 \leftarrow r1$
	STORE	2	
	LOAD	1	
	SUB	=1	$r3 \leftarrow r1 - 1$
	STORE	3	
while:	LOAD	3	
	JGTZ	continue	**while** $r3 > 0$ **do**
	JUMP	endwhile	
continue:	LOAD	2	
	MULT	1	$r2 \leftarrow r2 * r1$
	STORE	2	
	LOAD	3	
	SUB	=1	$r3 \leftarrow r3 - 1$
	STORE	3	
	JUMP	while	
endwhile:	WRITE	2	**write** $r2$
endif:	HALT		

CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

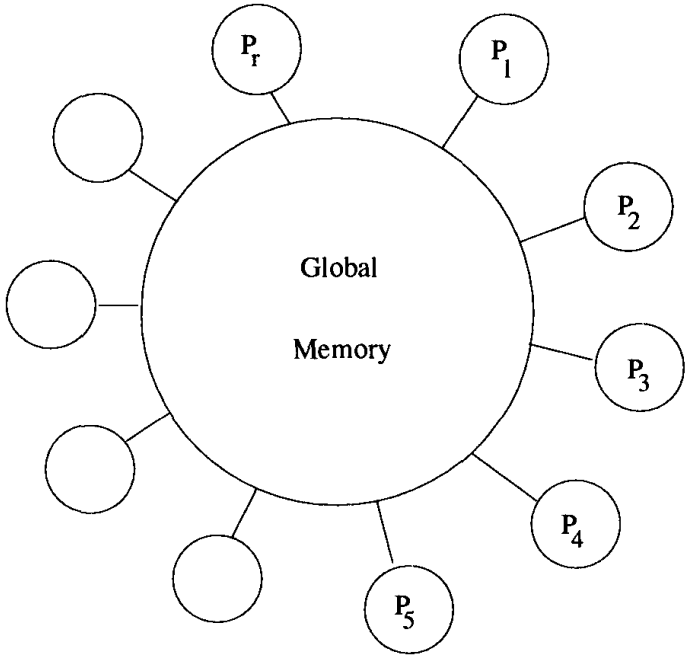
A *parallel random-access machine* (PRAM) is a shared-memory abstract machine.



CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

A *parallel random-access machine* (PRAM) is a shared-memory abstract machine.

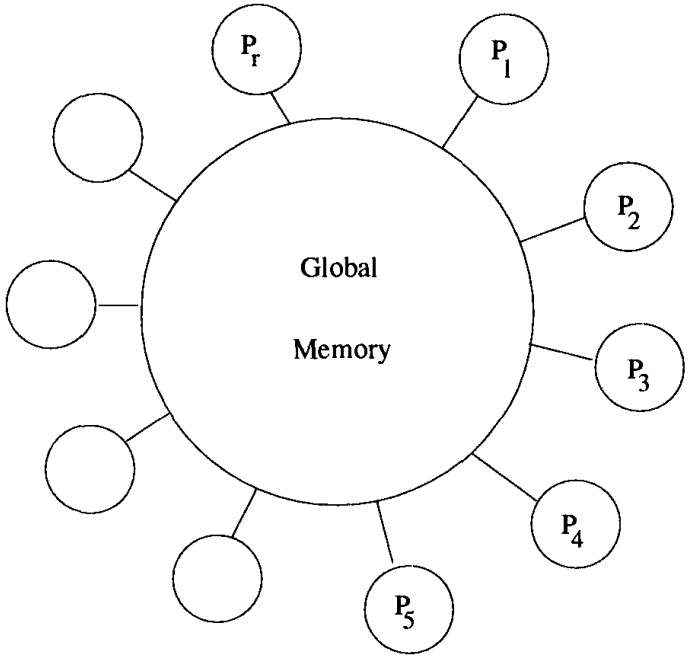
- Is the parallel-computing analogy to the RAM.



CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

A *parallel random-access machine* (PRAM) is a shared-memory abstract machine.

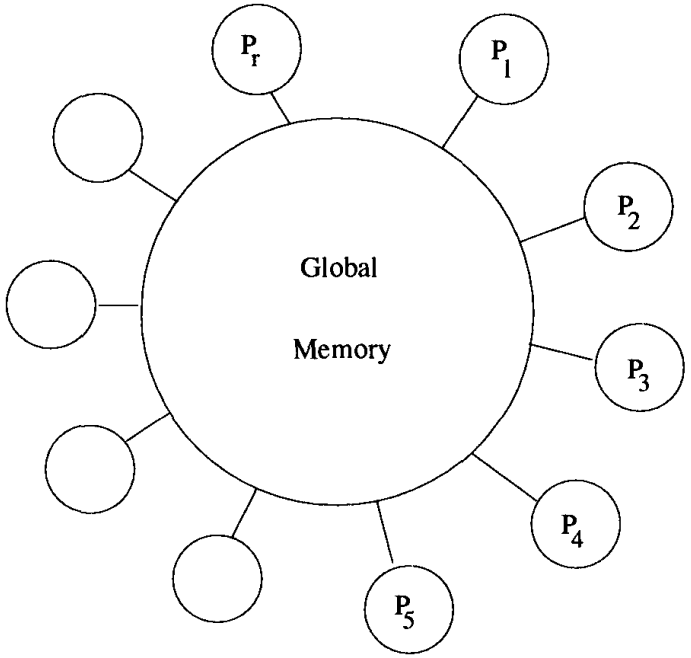
- Is the parallel-computing analogy to the RAM.
- It consists of a sequence of RAM's $(P_i)_{[r]}$, without input and output tape (each RAM uses the Global Memory).



CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

A *parallel random-access machine* (PRAM) is a shared-memory abstract machine.

- Is the parallel-computing analogy to the RAM.
- It consists of a sequence of RAM's $(P_i)_{[r]}$, without input and output tape (each RAM uses the Global Memory).
- It is *synchronous* (the processors (i.e. RAM's) work in lock step).



Categorization accordinng to read/write conflicts.

1. Exclusive read exclusive write (*EREW*): every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (*CREW*): multiple processors can read a memory cell but only one can write at a time
3. Concurrent read concurrent write (*CRCW*): multiple processors can read and write.

Categorization accordinng to read/write conflicts.

1. Exclusive read exclusive write (*EREW*): every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (*CREW*): multiple processors can read a memory cell but only one can write at a time
3. Concurrent read concurrent write (*CRCW*): multiple processors can read and write.

Categorization of CRCW PRAM's

1. *Common*: all processors write the same value; otherwise is illegal
2. *Arbitrary*: only one arbitrary attempt is successful, others retire
3. *Priority*: processor rank indicates who gets to write

Definition of CRAM

CRAM is a special type of Priority CRCW-PRAM.
Each RAM has a finite set of registers, including the following:

- *Processor*: containing the number between 1 and p(n) of the RAM
- *Address*: containing an address of global memory
- *Contents*: containing a word to be written or read from global memory
- *ProgramCounter*: containing the line number of the instruction to be executed next.

RAMs are identical except the Processor number.

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.
- **WRITE**: Write the *Contents* register into the Global Memory location specified by *Address*.

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.
- **WRITE**: Write the *Contents* register into the Global Memory location specified by *Address*.
- **OP** R_a R_b: Perform **OP** on R_a and R_b and leave the result in R_b. Here **OP** may be **Add**, **Subtract**, or, **Shift**.

♩ **Shift**(x, y) causes the word x to be shifted y bits to the right.

