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

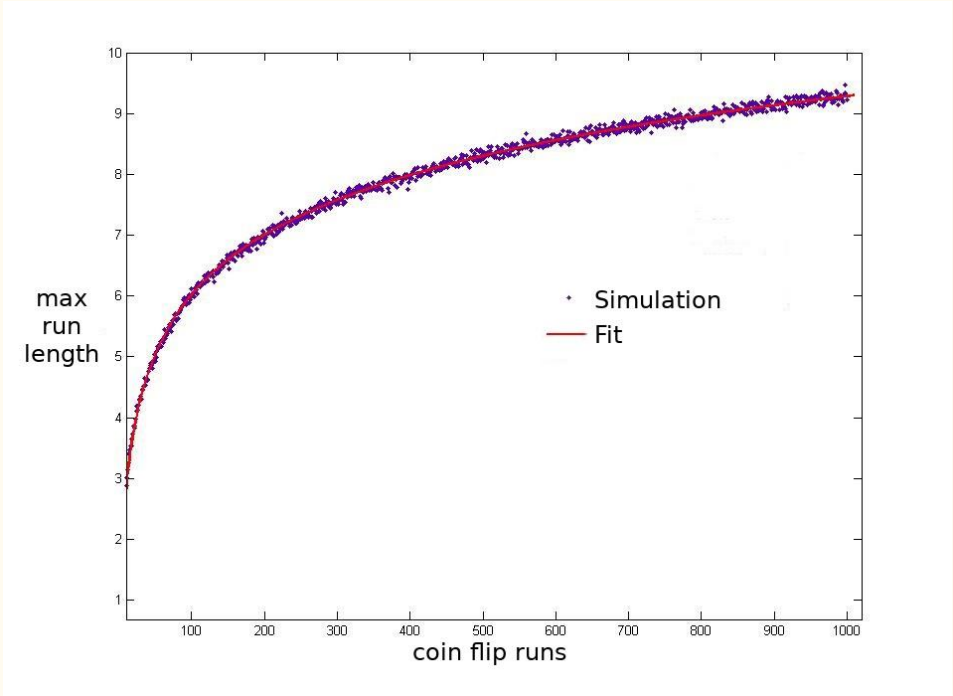
It consists of two phases:

- **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 $TIME_i \leftarrow 0$.

