Optimal Parallel Randomized Algorithms for Sparse Addition and Identification

Advanced algorithms and parallel programming

Valentina Ionata Pietro Ferretti

General problem

- *Parallel addition of n numbers in shared memory models*

Standard algorithms are not sensitive to properties of the input, thus randomized algorithms can beat standard parallel algorithms in specific cases:

e.g. let's consider the case when there is **a large amount of zero operands** among the numbers to be added

Proposed Solution

Randomized parallel algorithm:

- the number of non-zero operands must be much smaller than the number of zeros among the numbers to be added
- we suppose we are using a CRCW PRAM with unlimited memory and unlimited number of processors
- time complexity $O(\log m)$, space complexity $O(m)$ w.h.p.

The Algorithm

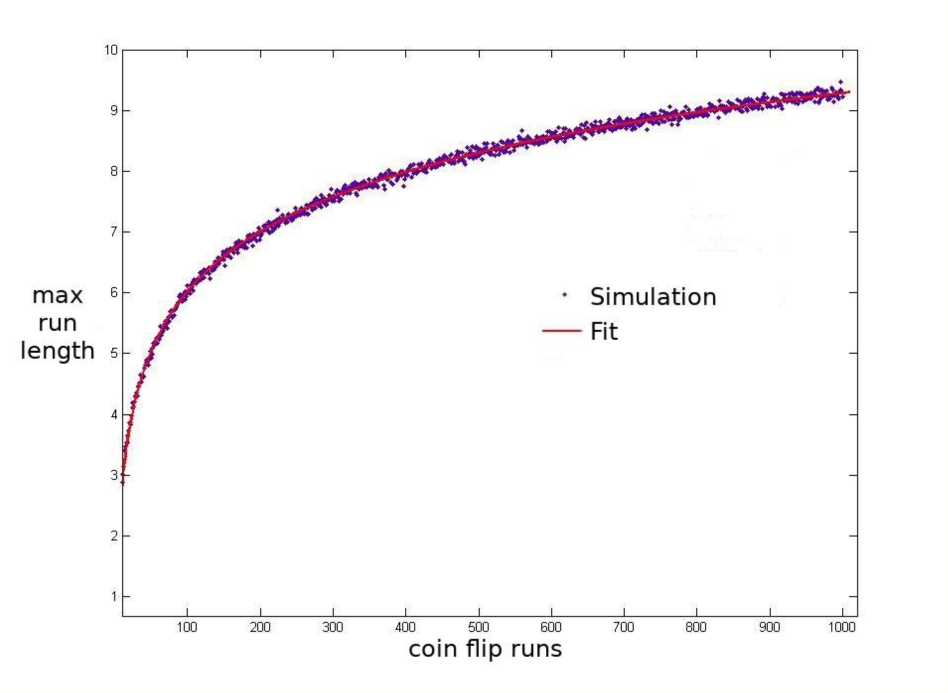
It consists of two phases:

- **Estimation** of the number m of non-zero terms
- **Parallel addition** on a smaller set of inputs

Estimation - *First phase*

1. Each processor P_i has a number to add.

2. **PRODUCE-AN-ESTIMATE**: every processor with $x_i \neq 0$ flips a coin until a head is found. A shared memory location will contain the *highest number of coin flips reached*.



Estimation - *First phase*

3. PRODUCE-AN-ESTIMATE is executed ***k* times.**

4. Then each P_i computes an estimation m' for the number of non-zero operands m using the following formula:

$$(1) \quad E \leftarrow (\log 2) \frac{E_1 + \cdots + E_k}{k}$$

$$(2) \quad m' \leftarrow \exp(2) \cdot \exp(E) + d$$

where $d \geqslant 1$ is a constant.

Code (Estimation)

INITIALIZATION

Processor P_1 initializes a special shared memory location (CLOCK) to zero.
Then, each P_i executes $TIME_i \leftarrow 0$.