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.
- **WRITE**: Write the *Contents* register into the Global Memory location specified by *Address*.
- **OP** R_a R_b: Perform **OP** on R_a and R_b and leave the result in R_b. Here **OP** may be **Add**, **Subtract**, or, **Shift**.
- **MOVE** R_a R_b: Move R_a to R_b

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.
- **WRITE**: Write the *Contents* register into the Global Memory location specified by *Address*.
- **OP** R_a R_b: Perform **OP** on R_a and R_b and leave the result in R_b. Here **OP** may be **Add**, **Subtract**, or, **Shift**.
- **MOVE** R_a R_b: Move R_a to R_b
- **BLT** R L: Branch to line (adress) L of the Program, if the contents of R is less than zero.

Definition of CRAM

The instructions of a CRAM consist of the following:

- **READ**: Read the word of Global Memory specified by *Address* into *Contents*.
- **WRITE**: Write the *Contents* register into the Global Memory location specified by *Address*.
- **OP** R_a R_b: Perform **OP** on R_a and R_b and leave the result in R_b. Here **OP** may be **Add**, **Subtract**, or, **Shift**.
- **MOVE** R_a R_b: Move R_a to R_b
- **BLT** R L: Branch to line (adress) L of the Program, if the contents of R is less than zero.

♩ The above instructions each increment the ProgramCounter, with the exception of **BLT**.

CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

- The Shift operation for the CRAM allows each bit of Global Memory to be available to every processor in constant time.

CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

- The Shift operation for the CRAM allows each bit of Global Memory to be available to every processor in constant time.
- We assume initially that the contents of the first $|\mathrm{bin}(\mathcal{A})|$ words of Global Memory contain one bit each of the input string $\mathrm{bin}(\mathcal{A})$.

- The Shift operation for the CRAM allows each bit of Global Memory to be available to every processor in constant time.
- We assume initially that the contents of the first $|\mathrm{bin}(\mathcal{A})|$ words of Global Memory contain one bit each of the input string $\mathrm{bin}(\mathcal{A})$.
- We assume that a section of Global Memory is specified as the output.

CRAM's complexity

Definitions

$\mathrm{CRAM}[t(n)]$: The set of boolean queries computable in parallel time $t(n)$ on a CRAM that has at most polynomially many processors.

CRAM's complexity

Definitions

$\mathrm{CRAM}[t(n)]$: The set of boolean queries computable in parallel time $t(n)$ on a CRAM that has at most polynomially many processors.
$\mathrm{CRAM{-}PROC}[t(n), p(n)]$: The set of boolean queries computable by a CRAM using at most $p(n)$ processors and time $\mathcal{O}(t(n))$.

CRAM's complexity

Definitions

$\mathrm{CRAM}[t(n)]$: The set of boolean queries computable in parallel time $t(n)$ on a CRAM that has at most polynomially many processors.
$\mathrm{CRAM{-}PROC}[t(n), p(n)]$: The set of boolean queries computable by a CRAM using at most $p(n)$ processors and time $\mathcal{O}(t(n))$.

Thus,

$$\mathrm{CRAM}[t(n)] = \mathrm{CRAM{-}PROC}[t(n), n^{\mathcal{O}(1)}]$$

Definitions

Let $\varphi(R, \overrightarrow{x})$ be an R-positive formula, where R is a relation symbol of arity k, and let $\mathcal{A}$ be a structure of size n. Define the depth of φ in $\mathcal{A}$, in symbols $|\varphi^{\mathcal{A}}|$, to be the minimum r such that

$$\mathcal{A} \models \left(\varphi^{r}(\emptyset) \leftrightarrow \varphi^{r+1}(\emptyset)\right)$$

Define the depth of φ as a function of n equal to the maximum depth of φ in $\mathcal{A}$ for any structure $\mathcal{A}$ of size n:

$$|\varphi|(n) \doteqdot \max_{\|\mathcal{A}\|=n} \left\{\left|\varphi^{\mathcal{A}}\right|\right\}$$

$\mathrm{IND}[f(n)]$ be the sublanguage of $\mathrm{FO(LFP)}$ in which only fixed points of first-order formulas φ for which $|\varphi|$ is $\mathcal{O}[f(n)]$ are included.

Iterating FO formulas

Moschovakis' Canonical Form for Positive Formulas

Lemma

Let φ be an R-positive first-order formula and $\overrightarrow{x} = (x_1, ..., x_k)$. Then φ can be written in the following form,

$$\varphi(R, \overrightarrow{x}) \equiv (Q_1 z_1 . M_1) \dots (Q_s z_s . M_s) (\exists x_1 \dots x_k . M_{s+1}) R(x_1, ..., x_k)$$

where the M_i's are quantifier-free formulas in which R does not occur.

Let $\mathrm{QB} \doteqdot [(Q_1 z_1 . M_1) \ldots (Q_s z_s . M_s)\,(\exists x_1 \ldots x_k . M_{s+1})]$. Then $\forall \mathcal{A}$ structure and $\forall r \in \mathbb{N}$

$$\mathcal{A} \models \left(\left(\varphi^{\mathcal{A}}\right)^r (\emptyset) \leftrightarrow ([\mathrm{QB}]^r \textbf{ false }) \right)$$

Let $\mathrm{QB} \doteqdot [(Q_1 z_1 . M_1) \ldots (Q_s z_s . M_s)\,(\exists x_1 \ldots x_k . M_{s+1})]$. Then $\forall \mathcal{A}$ structure and $\forall r \in \mathbb{N}$

$$\mathcal{A} \models \left(\left(\varphi^{\mathcal{A}}\right)^r (\emptyset) \leftrightarrow ([\mathrm{QB}]^r \textbf{ false }) \right)$$

Thus if $t = |\varphi|(n)$ and $\mathcal{A}$ is any structure of size n then

$$\mathcal{A} \models ((\mathrm{LFP}\varphi) \leftrightarrow ([\mathrm{QB}]^t \textbf{ false }))$$

Definition

A set $S \subseteq \mathrm{STRUC}[\tau]$ is a member of $\mathrm{FO}[t(n)]$ iff there exist quantifier free formulas M_i , $i \in [s]$, from $\mathcal{L}(\tau)$, a tuple $\overrightarrow{c}$ of constants and a quantifier block,

$$\mathrm{QB} = [(Q_1 z_1 . M_1) \ldots (Q_s z_s . M_s)]$$

such that $\forall \mathcal{A} \in \mathrm{STRUC}[\tau]$,

$$\mathcal{A} \in S \Leftrightarrow \mathcal{A} \models \left([\mathrm{QB}]^{t(\|\mathcal{A}\|)} M_0 \right) (\overrightarrow{c} / \overrightarrow{x})$$

Example

Let's recall the alternative inductive definition of the reflexive transitive closure, E^*, of E, that we saw in the previous lecture:

$$\varphi^* (R, x, y) \equiv x = y \vee E(x, y) \vee \exists z (R(x, z) \wedge R(z, y))$$

with depth

$$|\varphi^*| (n) = \lceil \log n \rceil + 1$$

We want to find out how to wright this inductive definition in the Moschovakis' Canonical Form for Positive Formulas.

