Sparse notes on sorting

Luca Tornatore - I.N.A.F.

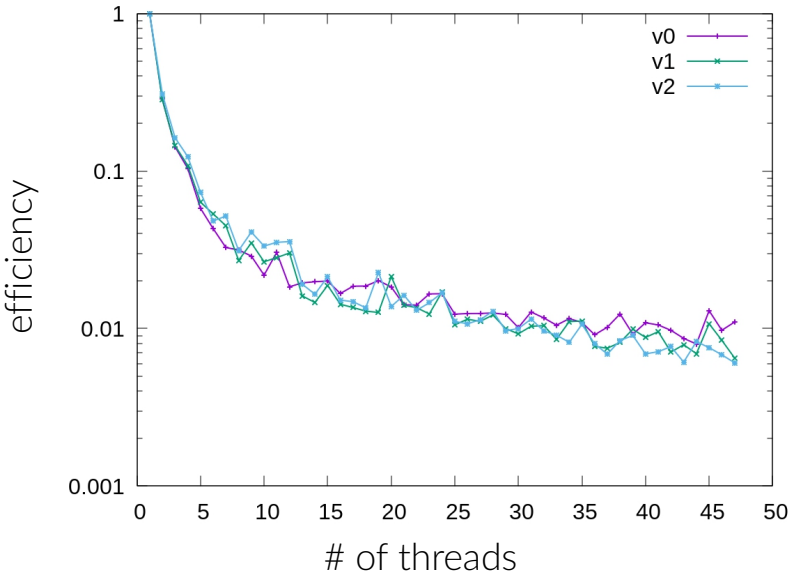
"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2021-2022 @ Università di Trieste

Recap

In the series of lecture about tasks in OpenMP we have discussed a parallel version of the popular *quicksort* algorithm.



We were left with the fact that the efficiency of that implementation is rapidly degrading with the number of threads.

Why is that? Is it due to a somehow bad implementation or is it intrinsic to the parallelism that have been implemented?

Objective

This small lecture is about algorithm analysis and how it is important in general and, particularly, in paralle computation.
As we have seen, the increased algorithmic complexity in parallel codes may lead to unwanted overhead; or, the way in which the parallelism is exposed my actually limit the possible speed-up.

In both cases, a careful algorithm analysis could expose this fact at early time and, as a consequence, the programmer can either spot the bottleneck since the beginning or indentify where the parallelism is lost, then opting for a different algorithm.

Algorithm analysis is not the aim of this course, and hence this is just a tiny an incomplete example with the purpose of raising the point and giving the flavour of the matter.

Mergesort

Let's consider another classical example among the *sorting algorithms*, i.e. the mergesort.
Like *quicksort*, it is an optimal *divide-et-impera* algorithm which subdivides a problem in smaller similar problems and solve them. However, it does so in a bottom-up way instead of top-down like *quicksort*.

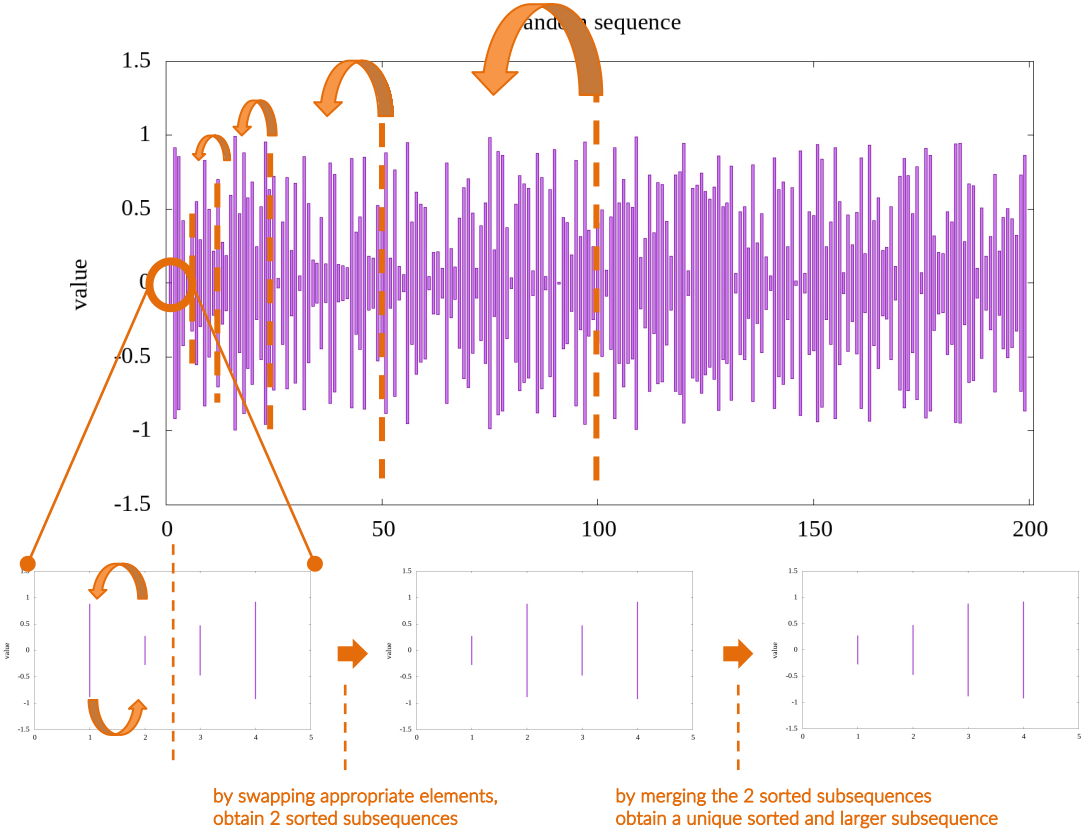
The strategy to *merge* sorted subsequence, and the rational is that starting from the whole set it walks down to pairs of data, which are of course sorted sequence of lenght 1 each, and merge upwards.

