CS855: Parallel machine models

Marc Moreno Maza

February 19, 2008

- The PRAM Model.
- Simulation of large PRAMs on small PRAMs.
- Handling Shared Memory Access Conflicts: PRAM submodels.
- Comparing the Computational Power of PRAM Submodels.

Random Access Machine is a convenient model of a sequential computer. Its features are as follows.

- The *computation unit* executes a user defined program.
- It uese a *read-only input tape* and a *write-only output tape*.
- The RAM has an unbounded number of local *memory cells*.
- Each memory cell can hold an integer of *unbounded size*.
- The *instruction set* includes operations for: moving data between memory cells, comparisons and conditional branching, additions, subtractions, multiplications.
- Execution starts with the first instruction of the program and terminates when an `Halt` instruction is reached.
- Each operation takes one *time unit* regardless of the the operand sizes.
- **Time complexity** = the number of instructions executed.
- **Space complexity** = the number of memory cells accessed.

Parallel Random Access Machine is a natural generalization of RAM. It is also an idealization of a *shared memory machine*. Its features are as follows.

- It holds an *unbounded collection of RAM processors* $P_0, P_1, P_2, \ldots$ **without tapes**.
- It holds an *unbounded collection of shared memory cells* $M[0], M[1], M[2], \ldots$
- Each processor P_i has its own (unbounded) local memory (register set) and P_i knows its index i.
- Each processor P_i can access any shared memory cell $M[j]$ in *unit time*, unless there is a conflict (see further).
- The input of a PRAM program consists of n items stored in $M[0], \ldots, M[n-1]$.
- The output of a PRAM program consists of n' items stored in n' memory cells, say $M[n], \ldots, M[n+n'-1]$.
- A PRAM instruction executes in a 3-phase cycle:

1. **Read** (if needed) from a shared memory cell,
2. **Compute** locally (if needed),
3. **Write** in a shared memory cell (if needed).

- All processors execute their 3-phase cycles **synchronously**.
- **Special assumptions** have to be made in order to resolve shared memory access conflicts.
- The only way processors can exchange data is by writing into and reading from memory cells.
- P_0 has a special *activation register* specifying the maximum index of an active processor:
  1. Initially, only P_0 is active; it computes the number of required active processors,
  2. Then, P_0 loads this number in the activation register,
  3. The corresponding processors start executing their programs.

- Computations proceed until P_0 halts, at which time all other active processors are halted.
- **Parallel time complexity** = the time for P_0's computations.
- **Parallel space complexity** = the maximum number of shared memory cells in use during the computations.

The PRAM Model is attractive for designing parallel algorithms:

- It is **natural**: the number of operations executed per one cycle on p processors is at most p.
- It is **strong**: any processor can read or write any shared memory cell in **unit time**.
- It is **simple**: it abstracts from any communication or synchronization overhead.

This natural, strong and simple PRAM model can be used as a benchmark: If a problem has no feasible (or efficient) solution on a PRAM then it is likely that it has no feasible (or efficient) solution on any parallel machine.

- The PRAM model is an idealization of existing shared memory parallel machines.
- The PRAM ignores lower level architecture constraints (memory access overhead, synchronization overhead, intercommunication throughput, connectivity, speed limits, etc.)

- A **small-memory PRAM** satisfies the axioms of a PRAM except that is has a bounded number of shared memory cells.
  - A m-**cell PRAM** is a small-memory PRAM with m shared memory cells.
  - If the input (or output) data set exceeds the capacity of the shared memory, then this data can be distributed evenly among the registers of the processors.
  - Limiting the amount of shared memory corresponds to restricting the amount of information that can be communicated between processors in one step.
  - For example, a distributed memory machine with processors interconnected by a shared bus can be modeled as a PRAM with a single shared memory.
- A **small PRAM** satisfies the axioms of a PRAM except that is has a bounded number of processors. A p-**processor PRAM** is a small PRAM with $p + 1$ processors (counting P_0).

To make the PRAM model realistic and useful, some mechanism has to be defined to resolve read and write conflicts to the same memory cell.

EREW (Exclusive Read Exclusive Write): No two processors are allowed to read or write the same shared memory cell simultaneously.

CREW (Concurrent Read Exclusive Write): Simultaneous reads of the same memory cell are allowed, but no two processors can write the same shared memory cell simultaneously.

PRIORITY CRCW (PRIORITY Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- Processors are assigned fixed and distinct priorities.
- In case of write conflict, the processor with highest priority is allowed to complete WRITE.

ARBITRARY CRCW (ARBITRARY Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- In case of write conflict, one randomly chosen processor is allowed

to complete WRITE.

- An algorithm written for this model should make no assumptions about which processor is chosen in case of write conflict.