Solution

First, code the base case using a dummy universal quantification:

$$\varphi^* (R, x, y) \equiv (\forall z. M_1) (\exists z) (R (x, z) \wedge R (z, y))$$

$$M_1 \equiv x = y \vee E (x, y)$$

Solution

First, code the base case using a dummy universal quantification:

$$\varphi^* (R, x, y) \equiv (\forall z. M_1) (\exists z) (R (x, z) \wedge R (z, y))$$

$$M_1 \equiv x = y \vee E (x, y)$$

Next, use universal quantification to replace the two occurrences of R with a single one:

$$\varphi^* (R, x, y) \equiv (\forall z. M_1) (\exists z) (\forall uv. M_2) (R (u, v))$$

$$M_2 \equiv (u = x \wedge v = z) \vee (u = z \wedge v = y)$$

Solution

Finally, requantify x and y:

$$\varphi^* (R, x, y) \equiv (\forall z.M_1)(\exists z)(\forall uv.M_2)(\exists xy.M_3)\, R(x, y)$$

$$M_3 \equiv (x = u \wedge v = y)$$

□

Define the quantifier block:

$$\mathrm{QB}^* \equiv (\forall z.M_1)\,(\exists z)\,(\forall uv.M_2)\,(\exists xy.M_3)$$

Define the quantifier block:

$$\mathrm{QB}^* \equiv (\forall z.M_1)\,(\exists z)\,(\forall uv.M_2)\,(\exists xy.M_3)$$

Thus $\forall r \in \mathbb{N}$:

$$\varphi^{*^r}(\emptyset) \equiv [\mathrm{QB}^*]^r\,(\textbf{false})$$

Define the quantifier block:

$$\mathrm{QB}^* \equiv (\forall z.M_1)\,(\exists z)\,(\forall uv.M_2)\,(\exists xy.M_3)$$

Thus $\forall r \in \mathbb{N}$:

$$\varphi^{*^r}(\emptyset) \equiv [\mathrm{QB}^*]^r\,(\textbf{false})$$

The boolean query REACH is expressible as:

$$\mathrm{REACH} \equiv \left(\mathrm{LFP}_{R_{xy}}\varphi^*\right)(s,t)$$

Define the quantifier block:

$$\mathrm{QB}^* \equiv (\forall z.M_1)\,(\exists z)\,(\forall uv.M_2)\,(\exists xy.M_3)$$

Thus $\forall r \in \mathbb{N}$:

$$\varphi^{*^r}(\emptyset) \equiv [\mathrm{QB}^*]^r\,(\textbf{false})$$

The boolean query REACH is expressible as:

$$\mathrm{REACH} \equiv \left(\mathrm{LFP}_{R_{xy}}\varphi^*\right)(s,t)$$

Thus by previous example we have that

$$\mathrm{REACH} \in \mathrm{FO}[\log n]$$

We are ready to prove the main Theorem of this Section.

We are ready to prove the main Theorem of this Section.

Theorem

Let S be a boolean query. For all polynomially bounded, parallel time constructible $t(n)$, the following are equivalent:

1. *S is computable by a CRAM in parallel time $t(n)$ using polynomially many processors and registers of polynomially bounded word size.*
2. *S is definable as a uniform first-order induction whose depth, for structures of size n, is at most $t(n)$.*
3. *There exists a first-order quantifier-block $[\mathrm{QB}]$, a quantifier-free formula M_0 and a tuple $\overrightarrow{c}$ of constants such that the query S for structures of size at most n is expressed as $[\mathrm{QB}]^{t(n)} M_0(\overrightarrow{c}/\overrightarrow{x})$.*

In symbols, the previous theorem can be stated as:

$$\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)]$$

In symbols, the previous theorem can be stated as:

$$\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)]$$

In order to prove this theorem we will need 3 Lemmas.

Lemma

For all $t(n)$ and all classes of finite structures,

$$\mathrm{IND}\left[t\left(n\right)\right] \subseteq \mathrm{FO}\left[t\left(n\right)\right]$$

Lemma

For all $t(n)$ and all classes of finite structures,

$$\mathrm{IND}\left[t\left(n\right)\right] \subseteq \mathrm{FO}\left[t\left(n\right)\right]$$

Proof.

Hint: Previous lemma and straight forward from definitions. □

Lemma

For any polynomially bounded $t(n)$ we have,

$$\mathrm{CRAM}\left[t\left(n\right)\right] \subseteq \mathrm{IND}\left[t\left(n\right)\right]$$

Lemma

For any polynomially bounded $t(n)$ we have,

$$\mathrm{CRAM}\left[t\left(n\right)\right] \subseteq \mathrm{IND}\left[t\left(n\right)\right]$$

Proof.

Sketching of solution: We want to simulate the computation of a CRAM M, on input $\mathcal{A}: \ \|\mathcal{A}\| = n$, by defining the contents of all the relevant registers for any processor of M by induction on the time step, through a relation $\mathrm{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$, meaning that bit $\overline{x}$ in register r of processor p just after step t is equal to b.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.
- For $\overline{t} = 0$ the memory is correctly loaded with $\mathrm{bin}(\mathcal{A})$.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.
- For $\overline{t} = 0$ the memory is correctly loaded with $\mathrm{bin}(\mathcal{A})$.
- The inductive definition of the relation $\mathrm{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of p's ProgramCounter at time $\overline{t} - 1$.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.
- For $\overline{t} = 0$ the memory is correctly loaded with $\mathrm{bin}(\mathcal{A})$.
- The inductive definition of the relation $\mathrm{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of p's ProgramCounter at time $\overline{t} - 1$.
- **Addition**, **Subtraction**, **BLT** are first-order expressible.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.
- For $\overline{t} = 0$ the memory is correctly loaded with $\mathrm{bin}(\mathcal{A})$.
- The inductive definition of the relation $\mathrm{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of p's ProgramCounter at time $\overline{t} - 1$.
- **Addition**, **Subtraction**, **BLT** are first-order expressible.
- **Shift** is first-order expressible due to relation BIT.

Proof.

- We need constant number of variables $x_1, ..., x_k$ each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation.
- For $\overline{t} = 0$ the memory is correctly loaded with $\mathrm{bin}(\mathcal{A})$.
- The inductive definition of the relation $\mathrm{VALUE}(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of p's ProgramCounter at time $\overline{t} - 1$.
- **Addition**, **Subtraction**, **BLT** are first-order expressible.
- **Shift** is first-order expressible due to relation BIT.

□

Thus we describe an inductive definition of relation VALUE, coding M's entire computation.

Lemma

For polynomially bounded and parallel time constructible $t(n)$,

$$\mathrm{FO}\left[t\left(n\right)\right] \subseteq \mathrm{CRAM}\left[t\left(n\right)\right]$$

Lemma

For polynomially bounded and parallel time constructible t(n),

$$\mathrm{FO}\left[t\left(n\right)\right] \subseteq \mathrm{CRAM}\left[t\left(n\right)\right]$$

Proof.

Let the $\mathrm{FO}[t(n)]$ problem be determined by the following quantifier free formulas, quantifier block, and tuple of constants,

$$M_0, ..., M_k; \quad \mathrm{QB} = \left(Q_1 x_1 . M_1\right) ... \left(Q_k x_k . M_k\right); \quad \overrightarrow{c}$$

Our CRAM must test whether an input structure $\mathcal{A}$, so that $\|\mathcal{A}\| = n$ satisfies the sentence,

$$\varphi_n \equiv [\mathrm{QB}]^{t(n)} \, M_0 \left(\overrightarrow{c} / \overrightarrow{x}\right)$$

Proof.

The CRAM will:

- use n^k processors (RAMs)

Proof.

The CRAM will:

- use n^k processors (RAMs)
  - Each processor will have a number $a_1 ... a_k$, where $a_i \in \{0, ..., n-1\} \doteqdot \mathtt{n}$
  - Using the **Shift** operation it can retrieve each of the a_i's in constant time.

Proof.

The CRAM will:

- use n^k processors (RAMs)
  - Each processor will have a number $a_1 ... a_k$, where $a_i \in \{0, ..., n-1\} \doteqdot \mathtt{n}$
  - Using the **Shift** operation it can retrieve each of the a_i's in constant time.
- use n^{k-1} bits of Global Memory

Proof.