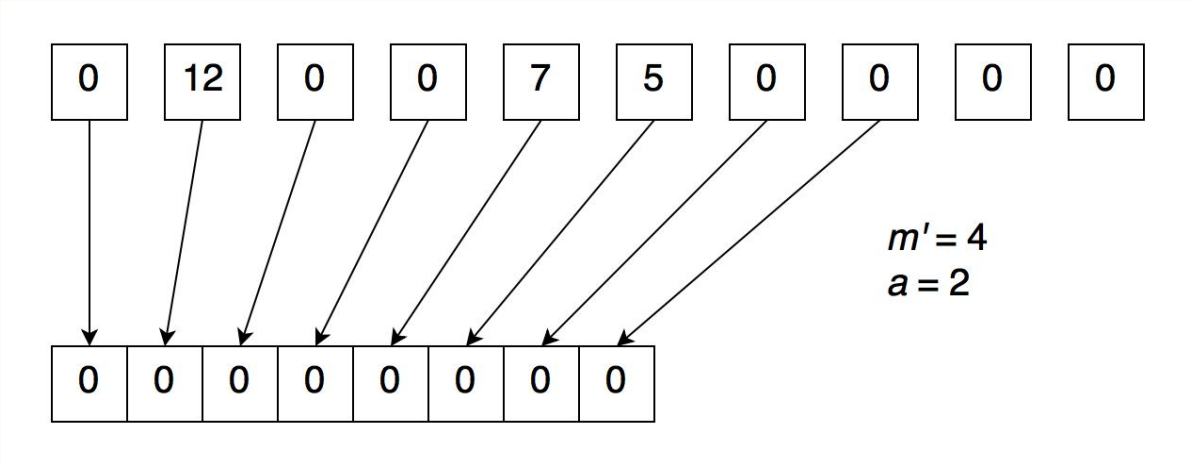
ESTIMATE

```
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.

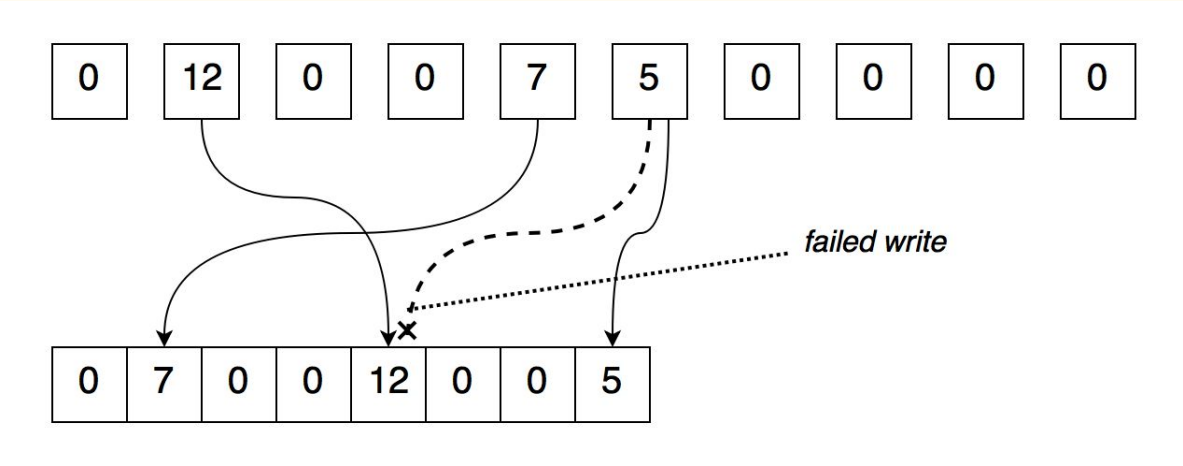
Addition - *Second phase*

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



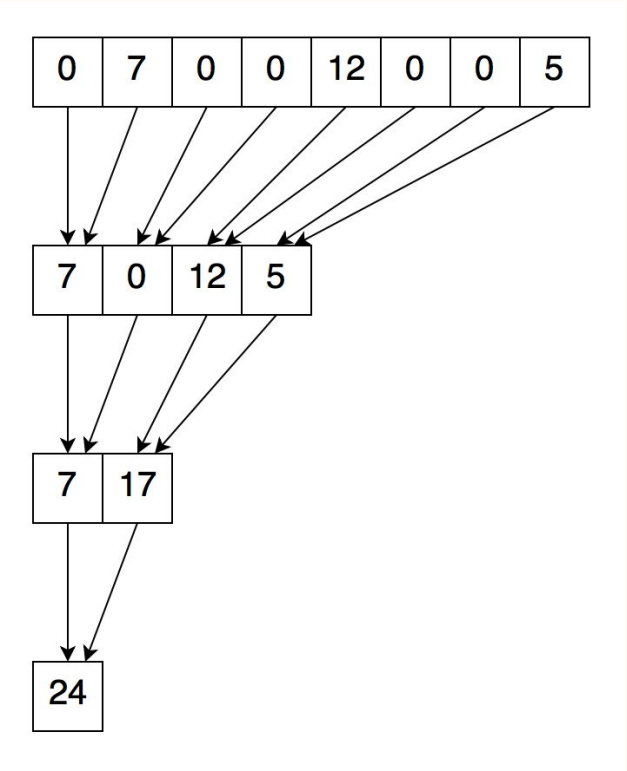
Addition - *Second phase*

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 · 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 ← 0.

MEMORY MARKING

```
Processor Pj
IF xj≠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)

Running time for Estimation

Given any $\delta > 0$, if we choose $k \geq 4/\delta$, then with probability at least 1 - δ we have that:

the total running time of estimation is $O(4/\delta \cdot \log m)$, *or* $O(k \cdot \log m)$

Brief explanation:

1. **a single iteration** takes time equal to the maximum of m independent geometric random variables with p = 1/2, which **can be approximated as log m/log 2 time** using the harmonic series
2. with Chebyshev's inequality **we can predict how often the sum** of the times for each iteration **will exceed the logarithmic expected time**, thus the relationship with δ

Estimation accuracy

The estimation is computed using the harmonic series as an approximation ($E \approx \log m / \log 2$), but the approximation **does not converge** for big ks.