COMMON CRCW (COMMON Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- In case of write conflict, all processors are allowed to complete WRITE iff all values to be written are equal.
- An algorithm written for this model should make sure that this condition is satisfied. If not, the algorithm is illegal and the machine state will be undefined.

Consider a p-processor PRAM with $p < n$. Assume that the shared memory contains n distinct items and P_0 owns a value x. The goal is to let P_0 know whether x occurs within the n items.

1. <u>EREW PRAM algorithm</u>:
   (a) P_0 broadcasts x to $P_1, P_2, \ldots, P_p$ in $O(\log(p))$ steps using binary broadcast tree.
   (b) Every processor $P_1, P_2, \ldots, P_p$ performs local searches on (at most) $\lceil n/p \rceil$ items, hence in $\lceil n/p \rceil$ steps.
   (c) Every processor defines a flag `Found` (equal to `true` or `false`). The final answer is obtained by a *parallel reduction* that is by means of a binary tree.

   This leads to $T(n,p) = O(\log(p) + \lceil n/p \rceil)$.

2. <u>CREW PRAM algorithm</u>: A similar approach, but $P_1, P_2, \ldots, P_p$ can read x in $O(1)$. However, the final reduction is still in $\log(p)$, leading again to $T(n,p) = O(\log(p) + \lceil n/p \rceil)$.

3. COMMON PRAM algorithm: Now, the final step takes $O(1)$. Indeed, those processors with their flag `Found` equal to `true` can write simultaneously to the same memory cell initialized to `false`. Hence, we have $T(n, p) = O(\lceil n/p \rceil)$.

Proposition 1 *Let $p' < p$. Then, any problem that can be solved on a p-processor PRAM in t steps can be solved on a p'-processor PRAM in $t' = O(tp/p')$ steps assuming the same size of shared memory.*

PROOF. In order to reach this result, each of the processors ${P'}_i$ of the p'-processor PRAM can simulate a group G_i of (at most) $\lceil p/p' \rceil$ processors of the p-processor PRAM as follows. Each simulating processor ${P'}_i$ simulates one 3-phase cycle of G_i by

1. executing all their READ instructions,
2. executing all their local computations,
3. executing all their WRITE instructions.

One can check that, whatever is the model for handling shared memory cell access conflict, the simulating PRAM will produce the same result as the larger PRAM. □

Observe that result holds also for $p' = 1$.

Proposition 2 *Assume $m' < m$. Then, any problem that can be solved on a p-processor and m-cell PRAM in t steps can be solved on a* $\max(p, m')$*-processor and m'-cell PRAM in $t' = O(tm/m')$ steps.*

Proof. Naturally, the idea is to use the register set of the processors of the m'-cell PRAM in order to compensate the diminution of shared memory. This is why it is necessary to assume that the m'-cell PRAM has at least m' processors. (After that, one can use Proposition 1 to save on processors.)

Let $P_1, \ldots, P_p$ be the processors of the m-cell PRAM:

- We use processors ${P'}_1, \ldots, {P'}_{m''}$ on the m'-cell PRAM to simulate $P_1, \ldots, P_p$ where $m'' = \max(p, m')$.

Moreover, we (mentally) group the m cells of the m-cell PRAM into m' continuous segments $S_1, \ldots, S_{m'}$ of size m/m'.

- We use the register set of processor ${P'}_i$ for simulating the segment S_i, for all $1 \leq i \leq m'$.

- We use the share memory cell $M'[i]$, for $1 \leq i \leq m'$, on the m'-cell PRAM, as an auxiliary memory.

Simulation of one 3-phase cycle of the m-cell PRAM:

`READ:` for all $0 \leq k < m/m'$ repeat

1. for all $1 \leq i \leq m'$, the processor P'_i writes the value of the k-th cell of S_i into $M'[i]$;
2. for all $1 \leq i \leq p$, the processor P'_i reads from the share memory, **provided** that P_i would read its value at position congruent to k modulo m/m'.

`COMPUTE:` the local computation of P'_i is simulated by P_i , for all $1 \leq i \leq p$.

`WRITE:` Analogous to `READ`. $\square$

By PRAM submodels, we mean either EREW, CREW, COMMON, ARBITRARY or PRIORITY.

Definition 1 *PRAM submodel A is computationally stronger than PRAM submodel B, written* A $\geq$ B *if any algorithm written for B will run unchanged on A in the same parallel time, assuming the same basic properties.*

Proposition 3 *We have:*

$$\text{PRIORITY} \geq \text{ARBITRARY} \geq \text{COMMON} \geq \text{CREW} \geq \text{EREW}.$$

Theorem 1 *Any polylog time PRAM algorithm is robust with respect to all PRAM submodels.*

In other words, any PRAM algorithm which runs in polylog time on one submodel can be simulated on any other PRAM submodel and run within the same complexity class. This results from Proposition 3 and Lemma 2. Lemma 1 provides a result weaker than Lemma 2 but the proof of the former helps understanding the proof of the latter.