The CRAM will:

- use n^k processors (RAMs)
  - Each processor will have a number $a_1 ... a_k$, where $a_i \in \{0, ..., n-1\} \doteqdot \mathrm{n}$
  - Using the **Shift** operation it can retrieve each of the a_i's in constant time.
- use n^{k-1} bits of Global Memory
- evaluate φ_n from right to left, simultaneously for all values of the variables $x_1, ..., x_k$.

Proof.

We will denote for $q \in \mathtt{t}(\mathtt{n})$, $i \in [k]$ and $r = k \cdot (q+1) + 1 - i$

$$\varphi^r \equiv (Q_i x_i . M_i) \ldots (Q_k x_k . M_k) \, [\mathrm{QB}]^q \, M_0$$

Proof.

We will denote for $q \in \mathtt{t}(\mathtt{n})$, $i \in [k]$ and $r = k \cdot (q+1) + 1 - i$

$$\varphi^r \equiv (Q_i x_i . M_i) \dots (Q_k x_k . M_k) \, [\mathrm{QB}]^q \, M_0$$

That is

$$\varphi^1 \equiv (Q_k x_k . M_k) \, M_0, \quad \varphi^2 \equiv (Q_{k-1} x_{k-1} . M_{k-1}) \, (Q_k x_k . M_k) \, M_0, \dots$$

Proof.

We will denote for $q \in \mathtt{t}(\mathtt{n})$, $i \in [k]$ and $r = k \cdot (q+1) + 1 - i$

$$\varphi^r \equiv (Q_i x_i . M_i) \dots (Q_k x_k . M_k) \, [\mathrm{QB}]^q \, M_0$$

That is

$$\varphi^1 \equiv (Q_k x_k . M_k) \, M_0, \quad \varphi^2 \equiv (Q_{k-1} x_{k-1} . M_{k-1}) \, (Q_k x_k . M_k) \, M_0, \dots$$

$$\varphi^k \equiv [\mathrm{QB}] \, M_0, \quad \varphi^{k+1} \equiv (Q_k x_k . M_k) \, [\mathrm{QB}] \, M_0, \dots$$

Proof.

We will denote for $q \in \mathtt{t}(\mathtt{n})$, $i \in [k]$ and $r = k \cdot (q+1) + 1 - i$

$$\varphi^r \equiv (Q_i x_i . M_i) \dots (Q_k x_k . M_k) \, [\mathrm{QB}]^q \, M_0$$

That is

$$\varphi^1 \equiv (Q_k x_k . M_k) \, M_0, \quad \varphi^2 \equiv (Q_{k-1} x_{k-1} . M_{k-1}) \, (Q_k x_k . M_k) \, M_0, \dots$$

$$\varphi^k \equiv [\mathrm{QB}] \, M_0, \quad \varphi^{k+1} \equiv (Q_k x_k . M_k) \, [\mathrm{QB}] \, M_0, \dots$$

$$\varphi^{t(n)k} \equiv [\mathrm{QB}]^{t(n)} \, M_0$$

Proof.

We will denote for $q \in \mathtt{t}(\mathtt{n})$, $i \in [k]$ and $r = k \cdot (q+1) + 1 - i$

$$\varphi^r \equiv (Q_i x_i . M_i) \dots (Q_k x_k . M_k) \, [\mathrm{QB}]^q \, M_0$$

That is

$$\varphi^1 \equiv (Q_k x_k . M_k) \, M_0, \quad \varphi^2 \equiv (Q_{k-1} x_{k-1} . M_{k-1}) \, (Q_k x_k . M_k) \, M_0, \dots$$

$$\varphi^k \equiv [\mathrm{QB}] \, M_0, \quad \varphi^{k+1} \equiv (Q_k x_k . M_k) \, [\mathrm{QB}] \, M_0, \dots$$

$$\varphi^{t(n)k} \equiv [\mathrm{QB}]^{t(n)} \, M_0$$

We will denote with $x_1 \dots \hat{x}_i \dots x_k$ the $k-1$-tuple resulting from $x_1 \dots x_k$ by removing x_i.

Proof.

We will now give a program for the CRAM which is broken into rounds each consisting of three processor steps such that: Just after round r, the contents of memory location $a_1 ... \hat{a}_i ... a_k$ is 1 or 0 according as whether $\mathcal{A} \models \varphi^r(a_1, ..., a_k)$ or not (Each processor $a_1 \ldots a_k$, at step $r+1$ sets $b := 1$ iff $\mathcal{A} \models \varphi^r$.).

Proof.

We will now give a program for the CRAM which is broken into rounds each consisting of three processor steps such that: Just after round r, the contents of memory location $a_1 ... \hat{a}_i ... a_k$ is 1 or 0 according as whether $\mathcal{A} \models \varphi^r(a_1, ..., a_k)$ or not (Each processor $a_1 \ldots a_k$, at step $r+1$ sets $b := 1$ iff $\mathcal{A} \models \varphi^r$.).

Base case:

At step 1, processor $a_1 \ldots a_k$ must set:

$$b = 1 \iff \mathcal{A} \models M_0(a_1, ..., a_k)$$

Proof.

Inductive step:

At round r, processor number $a_1 \ldots a_k$ executes the following three instructions according to whether Q_i is $\exists$ or Q_i is $\forall$:

Proof.

Inductive step:

At round r, processor number $a_1 \ldots a_k$ executes the following three instructions according to whether Q_i is $\exists$ or Q_i is $\forall$:

Q_i is $\exists$

1. $b := loc(a_1 \ldots a_i \hat{}_{+1} \ldots a_k)$;
2. $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 0$;
3. If $M_i(a_1, \ldots, a_k)$ and b then $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 1$;

Proof.

Inductive step:

At round r, processor number $a_1 \ldots a_k$ executes the following three instructions according to whether Q_i is $\exists$ or Q_i is $\forall$:

Q_i is $\exists$

1. $b := loc(a_1 \ldots a_i\hat{}_{+1} \ldots a_k)$;
2. $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 0$;
3. If $M_i(a_1, \ldots, a_k)$ and b then $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 1$;

Q_i is $\forall$

1. $b := loc(a_1 \ldots a_i\hat{}_{+1} \ldots a_k)$;
2. $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 1$;
3. If $M_i(a_1, \ldots, a_k)$ and $\neg b$ then $loc(a_1 \ldots \hat{a}_i \ldots a_k) := 0$;

$\square$

From the three previous lemmas we prove the requested Theorem.

CRAM$[t(n)]$ =IND$[t(n)]$ = FO$[t(n)]$

From the three previous lemmas we prove the requested Theorem.

Definition

A **boolean circuit** is a directed acyclic graph (DAG)

$$C = (V, E, G_\wedge, G_\vee, G_\neg, I, r)$$

where $\tau_c \doteqdot \langle E^2, G^1_\wedge, G^1_\vee, G^1_\neg, I^1, r \rangle$ (vocabulery of circuits).
An internal node w is:

- an and-gate iff $G_\wedge$ holds
- an or-gate iff $G_\vee$ holds
- an not-gate iff $G_\neg$ holds
- called a leaf iff it has no incoming edges and leaf w is on iff $I(w)$ holds

Define **Circuit Value Problem** (CVP) to consist of those circuits the root gate of which evaluate to one.

Let $\mathcal{A} \in \mathrm{STRUC}[\tau]$ and $\|\mathcal{A}\| = n$. A circuit C_n, with $\hat{n}_\tau(n) \doteqdot \|\mathrm{bin}_\tau(\mathcal{A})\|$ leaves, can take $\mathcal{A}$ as input by placing the binary string $\mathrm{bin}_\tau(\mathcal{A})$ into its leaves.