Note 1: Mergesort, in its simplest implementation is a stable out-of-place sorting algorithm, whereas quicksort is an unstable in-place one. A sorting is said to be «stable» if it preserves the original order of same-value keys. An in-place algorithm sorts the data set of size N using the same original array, whereas an out-of-place algorithm needs a temporary storage of the order O(N).

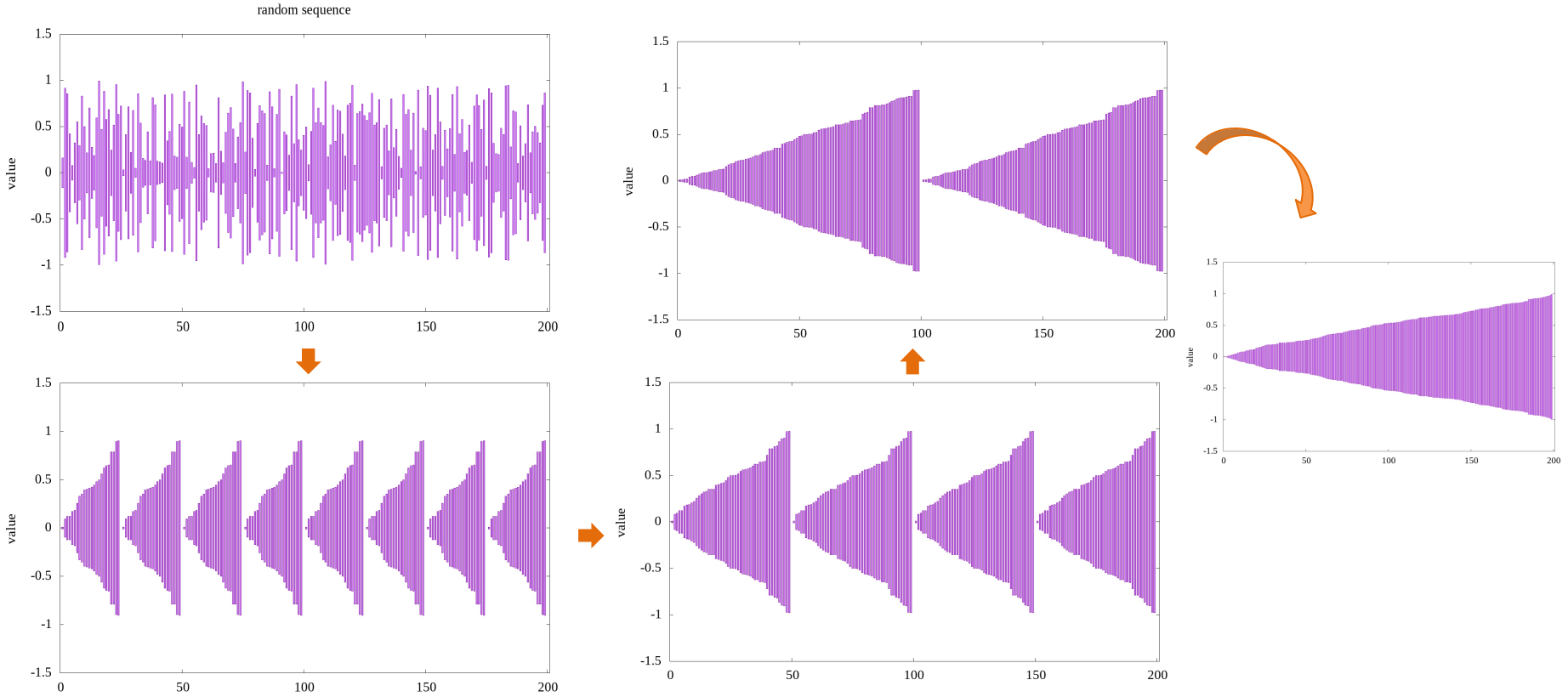
Note 2: what will be stated for the merging sections of mergesort as for the parallelization could be reversed symmetrically for the partitioning in quicksort

Mergesort

Starting from the whole data distribution, the “sort” section of the mergesort algorithm considers recursively smaller sections (tipically sub-dividing by factor of two at each level) down to pairs of elements that are swapped if needed.
The recursive call tree then returns with the “merge” section which merges the progressively larger sorted subsequences.



Mergesort



As an alternative way of describing it, Sedgewick and Wayne ("Algorithms Part I", 2014) offer an effective graphic way to visualize the mergesort process, applying to the alphabetical sorting of the string "mergesort".

On the right you can check the entire call tree.