If we choose $k \geq 4/\delta$, with probability $1 - \delta$ we have:

$$|E - \log m| \leq 2 , \qquad \text{i.e.} \quad m \leq m' \leq m \cdot e^4$$

(note: $e^4 \approx 55$)

We have no guarantees except for these bounds.

Running time for Addition

Memory Marking

With $a \cdot m$' free memory, each processor will have chance at least $(a - 1)/a$ to succeed in writing to an empty memory space.

The running time is the maximum of m geometric random variables, the same as in the estimation step, thus the time **can be approximated as logarithmic**. The memory marking step is therefore in $O(\log m)$ on average.

Parallel Addition

The parallel addition algorithm's running time is trivially in $O(\log m')$.

Total running time

The total average running time is $O(\log m) + O(\log m) + O(\log m)$ => **$O(\log m)$**

Worst case:

- *Estimation*: each iteration has standard deviation 2, very improbable to take much longer than log m
- *Memory marking*: each processor's probability to need a repeated number of tries decreases geometrically (again, improbable)
- *Parallel addition*: can't take more than log m' steps

Considerations

Compared to the standard parallel addition, we need to remember that:

- the estimation step is expensive (uses rand()) and grows with k
- the memory marking step usually takes less than log m' since the estimation is in excess
- the parallel addition takes log m' time, which can be up to $\log(e^4) = 4$ times more than log m

Thus we expect the algorithm to be better than the simple parallel addition o**nly on extreme cases**, when the array of number to sum is **very sparse** (a very small fraction of non-zero numbers).

Problems encountered

- rand() returns 30-bit numbers, but we needed 64-bit random numbers
  - solution: generate a number bit by bit
- difficulties in integrating the project on both Windows and Linux environments
- measuring the performance of the algorithm is not easy, because of the noise created by thread scheduling
  - solution: taking the average time of many runs; running when the machine is idle
  - consequence: hard to measure worst times
- the paper is not so clear and glosses over a few things, like the proofs for the algorithm complexity

Testing and Performance

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

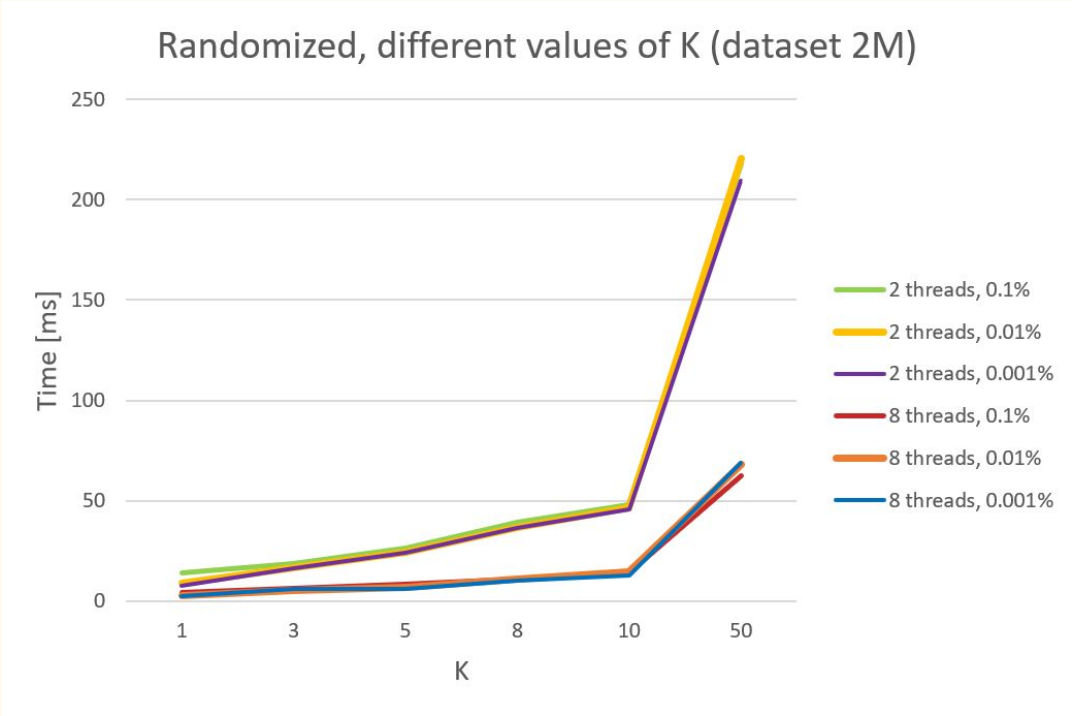
Tests to perform:

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

Comparison with other algorithms:

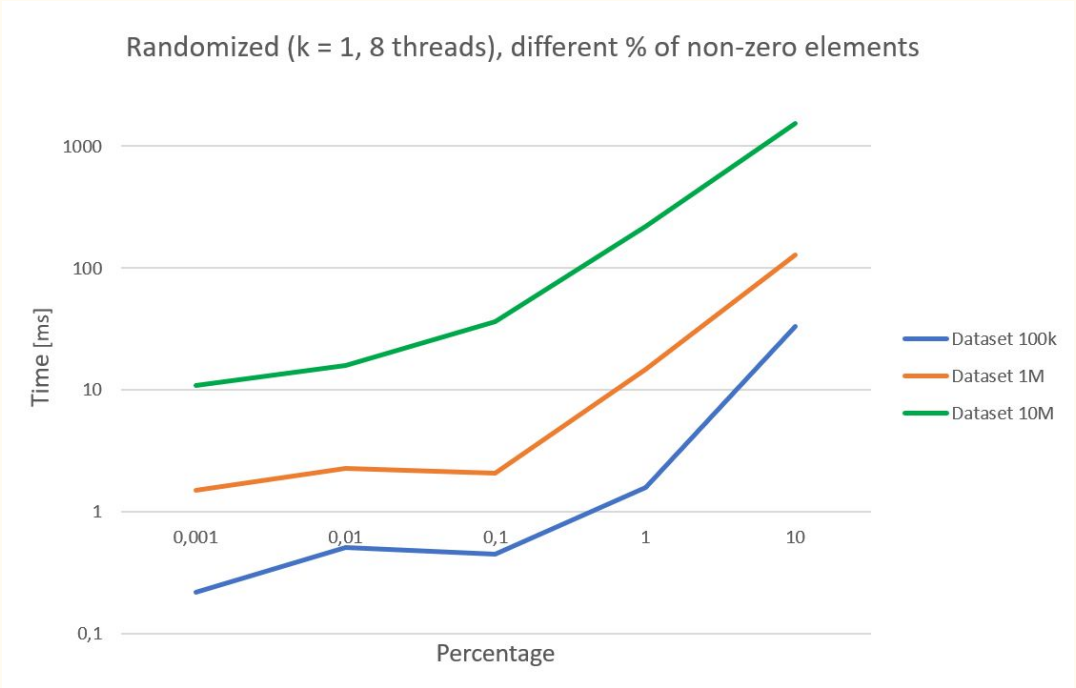
- performance gain w.r.t. a serial algorithm
- performance gain w.r.t. the standard parallel sum (parallel “lower bound”)