Let $\mathcal{A} \in \mathrm{STRUC}[\tau]$ and $\|\mathcal{A}\| = n$. A circuit C_n, with $\hat{n}_\tau(n) \doteqdot \|\mathrm{bin}_\tau(\mathcal{A})\|$ leaves, can take $\mathcal{A}$ as input by placing the binary string $\mathrm{bin}_\tau(\mathcal{A})$ into its leaves.

We write $C(w)$ to denote the output of circuit C on input w, i.e., the value of the root node r when w is placed at the leaves and C is then evaluated.

Let $\mathcal{A} \in \mathrm{STRUC}[\tau]$ and $\|\mathcal{A}\| = n$. A circuit C_n, with $\hat{n}_\tau(n) \doteqdot \|\mathrm{bin}_\tau(\mathcal{A})\|$ leaves, can take $\mathcal{A}$ as input by placing the binary string $\mathrm{bin}_\tau(\mathcal{A})$ into its leaves.

We write $C(w)$ to denote the output of circuit C on input w, i.e., the value of the root node r when w is placed at the leaves and C is then evaluated.

We say that circuit C *accepts* structure $\mathcal{A}$ iff $C(\mathrm{bin}_\tau(\mathcal{A})) = 1$.

Let $S \subseteq \mathrm{STRUC}[\tau_s]$ be a boolean query on binary strings. Let $\mathcal{C} = \{C_i\}_{\mathbb{N}_{\geq 1}}$ an infinite sequence of circuits, where C_n is a circuit with n input bits.
We say that $\mathcal{C}$ computes S iff for all $n \in \mathbb{N}_{\geq 1}$ and for all $w \in \{0,1\}^n$,

$$w \in S \iff C_n(w) = 1$$

Let $S \subseteq \mathrm{STRUC}[\tau_s]$ be a boolean query on binary strings. Let $\mathcal{C} = \{C_i\}_{\mathbb{N}_{\geq 1}}$ an infinite sequence of circuits, where C_n is a circuit with n input bits.
We say that $\mathcal{C}$ computes S iff for all $n \in \mathbb{N}_{\geq 1}$ and for all $w \in \{0,1\}^n$,

$$w \in S \iff C_n(w) = 1$$

A **threshold gate** with threshold value i has output one iff at least i of its inputs have value one.

We generalize the vocabulary of circuits to the *vocabulary of threshold circuits*, $\tau_{thc} \doteqdot \tau_c \cup \{G_t^2\}$, where $G_t(g,k)$ means that g is a threshold gate with threshold value k.

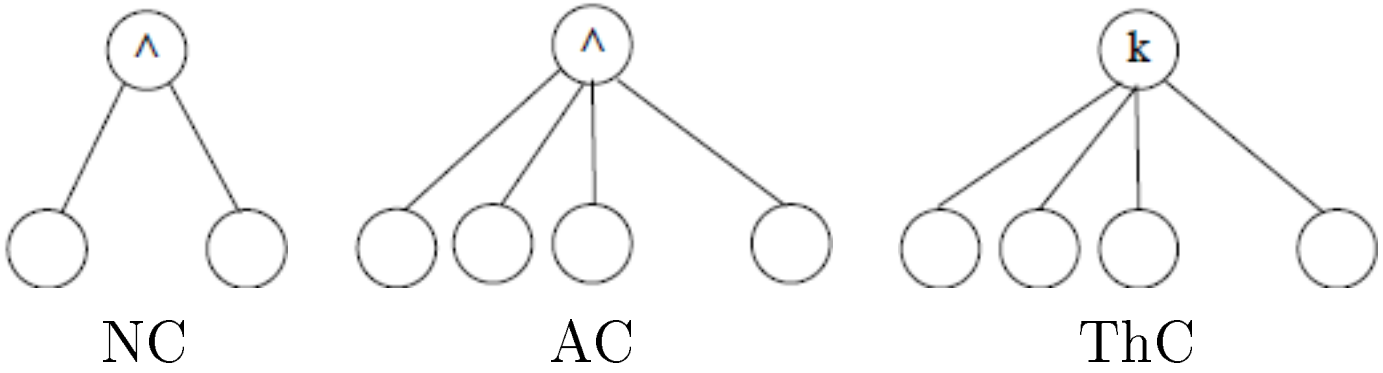
Definition

Let $\mathcal{C}$ be a sequence of circuits as above. Let $\tau \in \{\tau_c, \tau_{thc}\}$. Let $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau]$ be a query such that for all $n \in \mathbb{N}$, $I(0^n) = C_n$. Then:

- If $I \in \mathrm{FO}$, then $\mathcal{C}$ is a *first-order uniform* sequence of circuits.
- If $I \in \mathrm{L}$, then $\mathcal{C}$ is a *logspace uniform*.
- If $I \in \mathrm{P}$, then $\mathcal{C}$ is a *polynomial-time uniform*.
- e.t.c.

In the next frame we will define 3 families of circuit complexity classes. They vary depending on whether:

- all gates have bounded fan-in (NC)
- the "and" and "or" gates may have unbounded fan-in (AC)
- there are threshold gates (ThC)



Definition

Let $t(n)$ be a polynomially bounded function and $S \subseteq \mathrm{STRUC}[\tau]$ be a boolean query. Then S is in the *(first-order uniform) circuit complexity class* $\mathrm{NC}[t(n)]$, $\mathrm{AC}[t(n)]$, $\mathrm{ThC}[t(n)]$, respectively iff there exists a first-order query $I : \mathrm{STRUC}[\tau_s] \rightarrow \mathrm{STRUC}[\tau_{thc}]$ defining a uniform class of circuits $\mathcal{C} = \{C_n \mid C_n \doteqdot I(0^n)\}$ with the following properties:

Definition

Let $t(n)$ be a polynomially bounded function and $S \subseteq \mathrm{STRUC}[\tau]$ be a boolean query. Then S is in the *(first-order uniform) circuit complexity class* $\mathrm{NC}[t(n)]$, $\mathrm{AC}[t(n)]$, $\mathrm{ThC}[t(n)]$, respectively iff there exists a first-order query $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_{thc}]$ defining a uniform class of circuits $\mathcal{C} = \{C_n \mid C_n \doteqdot I(0^n)\}$ with the following properties:

1. For all $\mathcal{A} \in \mathrm{STRUC}[\tau]$, $\mathcal{A} \in S \iff C_{\|\mathcal{A}\|}$ accepts $\mathcal{A}$.
2. The depth of C_n is $\mathcal{O}(t(n))$.
3. The gates of C_n consist of binary "and" and "or" gates (NC), unbounded fan-in "and" and "or" gates (AC), and unbounded fan-in threshold gates (ThC), respectively.

Definition

Let $t(n)$ be a polynomially bounded function and $S \subseteq \mathrm{STRUC}[\tau]$ be a boolean query. Then S is in the *(first-order uniform) circuit complexity class* $\mathrm{NC}[t(n)]$, $\mathrm{AC}[t(n)]$, $\mathrm{ThC}[t(n)]$, respectively iff there exists a first-order query $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_{thc}]$ defining a uniform class of circuits $\mathcal{C} = \{C_n \mid C_n \doteqdot I(0^n)\}$ with the following properties:

1. For all $\mathcal{A} \in \mathrm{STRUC}[\tau]$, $\mathcal{A} \in S \iff C_{\|\mathcal{A}\|}$ accepts $\mathcal{A}$.
2. The depth of C_n is $\mathcal{O}(t(n))$.
3. The gates of C_n consist of binary "and" and "or" gates (NC), unbounded fan-in "and" and "or" gates (AC), and unbounded fan-in threshold gates (ThC), respectively.

For $i \in \mathbb{N}$ we denote $\mathrm{NC}^i \doteqdot \mathrm{NC}[(\log n)^i]$ and simillarly the AC^i and ThC^i. Also, $\mathrm{NC} \doteqdot \bigcup_{\mathbb{N}} \mathrm{NC}^i$.

