A small discussion on

Prefix Sum

INAF HPC School 2025
Catania, Sep. 22nd - 26th

Luca Tornatore
O.A.T.S

Algorithm analysis

In the exercise we have introduced the prefix sum; a brief search may convince you that is a fundamental algorithm upon which many other algorithms are built.

The serial algorithm is quite obivous, although the most obvious implementation may be optimized for modern architectures.

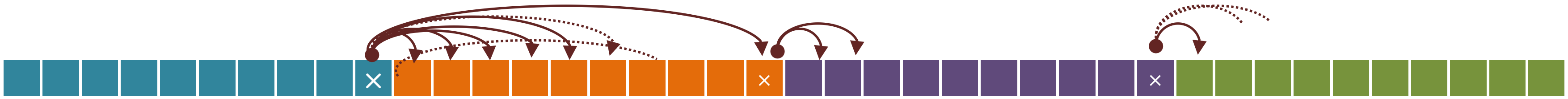
In these few slides, I intend to discuss two different parallel implementations.
I aim to convince you that not all parallelisms are born equal.

In the following, ***N*** is the size of the array and ***n*** indicates the number of threads.

Algorithm I

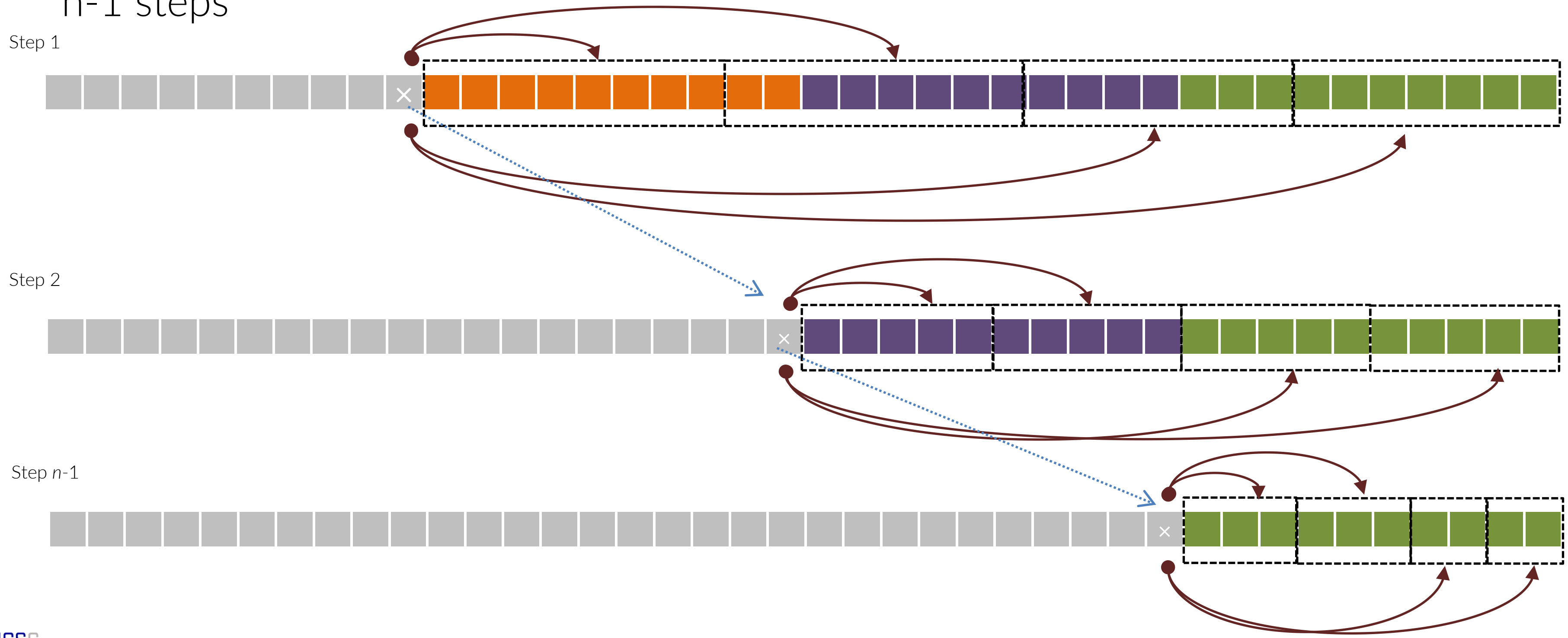
Let’s say that this is our array, of which we want to compute the prefix-sum. An obvious *phase-1* is to divide the array in chunks and compute the partial prefix sum for each chunk

The first chunk is correct, while the others lack the contributions from the previous chunks



Algorithm I

We can solve this by a recurrent subdivision of the work among *all* the tasks, with n-1 steps



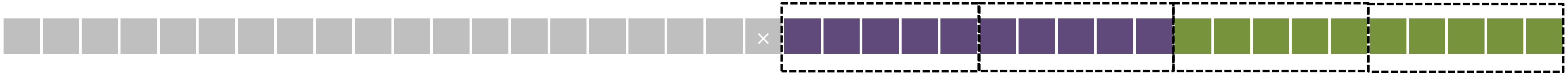
Algorithm I

How many ops are performed in this algorithm ?