Benchmark results - selection of K



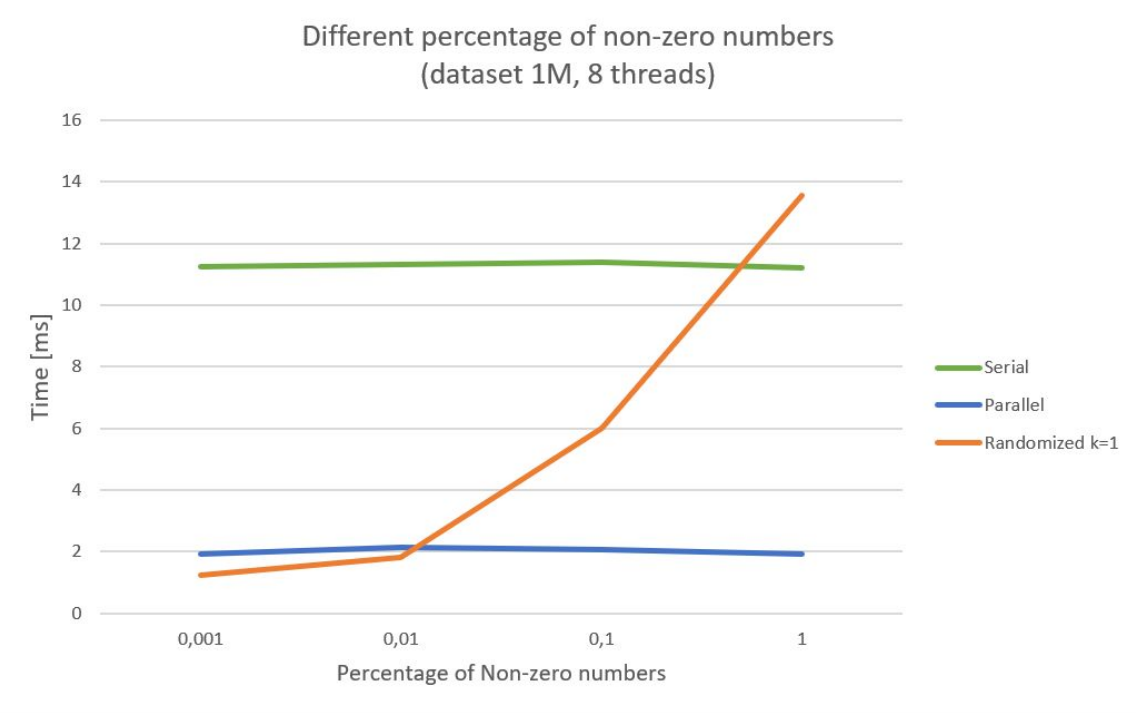
- The time to execute the algorithm always increases with K
- We choose a low K for our tests (K=1)

Benchmark results - percentage of nonzero numbers



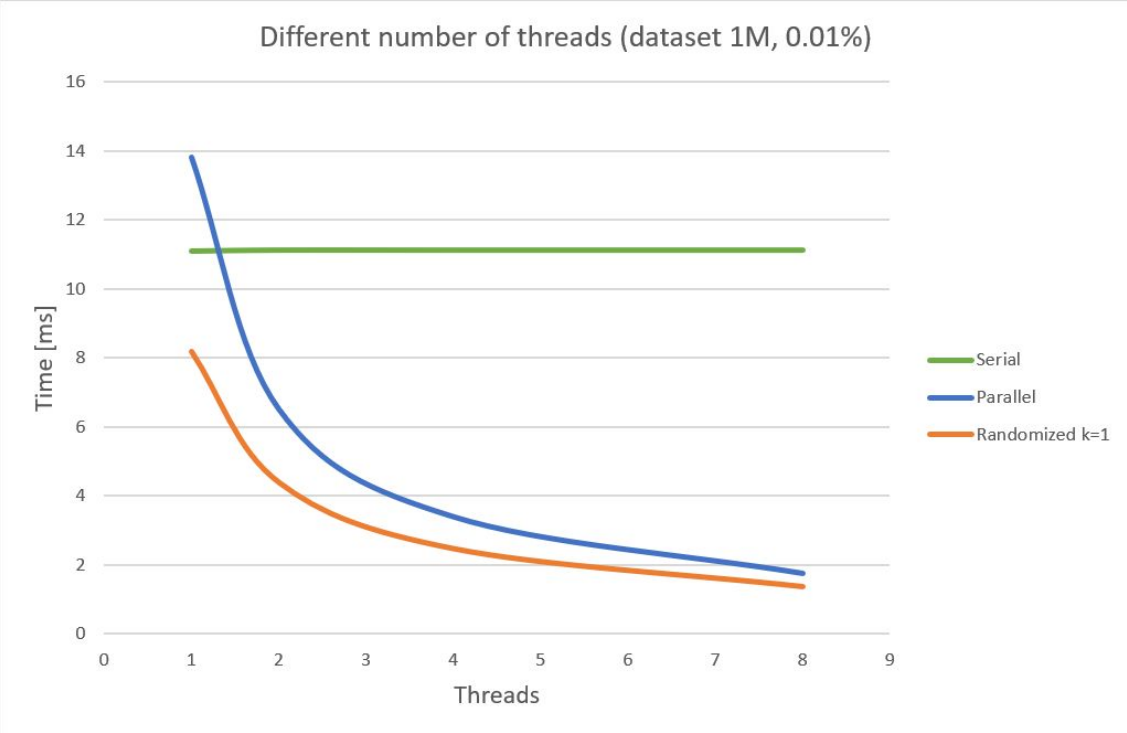
- The execution time for the randomized algorithm increases a lot when the number of non-zero elements is great
- We should use this algorithm only with a small percentage of non-zero elements