Proposition

Addition of natural numbers, represented in binary, is first-order expressible.

Proposition

Addition of natural numbers, represented in binary, is first-order expressible.

We have already proven this Proposition using the well-known "carry-look-ahead" algorithm, through the formula φ_{add}, where:

- $\varphi_{carry}(x) \equiv (\exists y.y < x)\,[A(y) \wedge B(y) \wedge (\forall z.y < z < x)\,[A(z) \vee B(z)]]$
- $a \oplus b \equiv (a \vee b) \wedge (\neg a \vee \neg b)$
- $\varphi_{add} \equiv A(x) \oplus B(x) \oplus \varphi_{carry}(x)$

We assumed that the columns are denoted $n-1, ..., 0$ and the numbers similarly $a = a_{n-1}...a_0$.

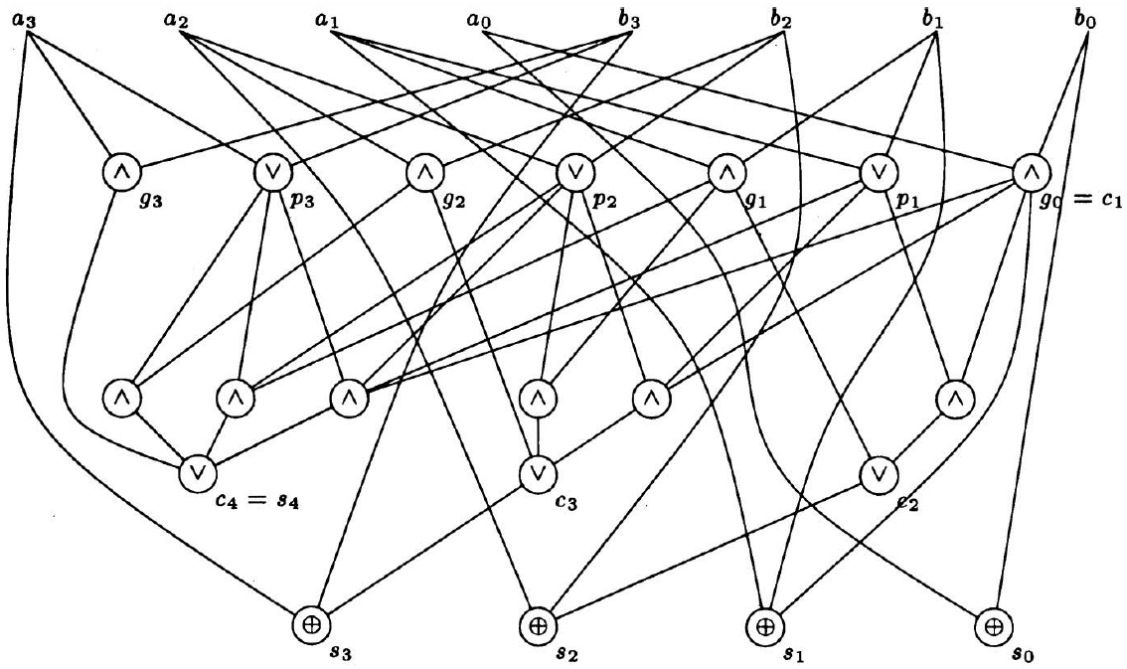
We want to express the formula φ_{add} through a boolean circuit.
Let:

- $a_i \doteqdot A(i)$ $b_i \doteqdot B(i)$ and $s_i \doteqdot \varphi_{add}(i)$
- $g_i \equiv A(i) \wedge B(i)$ and $p_i \equiv A(i) \vee B(i)$

We have:

$$c_i \doteqdot \varphi_{carry}(i) \equiv \bigvee_{j=0}^{i-1} \left(g_j \wedge \bigwedge_{k=j+1}^{i-1} p_k \right)$$

It is easy to see that the boolean circuit bellow computes the addition for $n = 4$



It is easy to see that depth of the equivalent circuit, resulted after the raplaement of the $\oplus$-gates with some "and", "or" and "not" gates, is constant (why?).
Thus Addition of two natural numbers is computable in AC^0.

It is easy to see that depth of the equivalent circuit, resulted after the raplaement of the $\oplus$-gates with some "and", "or" and "not" gates, is constant (why?).
Thus Addition of two natural numbers is computable in AC^0.

Every input in the new circuit can have at most n inputs. Therefore we can simulate each "and" ("or") gates with fan-in greater than 2, with at most $\log n$ "and" ("or") binary gates.
Thus Addition of two natural numbers is computable in NC^1.

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Let us calculate addition of two natural nunmbers using ambiguous arithmetic notation. That is a representation of natural numbers in binary, except that digits $0, 1, 2, 3$ may be used. For example:

$$3213 = 3{\cdot}2^3{+}2{\cdot}2^2{+}1{\cdot}2^1{+}3{\cdot}2^0 = 37 = 3221 = 3{\cdot}2^3{+}2{\cdot}2^2{+}2{\cdot}2^1{+}1{\cdot}2^0$$

We observe that we can calculate the carry from column i , by looking only at columns $i-1$ and $i-2$.

carries:	3	2	2	3	
		3	2	1	3
+		3	2	1	3
	3	2	2	1	0

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Similarly:

carries:	3	2	2	2	
		3	2	1	3
+		3	2	2	1
	3	2	2	1	0

Thus adding two n bit numbers in ambiguous notation can be done via an NC^0 circuit.

Theorem

For all $i \in \mathbb{N}$,

$$\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{ThC}^i \subseteq \mathrm{NC}^{i+1}$$

Theorem

For all $i \in \mathbb{N}$,

$$\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{ThC}^i \subseteq \mathrm{NC}^{i+1}$$

In order to prove the theorem above we will use the next proposition:

Proposition

The boolean majority query MAJ *is in* NC^1*, where*

$$\mathrm{MAJ} \doteqdot \{\mathcal{A} \in \mathrm{STRUC}[\tau_s] \mid \mathcal{A} \text{ contains more than } \|\mathcal{A}\|/2 \text{ "1"s}\}$$

Theorem

For all $i \in \mathbb{N}$,

$$\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{ThC}^i \subseteq \mathrm{NC}^{i+1}$$

In order to prove the theorem above we will use the next proposition:

Proposition

The boolean majority query MAJ *is in* NC^1*, where*

$$\mathrm{MAJ} \doteqdot \{\mathcal{A} \in \mathrm{STRUC}[\tau_s] \mid \mathcal{A} \text{ contains more than } \|\mathcal{A}\|/2 \text{ "1"s}\}$$

Hint: *Build an* NC^1 *circuit for majority by adding the n input bits via a full binary tree of height* $\log n$*, by using the ambiguous notation.*

We give a sketching of the proof:

Proof.

The first two containments are obvious (why?).

We give a sketching of the proof:

Proof.

The first two containments are obvious (why?).
For the third we can simulate any ThC-gate using a circuit of depth $\log n$ recognising MAJ. Let threshold gate with threshold value k.

- If $k \leq \|w\|/2$ we are just checking if $w1^{\|w\|-2k} \in \mathrm{MAJ}$.

We give a sketching of the proof:

Proof.

The first two containments are obvious (why?).
For the third we can simulate any ThC-gate using a circuit of depth $\log n$ recognising MAJ. Let threshold gate with threshold value k.

- If $k \leq \|w\|/2$ we are just checking if $w1^{\|w\|-2k} \in \mathrm{MAJ}$.
- If $k > \|w\|/2$, we are just checking if $w0^{2k-\|w\|} \in \mathrm{MAJ}$.