Step 0 : $^{N}/_{n} \times n \; - n$ operations (the term -*n* is due to the fact that the first elements of each chunk are untouched in this step

Step 1 : $N - {}^{N}/_{n}$ operations

Step 2 : $N - 2 \times {}^{N}/_{n}$ operations



Step *n*-1 : $N - (n-1) \times {}^{N}/_{n}$ operations

Algorithm I

How many ops are performed in this algorithm ?
We can express that recurrence as

$$N + \left(N - \frac{N}{n}\right) + \left(N - 2 \times \frac{N}{n}\right) + \ldots + \left(N - (n-1) \times \frac{N}{n}\right) \; - \; n =$$

$$\sum_{i=0}^{i=n-1} \left(N - i \times \frac{N}{n}\right) \; - \; n =$$

$$nN - \frac{N}{n} \times \frac{n(n-1)}{2} \; - \; n \; =$$

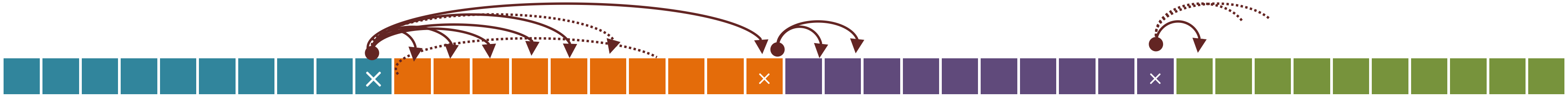
$$nN - \frac{Nn}{2} + \frac{N}{2} - n \; =$$

$$\boldsymbol{\frac{N}{2}(n+1)}$$

Algorithm II

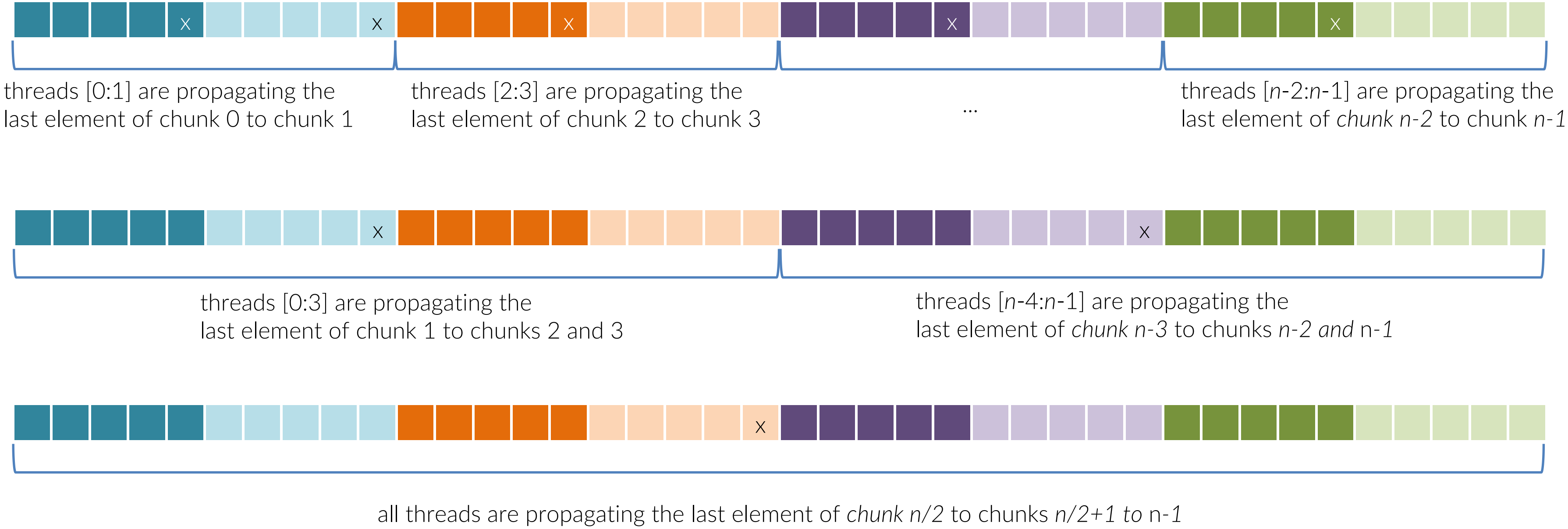
Let’s start from the first *phirst phase* as before