Proposition F. Top-down mergesort uses between ½ $N \lg N$ and $N \lg N$ compares to sort any array of length N.

from Sedgewick & Wayn, "Algorithms - part I", 2014

```
sort(a, 0, 15)
sort left half
  sort(a, 0, 7)
    sort(a, 0, 3)
      sort(a, 0, 1)
        merge(a, 0, 0, 1)
      sort(a, 2, 3)
        merge(a, 2, 2, 3)
      merge(a, 0, 1, 3)
    sort(a, 4, 7)
      sort(a, 4, 5)
        merge(a, 4, 4, 5)
      sort(a, 6, 7)
        merge(a, 6, 6, 7)
      merge(a, 4, 5, 7)
    merge(a, 0, 3, 7)
sort right half
  sort(a, 8, 15)
    sort(a, 8, 11)
      sort(a, 8, 9)
        merge(a, 8, 8, 9)
      sort(a, 10, 11)
        merge(a, 10, 10, 11)
      merge(a, 8, 9, 11)
    sort(a, 12, 15)
      sort(a, 12, 13)
        merge(a, 12, 12, 13)
      sort(a, 14, 15)
        merge(a, 14, 14,15)
      merge(a, 12, 13, 15)
    merge(a, 8, 11, 15)
merge results
  merge(a, 0, 7, 15)
```

Top-down mergesort call trace

Elementary parallel mergesort

A very simple parallelization of the merge sort is the following.

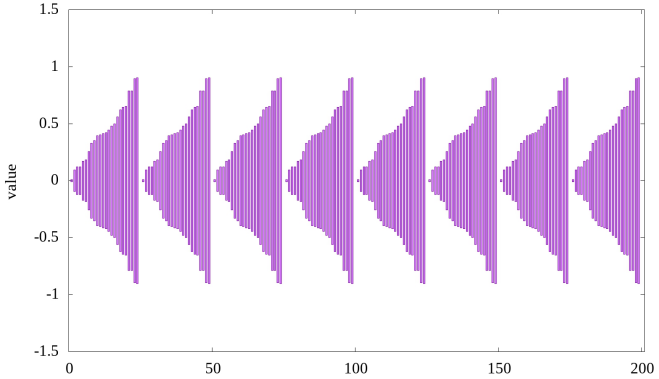
Since we just want to end up with a number of sorted sub-sets to be merged afterwards, we can very naturally distribute the data among *p* processors so that each one independently could sort its own subset, either with the same mergesort or with another sorting routine.

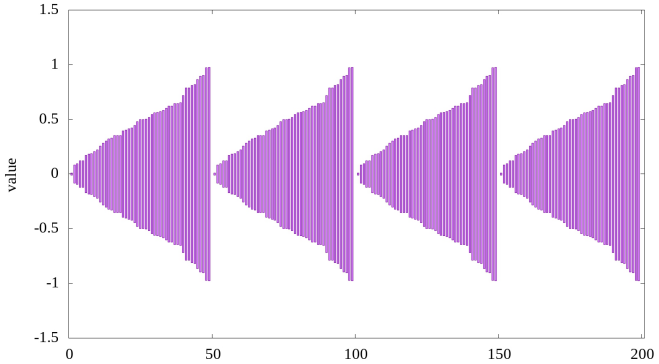
After this step, the resulting *p* sorted sub-sets will be merged pairwise.

What performance do we expect from this procedure?

Elementary parallel mergesort

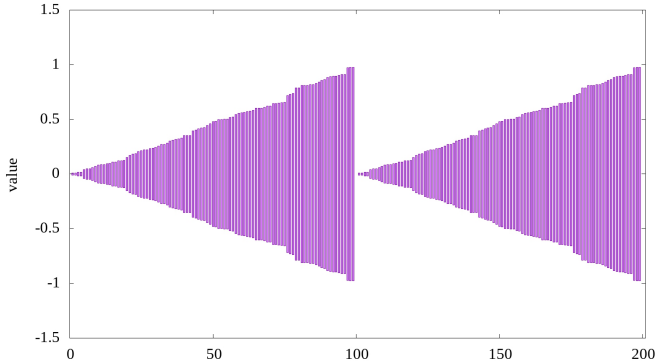
p processes (p=8 in this example) end up with *p* sorted subsequences that must then be merged



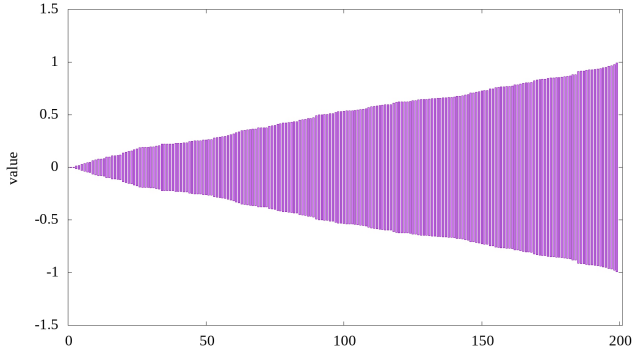


p/2 (i.e 4) processes perform the merge at the same time

p/4 (i.e 2) processes perform the merge at the same time

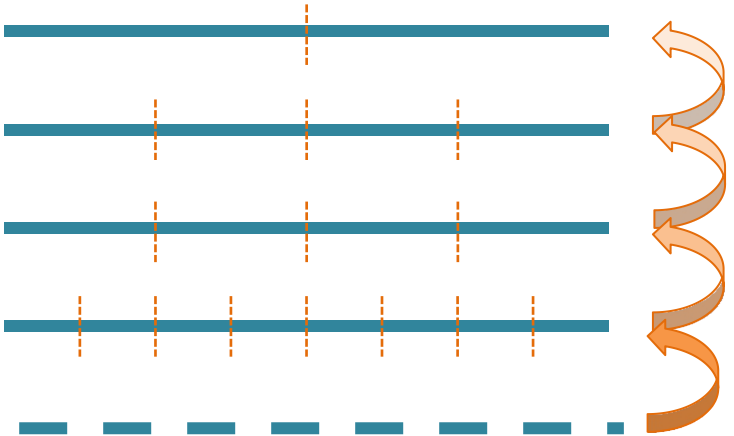


p/8 (i.e. 1) process performs the final merge



Elementary parallel mergesort

If we assume that each process performs ~N/p ×log N/p operations to sort its own chunks *at the same time than the other processes*, how many operations do we expect to be performed for the merging-up phase?



avg. # of compare
& swap operations
at the same time

N

N/2

N/4

N/8

...

$$N \times \sum_{n=0}^{log\, p} \frac{1}{n^2}$$

NOTE: we are assuming a flat random sequence without either skewnesses or clusterings, so that we can assume that in average the *p* chunks will require the same amount of work and that work is *~NlogN*.