ESTIMATE

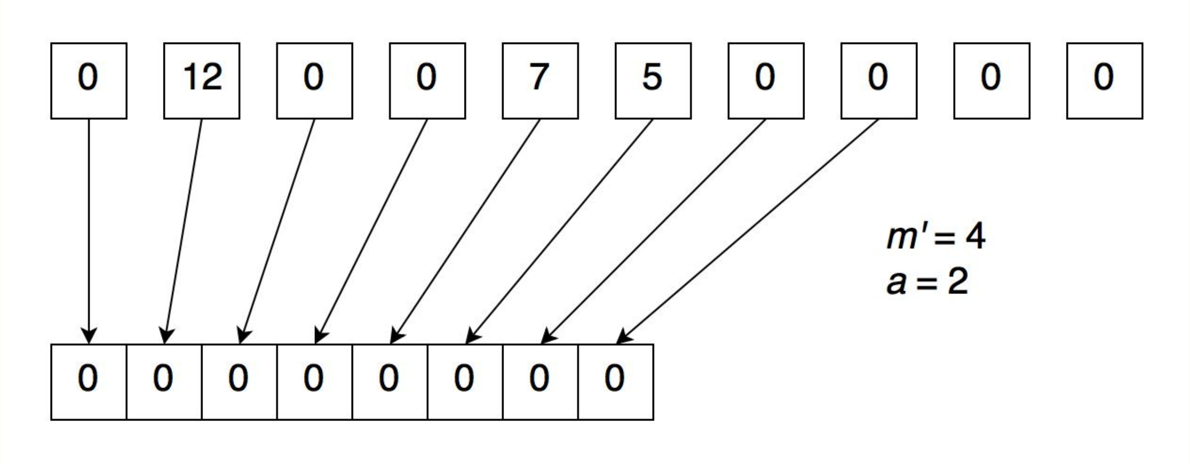
```
Processor Pi
        if x ≠ 0
                then begin (1) Flip a fair coin (two-sided)
                           (2 ) If the outcome is 'tail' then
                                 begin (2a) TIMEi = TIMEi + 1
                                            (2b) CLOCK ←TIME;
                                            (2c) goto(l)
                                 end
                     end
```

#all processors wait for everyone to be finished

Each P_i with $x_i \neq 0$ reads CLOCK and makes its value to be the current estimate.

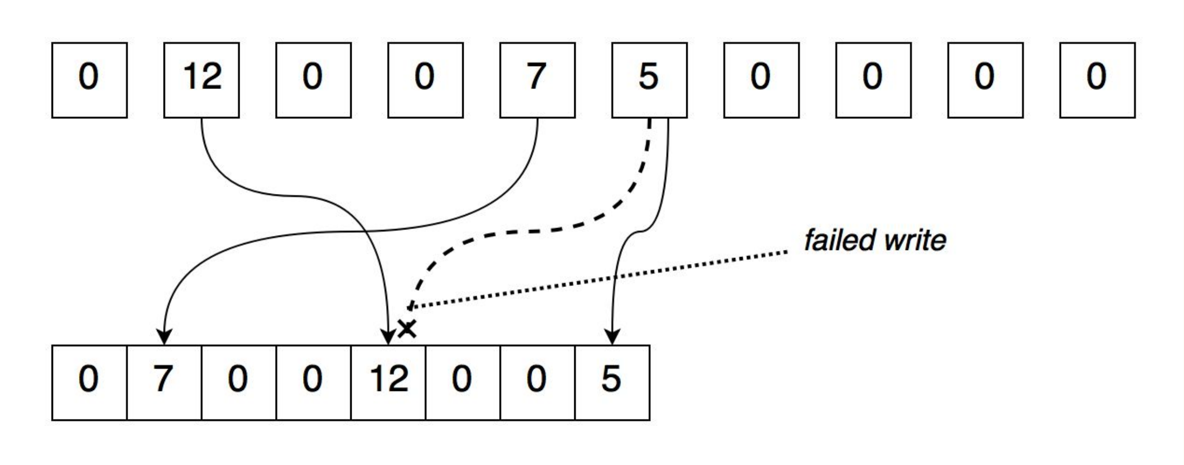
Addition - *Second phase*

1. **Initialization:**
   $a \cdot m'$ memory locations are set to zero. ($a \geq 4$ constant)