We give a sketching of the proof:

Proof.

The first two containments are obvious (why?).
For the third we can simulate any ThC-gate using a circuit of depth $\log n$ recognising MAJ. Let threshold gate with threshold value k.

- If $k \leq \|w\|/2$ we are just checking if $w1^{\|w\|-2k} \in \mathrm{MAJ}$.
- If $k > \|w\|/2$, we are just checking if $w0^{2k-\|w\|} \in \mathrm{MAJ}$.

$\square$

Corollary

$$\mathrm{NC} = \mathrm{AC} \dot{\div} \bigcup_{\mathbb{N}} \mathrm{AC}^i = \mathrm{ThC} \dot{\div} \bigcup_{\mathbb{N}} \mathrm{ThC}^i$$

Theorem

For all polynomially bounded and first-order constructible $t(n)$, the following classes are equal:

$$\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)] = \mathrm{AC}[t(n)]$$

Theorem

For all polynomially bounded and first-order constructible $t(n)$, the following classes are equal:

$$\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)] = \mathrm{AC}[t(n)]$$

Proof.

The equality of the first three classes has been proved.

♫ $\mathrm{FO}[t(n)] \subseteq \mathrm{AC}[t(n)]$

Let $S \subseteq \mathrm{STRUC}[\tau]$ a $\mathrm{FO}[t(n)]$ boolean query given by the quantifier block, $\mathrm{QB} = (Q_1 x_1 . M_1) \ldots (Q_k x_k . M_k)$, initial formula, M_0, and tuple of constants, $\overline{c}$.

Proof.

We must write a first-order query, I, to generate circuit $C_n = I(0^n)$, so that for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$,

$$\mathcal{A} \models [\mathrm{QB}]^{t(\|\mathcal{A}\|)}\, M_0\left(\vec{c}\,/\,\vec{x}\right) \iff C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}$$

Proof.

We must write a first-order query, I, to generate circuit $C_n = I(0^n)$, so that for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$,

$$\mathcal{A} \models [\mathrm{QB}]^{t(\|\mathcal{A}\|)}\, M_0\left(\vec{c}\,/\,\vec{x}\right) \iff C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}$$

Initially the circuit evaluates the quantifier-free formulas M_i, where $i \in \mathtt{n}+1$. The nodes $\langle M_i, b_1, ..., b_k\rangle$ will be the gates that have evaluated these formulas, i.e.,

$$\langle M_i, b_1, ..., b_k\rangle\left(\mathrm{bin}\left(\mathcal{A}\right)\right) = 1 \iff \mathcal{A} \models M_i\left(b_1, ..., b_k\right)$$

Proof.

Let φ^r defined as in the proof of $\mathrm{FO}[t(n)] \subseteq \mathrm{CRAM}[t(n)]$. We construct inductively the gate $\left\langle 2r, b_1 \ldots \hat{b}_i \ldots b_k \right\rangle$ so that

$$\left\langle 2r, b_1 \ldots \hat{b}_i \ldots b_k \right\rangle (\mathrm{bin}\,(\mathcal{A})) = 1 \iff \mathcal{A} \models \varphi^r\,(b_1, \ldots, b_k)$$

Proof.

Let φ^r defined as in the proof of $\mathrm{FO}[t(n)] \subseteq \mathrm{CRAM}[t(n)]$. We construct inductively the gate $\left\langle 2r, b_1 ... \hat{b}_i ... b_k \right\rangle$ so that

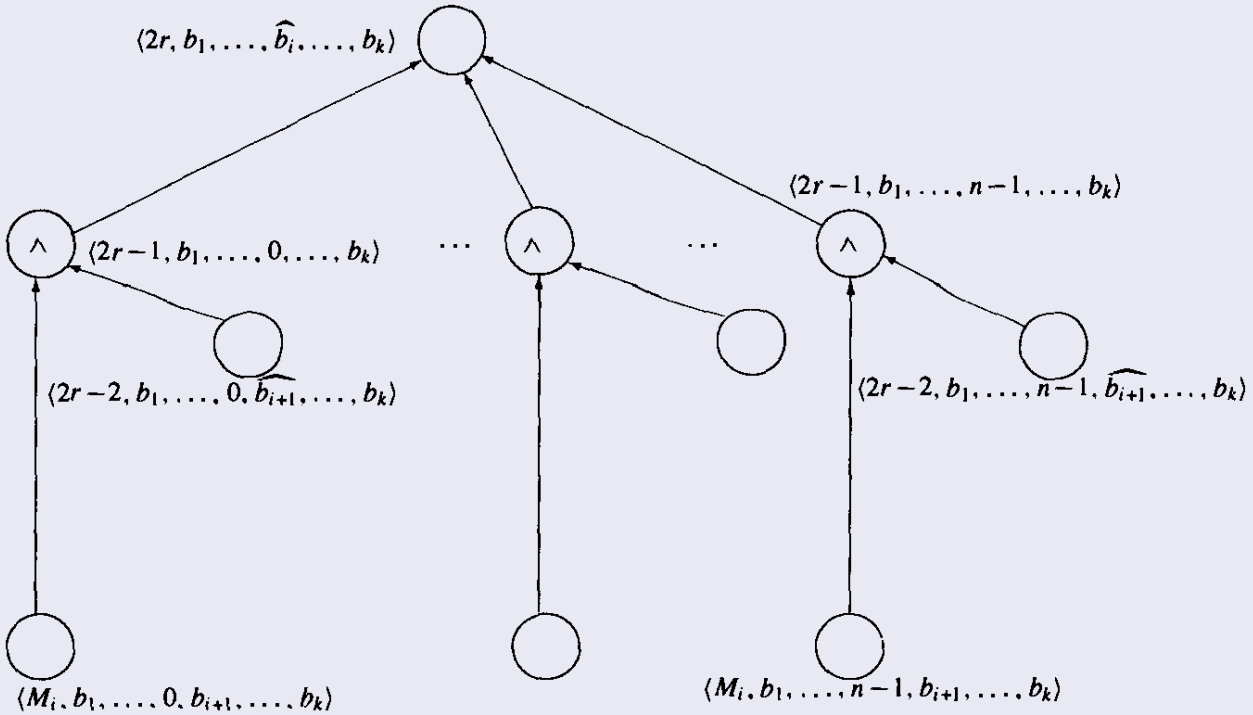
$$\left\langle 2r, b_1 ... \hat{b}_i ... b_k \right\rangle (\mathrm{bin}\,(\mathcal{A})) = 1 \iff \mathcal{A} \models \varphi^r\,(b_1, ..., b_k)$$

This is achieved by letting gate $\left\langle 2r, b_1 ... \hat{b}_i ... b_k \right\rangle$:

- Be "and"-gate ("or"), if $Q_i = \forall\ (\exists)$
- Has inputs $\left\langle 2r-1, b_1, ..., b_i, \widehat{b_{i+1}}, ..., b_k \right\rangle$, where $b_i \in |\mathcal{A}|$
  - $\left\langle 2r-1, b_1, ..., b_i, \widehat{b_{i+1}}, ..., b_k \right\rangle$ is a binary "and"-gate whoses inputs are $\langle M_i, b_1, ..., b_k \rangle$ and $\left\langle 2r-2, b_1, ..., b_i, \widehat{b_{i+1}}, ..., b_k \right\rangle$

Proof.

This circuit can be constructed via a first-order query I.



The $\left\langle 2r, b_1 \ldots \hat{b}_i \ldots b_k \right\rangle$ gate

$\square$

Proof.