Elementary parallel mergesort

1. each process will perform in average

$$\frac{N}{p} \times log\frac{N}{p}$$

operations on its subset

2. to merge the subsets, about ~N operations will be performed, in parallel, so that N will be performed by 1 process to merge the last 2 halves, N/2 by 2 processes to merge 4 quarters in 2 halves and so on.
So we expect

$$\sim N \times \sum_{i=0}^{logp} \frac{1}{i^2} \sim 2N \quad \text{for large enough p}$$

Elementary parallel mergesort

3. the expected speed-up is then

$$\frac{NlogN}{2N + \frac{N}{p} \times log\frac{N}{p}}$$

4. which becomes for a 2-level nested parallelism:

$$\frac{Nlog\,N}{f_1 + \frac{f_2}{p_1} + \frac{N}{p_1 \cdot p_2} log\frac{N}{p_1 \cdot p_2}}$$

basically p_1 processes take the place of the p processes mentioned so far, spawning p_2 processes which in turn sort $N/p_1/p_2$ subsequences.
You may think to p_1 as the number of sockets on a node or tas the number of nodes in a multi-nodes pool.

f_x is the true value of $\sum_{i=0}^{[logx]} \frac{1}{i^2}$ or p_1 and p_2

Elementary parallel mergesort

$$\frac{N log\, N}{f_1 + \frac{f_2}{p_1} + \frac{N}{p_1 \cdot p_2} log \frac{N}{p_1 \cdot p_2}}$$

for N»1

$$\frac{log\, N}{\frac{f_2}{p_1} + \frac{1}{p_1 \cdot p_2} log \frac{N}{p_1 \cdot p_2}}$$

$$\sim [1 - 2] log\, N$$

typical figures for p_1, p_2 are ~ 4 , 16-32 so that f_x is of the order 1.5-2

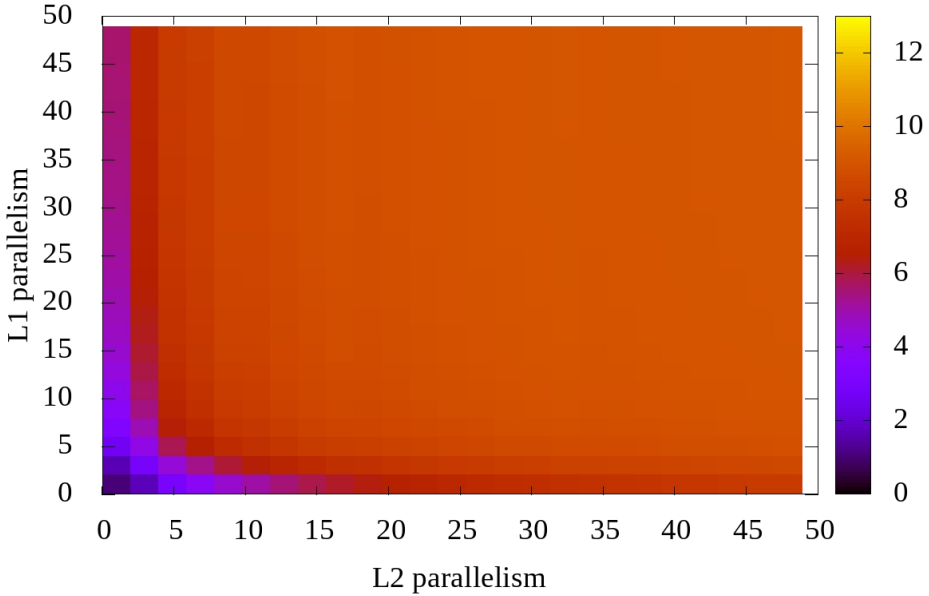
hence, $p_1 p_2$ ~ 100 and f_2/p_1 ~ 1/4

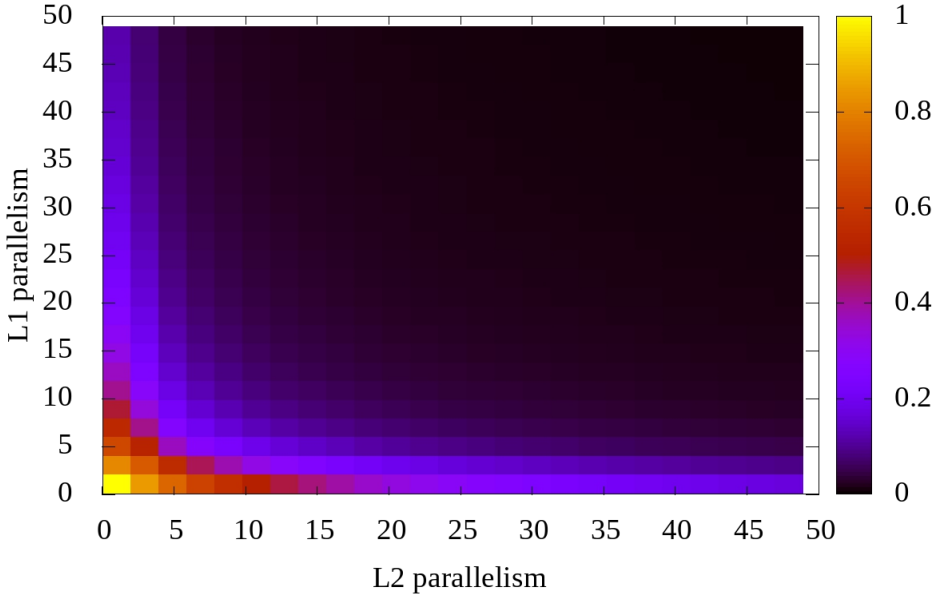
So we are loosing parallelism and the possible speed-up saturates, because the merging operations are intrinsically serial and are performed by progressively fewer processes on proportionally larger data sub-sets.