Lemma 1 *Assume PRIORITY CRCW with the priority scheme based trivially on indexing: lower indexed processors have higher priority. Then, one step of p-processor m-cell PRIORITY CRCW can be simulated by a p-processor mp-cell EREW PRAM in* $O(\log(p))$ *steps.*

PROOF. Naturally, the idea is to

- store all the WRITE (or READ) *needs* for one cycle in memory
- evaluate their priorities
- execute the instruction of the winner

But there is a trap, we should avoid access conflict also during this simulation algorithm.

1. Each PRIORITY processor P_k is simulated by EREW processor P'_k, for all $1 \leq k \leq p$.
2. Each shared memory cell $M[i]$, for all $i = 1, \ldots, m$, of PRIORITY
   - is simulated by an array of p shared memory cells

$M'[i,k], k = 1, \ldots, p$ of EREW,

- $M'[i,1]$ plays the role of $M[i]$,
- $M'[i,2], \ldots, M'[i,p]$ are auxiliary cells used for resolving conflicts,
  - initially empty,
  - $M'[i,1], \ldots, M'[i,p]$ are regarded as the row of a complete binary tree T_i with p leaves and height $\lceil \log(p) \rceil$; initially, they are regarded as the leaf row of T_i.

3. Simulation of a PRIORITY WRITE substep. Each EREW processor
   - must find out whether it is the processor with lowest index within the group asking to write to the same memory cell, and if so,
   - must become the group winner and perform the WRITE operation; the other processors of its group just fail and do not write.

   This is done as follows:

   (a) For all $1 \le k \le p$ repeat: if P_k wants to write into $M[i]$, then P'_k turns **active** and becomes the k-th leaf of T_i.

(b) Each active **left** processor stores its ID into the parent cell in its tree, marks it as **occupied** and remains active.

(c) Each active **right** processor checks its parent cell: if it is empty, then it stores its ID there and remains active, otherwise it becomes idle (and he lost).

(d) This is repeated one row after another from bottom to top in T_i, in $\lceil \log(p) \rceil$ iterations.

(e) The process who managed to reach the root of T_i, becomes the winner and can WRITE. □

Lemma 2 *One step of PRIORITY CRCW with p processors and m shared memory cells by an EREW PRAM in* $O(\log(p))$ *steps with p processors and* $m + p$ *shared memory cells.*

PROOF.

1. Each PRIORITY processor P_k is simulated by EREW processor P'_k.
2. Each PRIORITY cell $M[i]$ is simulated by EREW cell $M'[i]$.

3. EREW uses an auxiliary array A of p cells.
4. If P_k wants to access $M[i]$, then processor P'_k writes pair (i, k) into $A[k]$.
5. If P_k does not want to access any PRIORITY cell, processor P'_k writes $(0, k)$ into $A[k]$.
6. All p processors sort the array A into **lexicographic order** using $(\log p)$-time parallel sort.
7. Each P'_k appends to cell $A[k]$ a flag f defined as follows
   - $f = 0$ if the first component of $A[k]$ is either 0 or it is the same as the first component of $A[k-1]$.
   - $f = 1$ otherwise.
8. Further steps differ for simulation of WRITE or READ.

   PRIORITY WRITE:

   (a) Each P'_k reads the triple (i, j, f) from cell $A[k]$ and writes it into $A[j]$.

(b) Each P'_k reads the triple (i, k, f) from cell $A[k]$ and writes into $M[i]$ iff $f = 1$.

PRIORITY READ:

(a) Each P'_k reads the triple (i, j, f) from cell $A[k]$.

(b) If $f = 1$, it reads value v_i from $M[i]$ and overwrites the third component in $A[k]$ (the flag f) with v_i.

(c) In at most log$p$ steps, this third component is then distributed in subsequent cells of A until it reaches either the end or an element with a different first component.

(d) Each P'_k reads the triple (i, j, v_i) from cell $A[k]$ and writes it into $A[j]$.

(e) Each P'_k who asked for a READ reads the value v_i from the triple (i, k, v_i) in cell $A[k]$.

Most of the material of these course notes are inspired by the following.

- Parallel Algorithms (WISM 459, 2005/2006) by Rob Bisseling.
  `http://www.math.uu.nl/people/bisselin/pas1.html`
- Topics in Parallel Computing (CS 838, Univ. of Wisconsin-Madison, Spring 1999) by Pavel Tvrdik.
  `http://www.cs.wisc.edu/ tvrdik/cs838.html`
- Parallel Algorithms (CS 662, San Diego Univ., Spring 1996) by Roger Whitney.
  `http://www.eli.sdsu.edu/courses/spring96/cs662/notes/`