The first chunk is correct, while the others lack the contributions from the previous chunks



Algorithm II

Let’s design a different recurrent subdivision of the work among *all* the workers (we’re using 8 threads instead of 4 to illustrate it)



Note: you can easily adapt it to work with a non-power-of-two number of threads

Algorithm II

How many ops are performed in this algorithm ?
In the step 0 it performs *N*-*n* operations.

In step 1 it performs $N/_n \times n/_2$

In step $\boldsymbol{i}$ it performs $j \times \frac{N}{n} \times \frac{n}{2j}$ where $\boldsymbol{j = i^2}$

Since there will be $\boldsymbol{k = log_2 n}$ steps, we can sum up as

$$N - n + \sum_{i=1}^{k} \left(\frac{N}{2}\right) n =$$

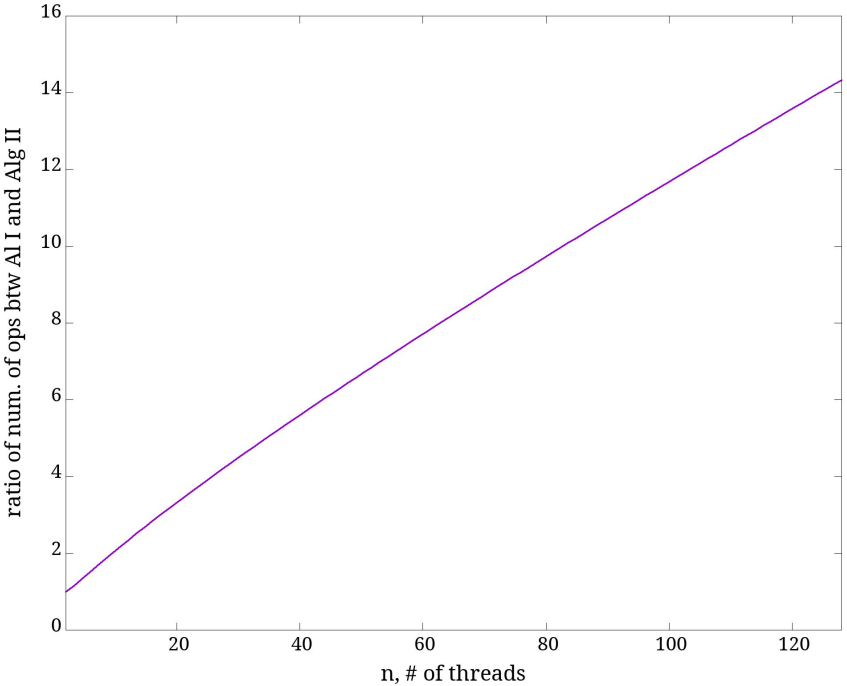
$$N - n + \frac{N}{2} k =$$

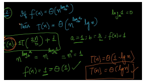
$$N \left(1 + k/_2\right) - n = \boldsymbol{N \left(1 + \frac{log_2 n}{2}\right) - n}$$

Algorithm I vs II

Algorithm II is manifestly superior: the ratio of the ops number goes as (ignoring the -*n* terms)

$$R = \frac{N/_2(n+1)}{N\left(1+\frac{log_2 n}{2}\right)} \approx \frac{n}{log_2 n}$$





Algorithm I vs II

Even more important is the fact that the speed-up of the two algorithms (supposing that the run time goes a #ops/n) has very different behaviour

algorithm 1:

run-time = $\frac{N}{2} \times \frac{n+1}{n}$

speed-up = $T_s(N)$ / $T_p(N,n)$ = $\frac{2n}{n+1} \approx 2$

algorithm 2:

run-time = $\frac{N}{n} \times \left(1 + \frac{log_2 n}{2}\right)$

speed-up = $\frac{n}{1+log_2 n} \approx \frac{n}{log_2 n}$

Memory issues

There may be another (severe) limiting factor to the *scalability* (i.e. the speed-up growth when you increase the number of threads) that is linked to the memory access.

We'll discuss this matter specifically when we'll discuss the threads-memory affinity.