Elementary parallel mergesort

Heat map of expected speed-up (left) and parallel efficiency (right) for N = 10^8

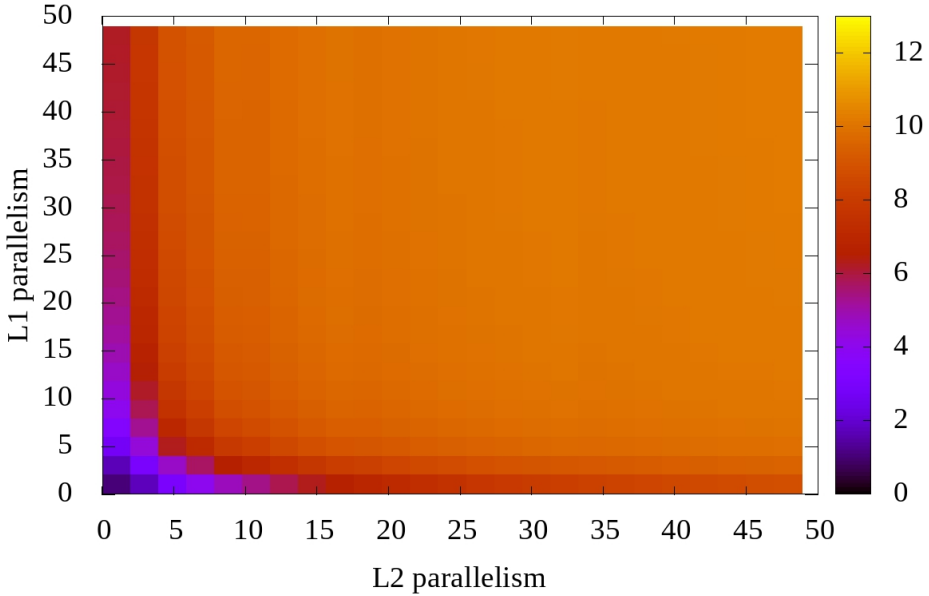
The y-axis ("L1 parallelism") corresponds to the number p_1 of processes whereas the x-axis ("L2 parallelism") corresponds to p_2 in the notation used in the previsou slide.
Note that the upper-right region, approximately $p_1 \times p_2 >$ ~100 is not realistic in every used platform.

