PRAMs

CS255

Chris Pollett

Mar. 6, 2006.

Outline

- The PRAM model of parallel computation
- Brent's Theorem
- Sorting on a PRAM

The PRAM Model

- Our third model of parallel computation will be synchronous parallel random access machines.
- A parallel processor will consist of p processors, each of which can be a viewed as supporting the RAM model of computation:
  - Each processor has a finite instruction set supporting direct and indirect addressing, simple branching, and arithmetic instructions.
  - These instructions are used to manipulate a set of registers each of which can hold one integer.
  - In the PRAM setting unlike the usual RAM setting, each processor only has finitely many registers reserved to itself.
- The p-processors share a global memory consisting of M locations, that each processor may read from and write to.

More on the PRAM Model

- A computation in the PRAM model proceeds as a series of parallel steps.
- In each such step, each processor:
  - chooses a global memory location to read from
  - executes an instruction on the operand fetched together with any of the operands in local registers. This might modify the local registers.
  - finally, it writes to a single location of its choice.
- We insist on synchrony: that is, each processor finishes the parallel step i before the parallel step i+1 begins.
- Conflict resolution can be handled in different ways: Exclusive Read/Exclusive Write (EREW), Concurrent Read/Exlusive Write(CREW), and Concurrent Read/Concurrent Write (CRCW).
- Exclusive means only one processor can do that action on a given memory location at a given time. Concurrent is the opposite of this.
- We will mainly be interested in EREW and CREW PRAMs.

Complexity Classes

We want to define complexity classes which characterize those problems which can be executed more efficiently if we have more processors:

Definition

The class NC consists of languages L that have a PRAM algorithm A such that for any $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts,
- $x \notin L \Rightarrow A(x)$ rejects,
- the number of processors used by A on x is polynomial in $|x|$
- the number of steps used by A on x is polylogarithmic in $|x|$

The class RNC (where processors are allowed to use random coins) is defined the same way except we modify the first two conditions to:

- $x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq 1/2$,
- $x \notin L \Rightarrow \Pr[A(x) \text{accepts}\} = 0$.

The class ZNC is RNC $\cap$ co-RNC where co-RNC are those languages whose complement is in RNC.

For each of these classes we can define an analogous function/algorithm class.

Brent's Theorem

Before we give some NC and RNC algorithms for various problems let's first connect the PRAM model with our earlier model.

Theorem: Any depth d, size n combinational circuit with bounded fan-in can be simulated by a p processor CREW algorithm in O(n/p + d) time.

Proof: We store the inputs to the combinational circuit in the PRAMs global memory. We also reserve in global memory a location to store each output of each combination element of the circuit. A single combinational element can be simulated by a single PRAM processor in O(1) steps: The processor reads the input values of the element from the memory location that are being used to store the elements inputs, stores these in registers, then using these registers computes the outputs of the element and then writes these back into the global memory in the appropriate output memories. This works since the fan-in of each element is finite. …(cont'd next page)

Proof of Brent's theorem cont'd

To simulate our combinational circuit we need a way to schedule each combinational element on the p-processors so that the total time to simulate all the elements is O(n/p +d). Our constraint is that we can't schedule an element until its inputs are ready. Concurrent reads will be used if several elements being simulated require the same value.

The idea is we first simulate all the depth 1 elements, then all the depth 2 elements, etc. At each depth we do p of the elements at a time. So if there are n_i elements at depth i, we can simulate this level in $O(\lceil n_i/p \rceil)$ time. If the depth of our circuit is d, then the total time to do the simulation will be some constant times

$$\sum_{i=1}^{d} \lceil \frac{n_i}{p} \rceil \leq \sum_{i=1}^{d} (\frac{n_i}{p} + 1) = \frac{n}{p} + d$$

as desired. Note the sum of the n_i will be n the total number of elements in the circuit.

Corollaries

Corollary Any depth d, size n combinational circuit with bounded fan-in **and bounded fan-out** can be simulated on a p-processor EREW PRAM in O(n/p +d) time.

Proof: To avoid concurrent reading, after the output of a combinational element is computed, it is not directly read by the processors requiring its value. Instead, the output is copied by the processor to the O(1) many inputs memory locations of elements that require its value. (Here is where we use the bounded fan-out restriction.) QED.

Corollary If a p processor PRAM algorithm A runs in time t, then for any p' < p, there is a p'-processor PRAM algorithm A' for the same problem that runs in time O(pt/p').

Proof: Let the time steps of A be numbered 1,2…,t. Algorithm A' simulates the execution of each time step in O($\lceil p/p' \rceil$) time. As the total time is t and p' <p the total time of the simulation is O($\lceil p/p' \rceil$t) =O(pt/p').

Sorting on a PRAM

- We now work towards a ZNC algorithm for sorting which run in O(log n) time doing a total of O(n log n) operations on all processors. Let P_i denote the *i*th processor.
- We begin by considering a PRAM variant of Quicksort:

0. If n=1 stop.
1. Otherwise, pick a splitter uniformly at random from the n input elements
2. Each processor determines whether its element is bigger or smaller than the splitter.
3. Let j denote the rank of the splitter. If j is not in [n/4, 3n/4] the step is declared a failure and we go back to step 1. Otherwise, we move the splitter to processor P_j. Each element that is smaller than the splitter is moved to a distinct processor is P_i such that i <j. Each element that is larger than the splitter is moved to a distinct processor P_k where k>j.
4. We sort recursively the data in the processors P_1 through P_{j-1} and the data in P_{j+1} through P_n.

Analysis

- The second step above is O(1) time.
- To do step 3 each processor P_i sets a bit b_i in one of its registers to 0 if its element is greater than the splitter and to 1 otherwise.
- For all i, let $S_i=\sum_{t<=i} b_t$. From HW3, we will know S_i can be computed in O(log n) steps and from this we can do stage 3 in O(log n) steps.
- The expected number of time before step 1-3 succeeds is 1/(1/2) = 2 so the total runtime will be $O(\log^2 n)$.

More Sorting

- Suppose we have n processors and n elements. Suppose the first r processors have values in sorted order.
- We can use these r elements as splitters to sort the remaining n-r elements.
- The goal is to insert the n-r unsorted elements among the splitters in the following sense.
  1. Each processor should end up with a distinct input element.
  2. Let s_j denote the jth largest splitter and $i(s_j)$ denote the index of the processor containing s_j after the insertion. Then for all $k< i(s_j)$, processor P_k contains an element that is smaller than s_j and for all $k>i(s_j)$, P_k contains an element that is larger than s_j.
- We hope we can do this in O(log n) time.

Yet More Sorting

- If so, we could pick $n^{1/2}$ elements at random. Then using all n processors sort them using our first algorithm in $O(\log^2 n^{1/2})=O(\log n)$ steps.
- Then using these sorted elements insert the remaining elements among them in O(log n) steps (idea is again like the HW problem).
- Treat the remaining elements that are inserted between splitter as subproblems, recur on each subproblem whose size exceed log n otherwise use LogSort:
  - compare each element in parallel with its neighbour first to its left, swap if necessary; then to its right, swap if necessary, do this O(log n) times.
- We analyze the whole algorithm next day.