♫ $\mathrm{AC}[t(n)] \subseteq \mathrm{IND}[t(n)]$

Let $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_c]$, a first-order query and $\mathcal{C} = \{C_i\}_{\mathbb{N}_{\geq 1}} = \{I(0^i)\}_{\mathbb{N}_{\geq 1}}$, a uniform sequence of $\mathrm{AC}[t(n)]$ circuits.

Proof.

♫ $\mathrm{AC}[t(n)] \subseteq \mathrm{IND}[t(n)]$

Let $I : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_c]$, a first-order query and $\mathcal{C} = \{C_i\}_{\mathbb{N}_{\geq 1}} = \{I(0^i)\}_{\mathbb{N}_{\geq 1}}$, a uniform sequence of $\mathrm{AC}[t(n)]$ circuits.

We must wright an inductive formula:

$$\Phi \equiv (\mathrm{LFP}\varphi\,(\bar{c}))$$

so that for all $\mathcal{A} \in \mathrm{STRUC}[\tau]$,

$$\mathcal{A} \models \Phi \iff C_{\|\mathcal{A}\|} \text{ accepts } \mathcal{A}$$

Proof.

From $\mathcal{A}$ we can get the circuit $C_{\|\mathcal{A}\|} \doteqdot \langle E, G_\wedge, G_\vee, G_\neg, \mathrm{bin}(\mathcal{A}), r\rangle$ via the first-order query I.

Proof.

From $\mathcal{A}$ we can get the circuit $C_{\|\mathcal{A}\|} \doteqdot \langle E, G_\wedge, G_\vee, G_\neg, \mathrm{bin}(\mathcal{A}), r\rangle$ via the first-order query I.

The following is a first-order inductive definition of the relation $V(x, b)$ meaning that gate x has boolean value b,

$$V(x,b) \equiv \mathrm{DEFINED}(x) \wedge [(L(x) \wedge (I(x) \leftrightarrow b)) \vee$$

$$(G_\wedge(x) \wedge (C(x) \leftrightarrow b)) \vee$$

$$(G_\vee \wedge (D(x) \leftrightarrow b)) \vee$$

$$(G_\neg(x) \wedge (N(x) \leftrightarrow b))]$$

Proof.

Where we have the abbreviations:

Proof.

Where we have the abbreviations:

$L(x) \equiv (\forall y)\neg E(y,x)$ x is a leaf

Proof.

Where we have the abbreviations:

$L(x) \equiv (\forall y)\neg E(y,x)$ x is a leaf

$\mathrm{DEFINED}(x) \equiv (\forall y)(\exists c)\,(E(y,x) \rightarrow V(y,c))$

x is ready to be defined.

Proof.

Where we have the abbreviations:

$L(x) \equiv (\forall y)\neg E(y,x)$ — x is a leaf

$\mathrm{DEFINED}(x) \equiv (\forall y)(\exists c)\,(E(y,x) \rightarrow V(y,c))$ — x is ready to be defined.

$C(x) \equiv (\forall y)\,(E(y,x) \rightarrow V(y,1))$ — all inputs of x's inputs are true

Proof.

Where we have the abbreviations:

$L(x) \equiv (\forall y)\neg E(y,x)$ — x is a leaf

$\mathrm{DEFINED}(x) \equiv (\forall y)(\exists c)\,(E(y,x) \rightarrow V(y,c))$ — x is ready to be defined.

$C(x) \equiv (\forall y)\,(E(y,x) \rightarrow V(y,1))$ — all inputs of x's inputs are true

$D(x) \equiv (\exists y)\,(E(y,x) \wedge V(y,1))$ — some of x's inputs are true

Proof.

Where we have the abbreviations:

$L(x) \equiv (\forall y)\neg E(y,x)$ — x is a leaf

$\mathrm{DEFINED}(x) \equiv (\forall y)(\exists c)\,(E(y,x) \rightarrow V(y,c))$ — x is ready to be defined.

$C(x) \equiv (\forall y)\,(E(y,x) \rightarrow V(y,1))$ — all inputs of x's inputs are true

$D(x) \equiv (\exists y)\,(E(y,x) \wedge V(y,1))$ — some of x's inputs are true

$N(x) \equiv (\exists! y)E(y,x) \wedge (\exists y)\,(E(y,x) \wedge V(y,0))$ — x's (unique) input is false.

Proof.

The inductive definition of V closes in exactly the depth of C_n , which is $\mathcal{O}(t(n))$ iterations.

Proof.

The inductive definition of V closes in exactly the depth of C_n , which is $\mathcal{O}(t(n))$ iterations.
Once it closes, $\Phi \equiv V(r, 1)$ expresses the acceptance condition in $\mathrm{IND}[t(n)]$, as desired.

$\square$

Proof.

The inductive definition of V closes in exactly the depth of C_n , which is $\mathcal{O}(t(n))$ iterations.
Once it closes, $\Phi \equiv V(r,1)$ expresses the acceptance condition in $\mathrm{IND}[t(n)]$, as desired.

□

Proposition

$$\mathrm{NC} = \mathrm{AC} = \mathrm{ThC} = \bigcup_{k=1}^{\infty} \mathrm{FO}\left[(\log n)^k\right] = \bigcup_{k=1}^{\infty} \mathrm{CRAM}\left[(\log n)^k\right]$$

Summary

Today we have seen:

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).
- An introduction to Circuit complexity.

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).
- An introduction to Circuit complexity.
- Relations between complexity calsses:

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).
- An introduction to Circuit complexity.
- Relations between complexity calsses:
  - $\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)] = \mathrm{AC}[t(n)]$

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).
- An introduction to Circuit complexity.
- Relations between complexity calsses:
  - $\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)] = \mathrm{AC}[t(n)]$
  - $\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{ThC}^i \subseteq \mathrm{NC}^{i+1}$

Summary

Today we have seen:

- An introduction a new model of computaion (RAM).
- An introduction to Iterating FO formulas, using Quantifier Blocks (QB).
- An introduction to Parallel computation.
- A precise model of Parallel computation (CRAM).
- An introduction to Circuit complexity.
- Relations between complexity calsses:
  - $\mathrm{CRAM}[t(n)] = \mathrm{IND}[t(n)] = \mathrm{FO}[t(n)] = \mathrm{AC}[t(n)]$
  - $\mathrm{NC}^i \subseteq \mathrm{AC}^i \subseteq \mathrm{ThC}^i \subseteq \mathrm{NC}^{i+1}$
  - $\mathrm{NC} = \mathrm{AC} = \mathrm{ThC} = \bigcup_{k=1}^{\infty} \mathrm{FO}\left[(\log n)^k\right] = \bigcup_{k=1}^{\infty} \mathrm{CRAM}\left[(\log n)^k\right]$

Bibliography

- Immerman, N. "Descriptive Complexity." (1999).
- Immerman, Neil. "Expressibility and parallel complexity." SIAM Journal on Computing 18.3 (1989): 625-638.
- Aho, Alfred V., and John E. Hopcroft. The design and analysis of computer algorithms. Pearson Education India, (1974).
- Stockmeyer, Larry, and Uzi Vishkin. "Simulation of parallel random access machines by circuits." SIAM Journal on Computing 13.2 (1984): 409-422.
- Vollmer, H. "Introduction to Circuit Complexity: A Uniform Approach. 1999." Texts Theoret. Comput. Sci (1999).
- Parhami, Behrooz. Introduction to parallel processing: algorithms and architectures. Springer Science & Business Media, (1999): 5-13.

Bibliography

- Barrington, David A. Mix, Neil Immerman, and Howard Straubing. "On uniformity within NC1." Journal of Computer and System Sciences 41.3 (1990): 274-306.
- Moschovakis, Y. "Elementary Induction on Abstract Structures. Vol. 77." Studies in Logic series", (1974): 57-59.