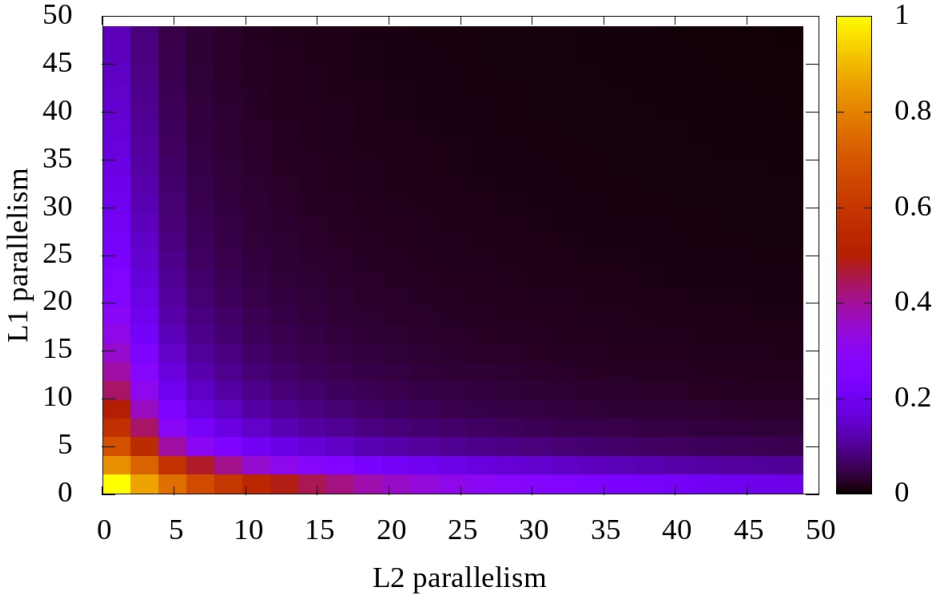




Elementary parallel mergesort

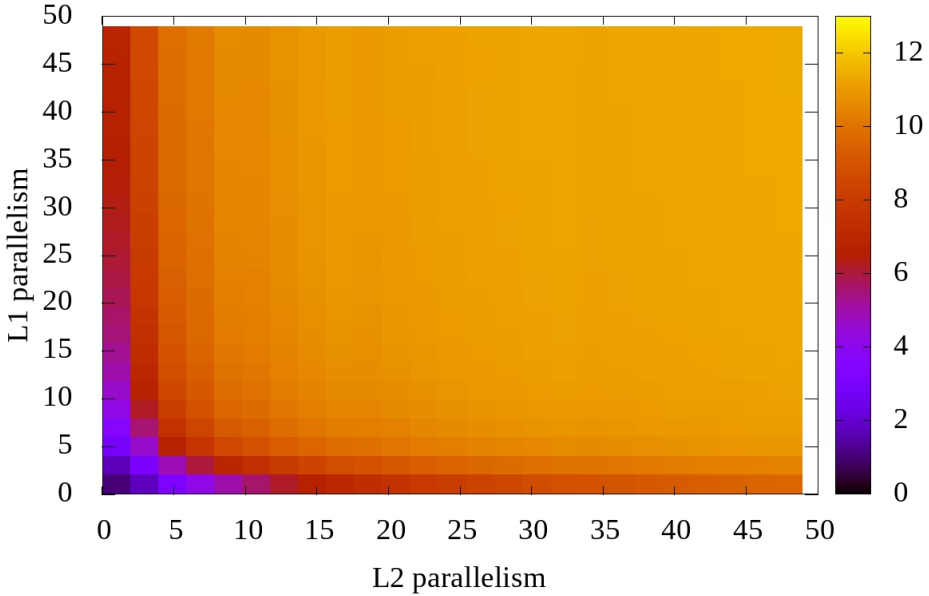
Heat map of expected speed-up (left) and parallel efficiency (right) for N = 10^8

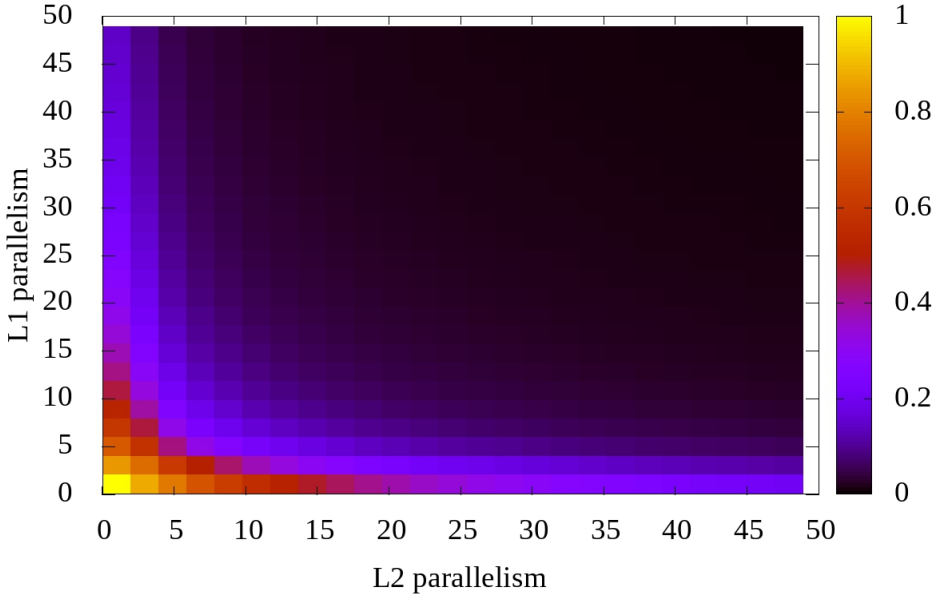




Elementary parallel mergesort

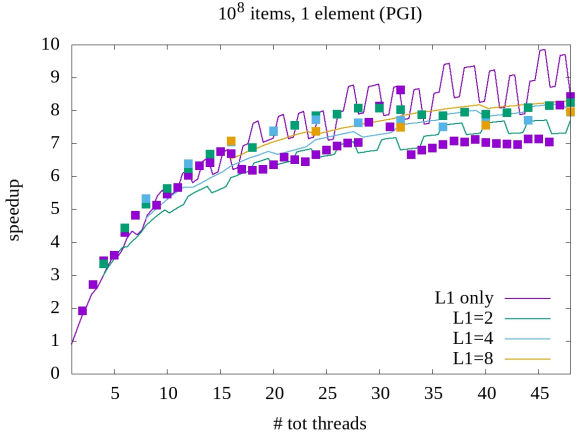
Heat map of expected speed-up (left) and parallel efficiency (right) for N = 10^8

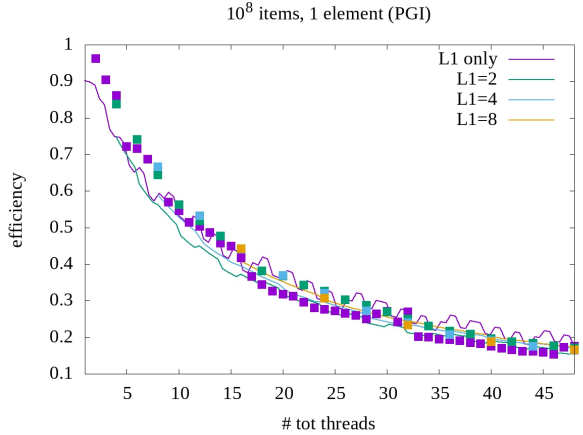


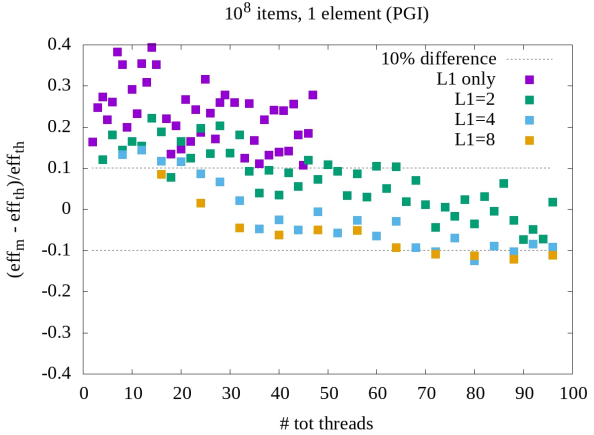


Elementary parallel mergesort

Measured speed-up *(left)*, parallel efficiency *(central)* and deviation from expected efficiency *(right)* for N=10^8