Benchmark results - % nonzero, comparison



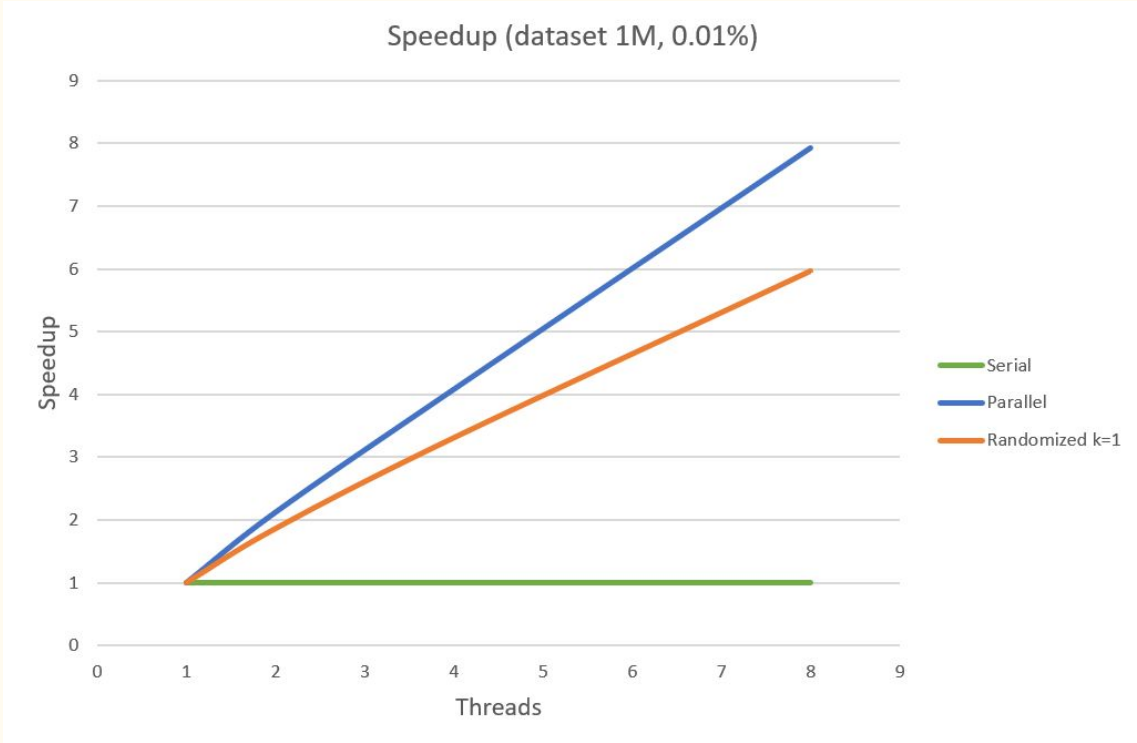
- The randomized algorithm is better than the parallel only when the % of nonzero numbers is under 0.01% (1 over 1000)

Benchmark results - threads



- Fixed K=1 and non-zero=0.01%
- The algorithm is well parallelized, the execution times easily improves when running with more threads

Benchmark results - threads (speedup)



- The speedup from the parallelization doesn't saturate (the dataset is much bigger than the number of threads)

Conclusions

- The randomized algorithm works and it is feasible, but it is better than the standard parallel sum only when the non-zero numbers in the dataset are **very sparse** ($\leq 0.01\%$ of all numbers).

- Even if having a bigger k gives more theoretical guarantees, this in practice makes the performance worse. The best results are reliably obtained with $\boldsymbol{k=1}$ (only one estimation).