Distributed Algorithms

CS255

Chris Pollett

Mar. 15, 2006.

Outline

- Our Distributed Model
- The Choice Coordination Problem

Our Distributed Model

- In the PRAM model all of the processors used the same clock so all the processors computation steps were in sync.
- We will now consider the situation where we have n processors each with its own clock. So a step on one processor might be longer or shorter than some other processor.
- There is a global memory consisting of m registers.
- As several processors might attempt to simultaneously read/modify a register, we will assume before a processor accesses a global register it must first get a lock for it.
- While it has the lock of a register, no other processor can access that register and must wait.
- When the lock is released, all waiting processors are notified and can contend for the lock again.

The Choice Coordination Problem

- Motivating story:
  - Have mites which survive by making colonies in the ears of moths.
  - If they infect both ears of the same moth, the moth can’t hear bat sonar calls, and it and the mite colonies will be eaten.
  - How can the mites agree on only one ear to infect?
- We will be interested in a computer science variation on this problem called the **Choice Coordination Problem**.
- We will have n processors in our distributed setting.
- We want them to agree on a value between 1 and m.
- We will know an agreement has been met at the point when exactly one of the registers our processors have access to contains a special symbol #.
- An $\Omega(n^{1/3})$ time deterministic lower bound is known for this problem in the distributed model.
- We will show there is a expected constant time randomized algorithm.

Warm-Up Algorithm

- We first consider the simplified case of only two processor which are synchronized.
- Let P_i denote the processor and C_i denote its choice.
- Finally, let B_i be a value which is local to processor P_i only.

Synch-CCP:

Input: Registers C_0 and C_1 initialized to 0.

Output: Exactly one of the register has the value #.

```
0.  $P_i$ is initially scanning the register $C_i$ and has its local variable $B_i$ initialized to 0.
1.  Read the current register and obtain a bit $R_i$.
2.  Select one of three cases:
    1.  case [$R_i$ = #]: halt;
    2.  case [$R_i = 0$, $B_i = 1$]: write # into the current register and halt;
    3.  case [otherwise]: assign an unbiased random bit to $B_i$ and write $B_i$ into the current register.
3.  $P_i$ exchanges its current register with $P_{1-i}$ and returns to step 1.
```

Analysis

- Let's look at the correctness of Synch-CCP:
  - First notice, at most one register can ever have # written into it. Why? If both registers get the same value # then by 2.1 they must have both written # in the same iteration. Suppose this happens on the kth iteration. Let $B_i(k)$ and $R_i(k)$ denote the values use by P_i just after step 1 of the kth iteration. The previous wound must have used case 2.3, so we know $R_0(k) = B_1(k)$ and $R_1(k) = B_0(k)$. The only way a # could be written is if $R_i = 0$ and $B_i=1$; but then $R_{1-i} = 1$ and $B_{1-i}=0$, so P_{1-i} can't write 0 in that iteration.
- Notice during each iteration, the probability that both B_i have the same value is a 1/2. If the two bits are ever different then within two stages the algorithm stops. So after k steps the odds the algorithm has not stop is $O(1/2^k)$.
- So with odds $1-O(1/2^k)$ the algorithm terminate in k steps.

The Asynchronous Problem

- We now assume the two processors may be executing at varying speeds and cannot exchange the registers after each iteration.
- We no longer assume that the two processors begin by scanning different registers.
- We assume that each processor chooses its starting processor at random.
- The two processors could be in a conflict at the very first step so locking needs to used.
- To do coordination we want to use time-stamps.
- We will assume a read on C_i will yield a pair $<t_i, R_i>$ where t_i is the timestamp and R_i is the register.

Asynchronous-CCP

Input: Registers C_0 and C_1 initialized to <0,0>.
Output: Exactly one of the two registers has value #.

0. P_i is initially scanning a randomly chosen register. Thereafter, it changes its current register at the end of each iteration. The local variables T_i and B_i are initialized to 0.
1. P_i obtains a lock on the current register and reads $<t_i,R_i>$.
2. P_i selects one of five cases:
    1. case [$R_i = \#$]: halt;
    2. case [$T_i < t_i$]: set $T_i = t_i$ and $B_i = R_i$
    3. case [$T_i > t_i$]: write # into the current register and halt;
    4. case [$T_i = t_i, R_i = 0, B_i = 1$]: write # into the current register and halt
    5. case [otherwise]: Set $T_i = T_{i,}+1$ and $t_i = t_{i,}+1$ assign a random bit-value to B_i , and write $<t_i , B_i>$ to the current register.
3. P_i releases the lock on its current register, moves to the other register, and returns to step 1.

Analysis

Theorem For any $c>0$, Asynchronous-CPP has total cost exceeding c with probability at most $2^{-\Omega(c)}$.

Proof: The main difference between this and the synchronous case is in step 2.2 and 2.3. Case 2.2 is supposed to handle where the processor is playing catch up with the other processor; Case 2.3 handles where the processor is ahead of the other processor. To prove correctness of the protocol, we consider the two cases (2.3, 2.4) where a processor can write a # to its current cell. At the end of an iteration, a processor T_i will equal that of the current register t_i . Further # cannot be written in the first iteration by either processor. …

More Proof

Suppose P_i has just entered case 2.3, with some timestamp T^*_i, and its current cell is C_i with timestamp $t^*_i < T_{i,}$. The only possible problem is that P_{1-i} might write # into register C_{1-i} . Suppose this error occurs, and let t^*_{1-i} and T^*_{1-i} be the timestamp during the iteration for the other processor.

As P_i comes to C_i with a timestamp of T^*_i , it must have left C_{1-i} with a timestamp before P_{1-i} could write # into it. Since timestamps don't decrease $t^*_{1-i} >= T^*_i$. Further P_{1-i} cannot have its timestamp T^*_{1-i} exceed t^*_i since it must go to C_{1-i} from C_i and the timestamp of that register never exceeds t_i. So we have $T_{1-i} <= t^*_i < T_{i,} <= t^*_{1-i}$. This means P_{1-i} must enter case 2.2 as $T_{1-i} < t^*_{1-i}$. This contradicts it being able to write a #.

We can analyze the case P_i has just entered case 2.4 in a similar way, except we would reach the conclusion that $T_{1-i} <= t^*_i = T_{i,} <= t^*_{1-i}$ and so it is possible that $T_{1-i} = t^*_{1-i}$. But if this even happens, we are in the synchronous situation so our earlier correctness argument works. Thus,we established correctness of the algorithm.

Runtime

- We now just need to analyze the runtime of our above algorithm.
- The cost is proportional to the largest timestamp.
- Only case 2.5 increase the timestamp of a register, and this only happens if case 2.4 does not apply.
- Furthermore, the processor that raises the timestamp must have its current B_i value chosen during a visit to the other register. So the analysis of the synchronous case applies.