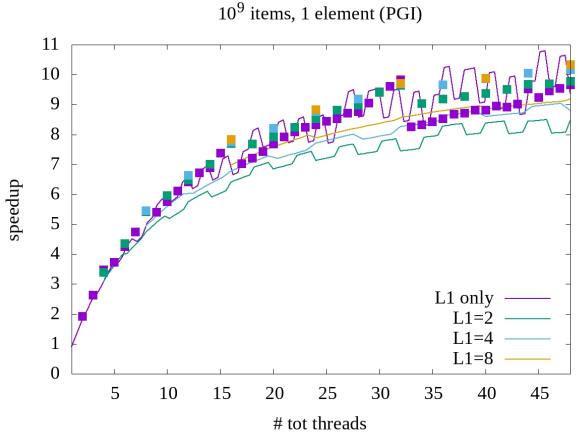


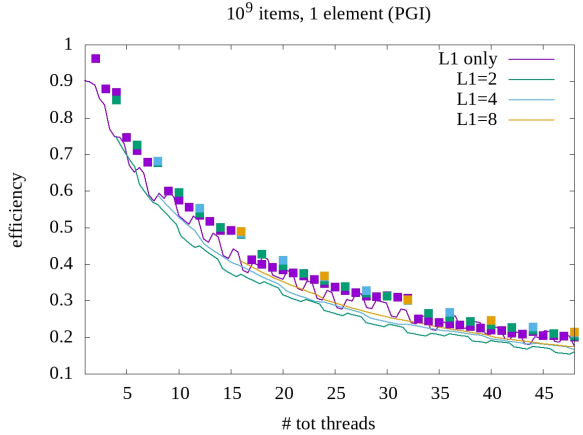


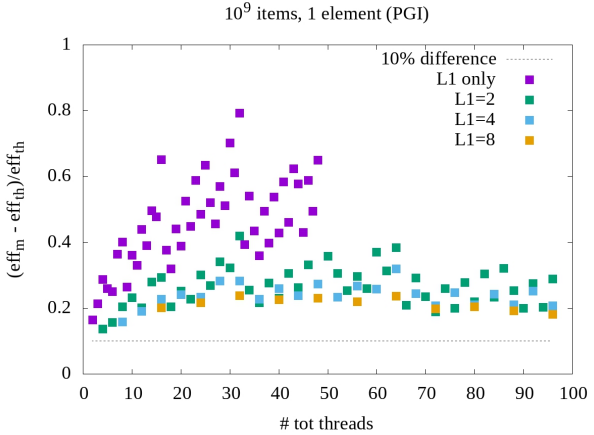
Elementary parallel mergesort

Measured speed-up *(left)*, parallel efficiency *(central)* and deviation from expected efficiency *(right)* for N=10^9

note: the PGI compiler on the platform used for this case issues a code significantly more performant than gcc whose measured efficiency lies within -10%:10% from the theoretical one







Elementary parallel mergesort

Exactly as we have seen for the *quicksort*, some algorithmically simpler - asymptotically non optimal but pragmatically more performant for small-enough data sets - sorting algorithm can be used when the recursive descent arrives at “small-enough” chunks of data.
That proves to be more efficient than continuing the sort descent down to data pairs and then merging up from that level.

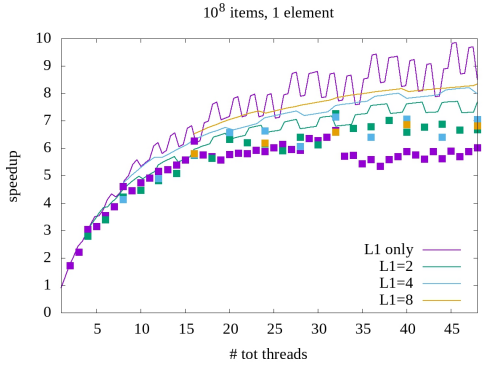
Even that choice, which algorithm to use, may affect significantly the overall result since it impacts the larger $log\ N/(p_1 p_2)$.

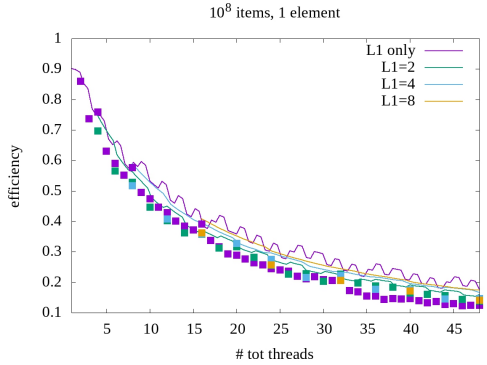
In the next slide we show the difference between an insertion-sort and a more efficient shell-sort.