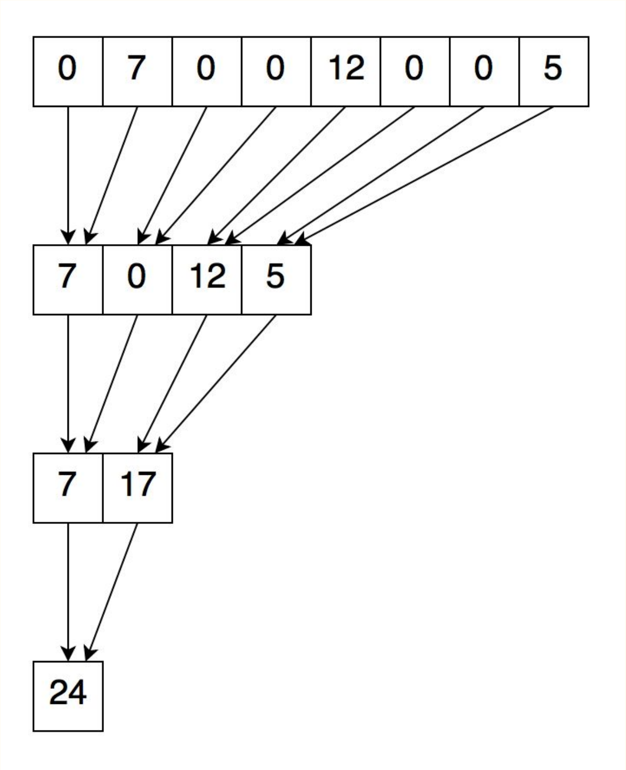
Addition - *Second phase*

2. **Memory Marking:**
   each processor with a non-zero number tries to write it to a random empty location in the zeroed space, until they all succeed.



Addition - *Second phase*

3. **Addition:**
   standard parallel addition of all the $a \cdot m'$ numbers (zero and non-zero). The result is of course the sum of all the non-zero numbers.



Code (Addition)

INITIALIZATION

In one parallel step, processors initialize $a \cdot m' + 2$ shared memory locations to zero, by executing: "Processor Pi writes a zero to M(j), if $j < am' + 2$". Then, they all execute TIMEj $\leftarrow$ 0.

MEMORY MARKING

Processor Pj

IF xj$\neq$0 then

```
    BEGIN (1) Select y equiprobably at random from { 1, 2, . . . . am' }
          (2) TIMEj ←TIMEj + 1
          (3) Read M(y); TIMEj ← TIMEj ← 1
          (4) If M(y) = 0 then write xj into M(y). Also, TIMEj ← TIMEj + 1
          (5) If the "write" failed then
              BEGIN (5a) write TIME, into M(am' + 1)
                    (5b) go to (1)
              END
    END
```

Code (Addition)

The following part is executed by Pj with xj= 0 and by "successful" Pj with xj ≠0.

(6) Read M(am' + 1) into a local variable R1

(7) Wait for 8 steps

(8) Read M(am' + 1) into a local variable R2

(9) If R1 ≠R2, then go to (6)

ADDITION

(Processor P, is assigned to location M(j), 1 <j < am') From this point on, processors Pj (where 1 <j< am') perform a standard parallel addition of the numbers M(l), ... M(am'). In the i-th parallel step of the addition, processor Pj adds M(j) and M(j+2^i) into M(j), for j=k · 2^i+ 1, k=0, 1,2 ,..,am'/2^i.

Properties

- The sum is always correct

- The time complexity is in O(log m) and the space complexity in O(m) (where m is the number of non-zero entries among the terms we need to sum) with high probability

- We can control the precision of the estimate, and thus the variance of the running time, by changing the amount of estimations of m to run (represented by the constant k)

Possible Applications

Marking algorithm:

- solves the processor identification problem for a strong WRAM in O(m) parallel time with arbitrarily high probability.

Processor identification problem:

N processors are given, each keeping either a 0 or a 1. The problem is for each processor to find out which are the processors with the 1's.

Potential Problems

- The algorithm assumes a CRCW PRAM
    - we think this can be implemented using atomic compare-and-swap primitives
- We have few cores at our disposal (up to 4). It's not a given that the performance will be as good as the one predicted by the paper for n processors.
- We will have to find working values for k, a, d since they're not specified in the paper
- Since we will be summing fixed-width numbers we will have to consider overflows

Testing and Performance

Datasets for testing are easy to generate (random numbers).

Tests to perform:

- different amounts of numbers to sum
- different ratios of zero and non-zero terms
- different values of k, a, d

Comparison with other algorithms:

- performance gain w.r.t. a serial algorithm
- performance gain w.r.t. the standard parallel sum (parallel “lower bound”)

Planned Schedule

- ➢ Implementation:
  - serial algorithm
  - parallel “lower bound”
  - parallel randomized algorithm
- ➢ Testing
- ➢ Presentation