Parallel Random Access Machines

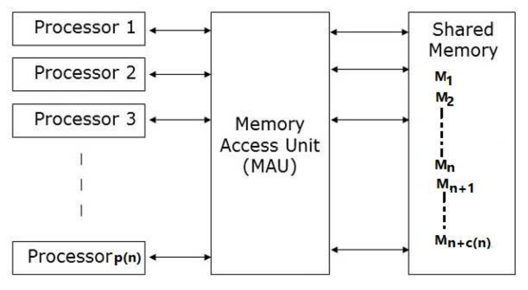
George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

Seminar Presentation
Lake Superior State University

Parallel Random Access Machines

- A **parallel random access machine** (**PRAM**) for n Boolean inputs consists of:



  - $p(n)$ processors P_i, $1 \leq i \leq p(n)$;
  - a read-only input tape of n cells $M_1, \ldots, M_n$ (or $X_1, \ldots, X_n$) containing the inputs $x_1, \ldots, x_n$; and
  - a shared memory of cells M_j, $n < j \leq n + c(n)$ (or Y_j, $j = 1, \ldots, c(n)$), all containing at first zeros (c is the **communication width**).

Operation of PRAMs

- P_i starts in the state $q(i, 0)$.
- At time step t:
  - depending on its state $q(i, t)$, P_i reads the contents of some cell M_j of the shared memory;
  - depending on $q(i, t)$ and the contents of M_j, it assumes a new state $q(i, t + 1)$; and
  - depending on $q(i, t + 1)$, it writes some information into some cell of the shared memory.
- The PRAM **computes** $f_n \in B_n$ **in time** $T(n)$ if the cell M_{n+1} (i.e., Y_1) of the shared memory contains on input $x = (x_1, \ldots, x_n)$ at time step $T(n)$ the output $f_n(x)$.

Exclusivity vs. Concurrency

- We distinguish between some models of PRAMs with different rules for solving read and write conflicts:
  - An **EREW PRAM** (**exclusive read, exclusive write**) works correctly only if, at any time step and for any cell, at most one processor reads the contents of this cell and at most one processor writes into this cell.
  - A **CREW PRAM** (**concurrent read, exclusive write**), or, shortly, **PRAM** allows that many processors read the contents of the same cell at the same time step, but it works correctly only if at any time step and for any cell at most one processor writes into this cell.

Resolution of Write Conflicts

- We distinguish between two more models of PRAMs with different rules for solving read and write conflicts:
  - A **CRCW PRAM** (**concurrent read, concurrent write**), or, shortly, **WRAM** solves write conflicts:

    If more than one processor tries to write at time step t into cell M_j, then the processor with the smallest number wins.
    This processor writes into M_j and all competitors fail to write.
  - A **WRAM** satisfies the **common write rule** (**CO WRAM**) if whenever several processors are trying to write into a single cell at the same time step, the values that they try to write are the same.

WRAMs vs. CO WRAMs

- It is obvious that a CO WRAM with p processors, communication width c and time complexity t can be simulated by a WRAM with the same p, c and t.
- In fact, the only change needed is replacing the Memory Access Unit by one that resolves write conflicts according to the processor index priority rule.

  Since competitors who are accessing the same memory location are attempting to write identical bits, accepting the one written by the winner would do as well as any other.

  This guarantees correctness of the operation.

Kucera's Theorem

Kucera's Theorem

A WRAM of p processors, communication width c and time complexity t may be simulated by a CO WRAM of $\binom{p}{2}$ processors, communication width $c+p$ and time complexity $4t$.

- The simulation is step-by-step.

  We use processors P_j, $1 \leq j \leq p$, for the simulation and P_{ij}, $1 \leq i < j \leq p$, for some extra work.

  Since P_j and P_{ij} never work simultaneously, $\binom{p}{2}$ processors are sufficient if $p \geq 3$.

  Each computation step of the WRAM is simulated by 4 computation steps of the CO WRAM.

Kucera's Simulation

- At first the processors P_j, $1 \leq j \leq n$, simulate the reading and the internal computations of the WRAM.

  P_j writes into the j-th extra cell of the shared memory the number of that cell into which P_j likes to write.
- In the following two steps:
  - P_{ij} decides whether P_j loses a write conflict against P_i;
  - P_{ij} writes a mark $\#$ into the j-th extra cell iff P_j has lost a write conflict against P_i.

    This causes no conflict for CO WRAMs.
- In the fourth step P_j reads whether it has lost a write conflict.

  Only if P_j has not lost a write conflict, P_j simulates the write phase of the WRAM.

  This causes no write conflict at all.

An Upper Bound for EREW PRAMs

Theorem

Assuming processors of arbitrary power, every $f : \{0,1\}^n \to \{0,1\}$ can be computed in time $\lceil \log n \rceil + 1$ by an EREW PRAM having n processors and communication width n.

- Sketch of the proof (we take $n = 8$).

  The goal is to compute $f(x_1, x_2, \ldots, x_8)$, where x_i is in M_i.

Processor	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Memory	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8
Step 0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Step 1	x_1, x_2	x_2, x_3	x_3, x_4	x_4, x_5	x_5, x_6	x_6, x_7	x_7, x_8	x_8
Step 2	x_{1-4}	x_{2-5}	x_{3-6}	x_{4-7}	x_{5-8}	x_{6-8}	x_{7-8}	x_8
Step 3	$f(x_{1-8})$	x_{2-7}	x_{3-8}	x_{4-8}	x_{5-8}	x_{6-7}	x_{7-8}	x_8

An Upper Bound for PRAMs

Theorem

Assuming processors with realistic power, every $f : \{0,1\}^n \to \{0,1\}$ can be computed in time $\lceil \log n \rceil + 2$ by an PRAM having $n \cdot 2^n$ processors and communication width $n \cdot 2^n$.

- We assume the function is presented in disjunctive normal form

$$f(x_1, \ldots, x_n) = \bigvee_{i=1}^{k} (\ell_{i,1} \wedge \cdots \wedge \ell_{i,n}).$$

  Using the preceding algorithm the i-th group of n processors $(1 \leq i \leq 2^n)$ can compute the value of the i-th conjunction with n literals in $\lceil \log n \rceil + 1$ steps.

  In the last step, a processor having a disjunct evaluated to one, writes a 1 in position M_{n+1}, which is prearranged to contain a 0.

An Upper Bound for CO WRAMs

Theorem

Assuming processors with realistic power, every $f : \{0,1\}^n \to \{0,1\}$ can be computed in 2 steps by an CO WRAM having $n \cdot 2^n$ processors and communication width 2^n.

- We assume the function is presented in conjunctive normal form

$$f(x_1, \ldots, x_n) = \bigwedge_{i=1}^{k} (\ell_{i,1} \vee \cdots \vee \ell_{i,n}).$$

The i-th group of n processors ($1 \leq i \leq 2^n$) can compute the value of the i-th clause with n literals in a single step and store it to the i-th memory location.

In the last step, each processor having a clause evaluated to zero, writes a 0 in position M_{n+1}, which is prearranged to contain a 1.

Thank you!

- In closing...

Thank you for your Attention!!