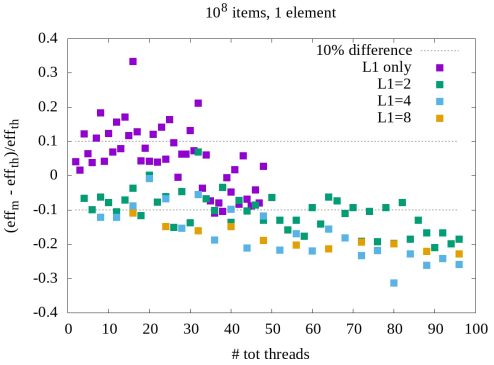
Elementary parallel mergesort

Measured speed-up *(left)*, parallel efficiency *(central)* and deviation from expected efficiency *(right)* for N=10 and 2 different algorithms for sorting of “small-enough” chunks of data

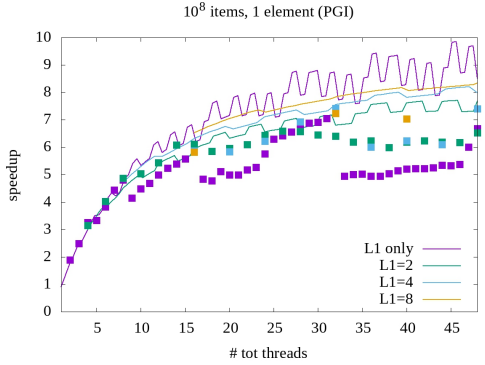
shellsort

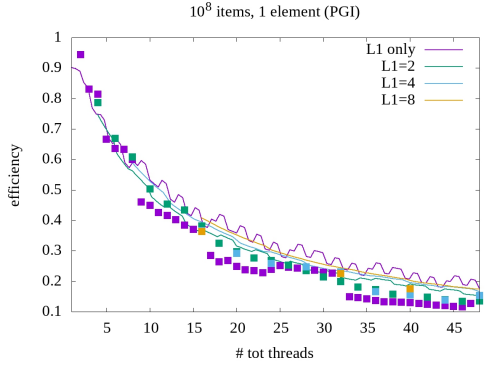


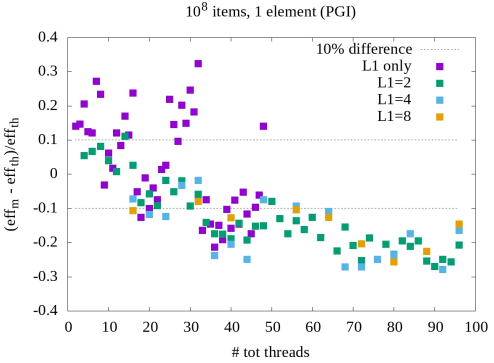




insertionsort

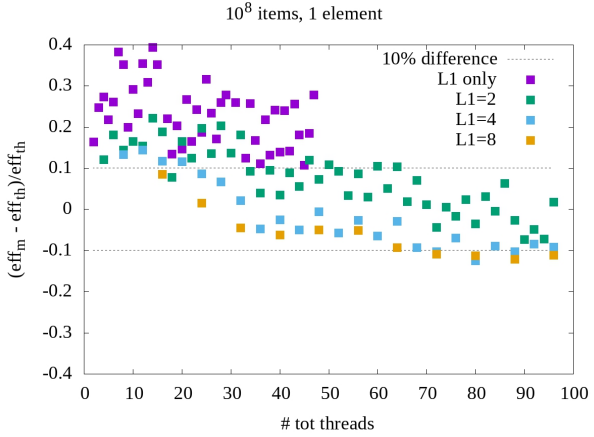




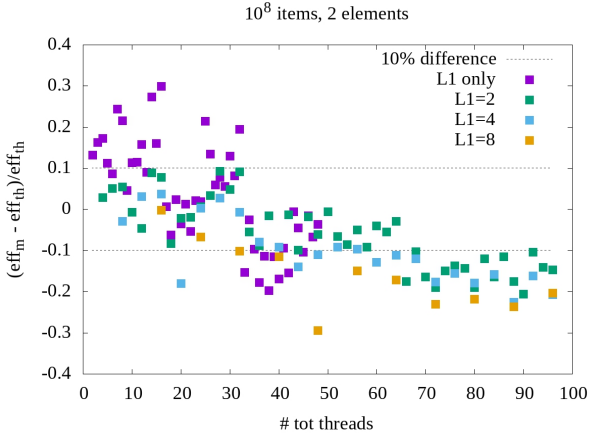


Elementary parallel mergesort

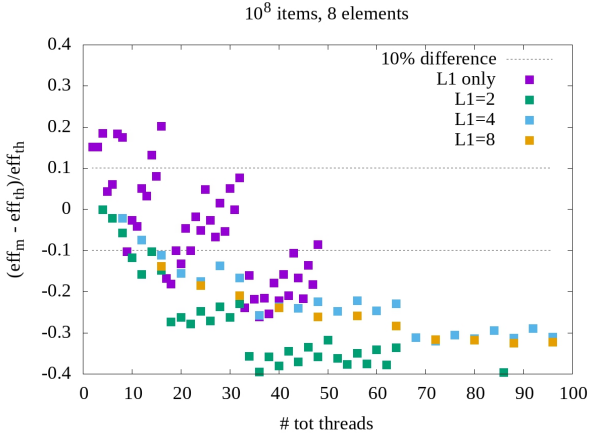
One additional element that impacts on the performance of a sorting algorithm that moves the data of the data set is **the data size of each item**. Here we contrast the efficiency measured for items made of 1, 2 and 8 double respectively.



8 bytes



16 bytes



64 bytes

that’s all, have fun