UNIVERSITÀ DEGLI STUDI DI PADOVA

Parallel Computing 2021/2022

HyperQuicksort

A Parallel QuickSort on a Hypercube

Federico Michelotto 1218322

1 Introduction

The quicksort algorithm is a popular sorting algorithm published in 1961 by C. A. R. Hoare [3]. Quicksort implements a divide-and-conquer paradigm to sort the original sequence given in input. The three steps of the divide-and-conquer approach for sorting a generic sequence $A[p...r]$ are described below:

Divide: Given an array $A[p...r]$, if $p - r > 1$ select a pivot value among the values in the array and *partition* (rearrange) $A[p...r]$ into two sub-sequences $A[p...q-1]$ and $A[q+1...r]$ such that all elements of $A[p...q-1]$ are less than or equal to the pivot, and all elements of $A[q+1...r]$ are strictly greater than the pivot. If instead $p - r = 1$ nothing has to be done since there is only one value in the array, and therefore the procedure ends.

Conquer: Sort the sub-arrays $A[p...q-1]$ and $A[q+1...r]$ applying recursively the quicksort algorithm.

Combine: When the conquer stage is terminated, the sub-arrays in $A[p...r]$ are sorted, thus also the input array $A[p...r]$ is now sorted and therefore nothing more has to be done.

Starting from the sequential quicksort algorithm just described, in the next section will be introduced the parallel approach implemented, while in section 3 and 4 will be presented the experimental results obtained and the related conclusions.
The parallel program has been implemented in C, exploiting the Open MPI library [2], and tested on the computing platform CAPRI (Calcolo ad Alte Prestazioni per la Ricerca e l'Innovazione) [5]. The source code can be found at `github.com/federicomichelotto/hyperquicksort` with an example on how to run it.

2 Parallel Approach

As we have seen in section 1, the quicksort algorithm implements a divide-and-conquer strategy; a similar divide-and-conquer approach is also used in the hyperquicksort algorithm [4]. Let us suppose we have $P = 2^k$ processors 0,1, ..., $P-1$, where k is a positive integer. The hyperquicksort algorithm begins with an initialization phase in which the input data is evenly distributed among the P processors and then, each sub-sequence is sorted in parallel. When the initialization is concluded, the divide-and-conquer procedure of the hyperquicksort begins. Such procedure can be described as follows:

Divide: Given a set of processors $p, p+1, ..., r-1$, such that $r - p = 2^l$. If $l > 0$ partition the processors in two halves, lower-half and upper-half, such that the lower-half contains the nodes $p, p+1, ..., 2^{l-1}-1$, and the upper-half contains the nodes 2^{l-1},$2^{l-1}+1, ..., r-1$, and such that the i-th processor in the lower-half communicates only with the i-th processor in the upper-half and vice versa.
If instead there is only one processor ($l = 0$), nothing has to be done since all the values in the processor are already sorted, and therefore the procedure stops.

Conquer: Pick a pivot value such that each processor in the lower-half sends only the values greater than the pivot to its corresponding upper-half processor, and vice versa, each processor in the upper-half sends only the values smaller than or equal to the pivot to its corresponding lower-half processor. After that, each processor merges the remained values with the ones received by its counterpart and then the hyperquicksort algorithm is applied recursively on both halves.

Combine: Let us denote with $A[i]$ the sequence of values associated to the i-th processor. Then after $\log_2(r-p) = l$ steps, we have that for $p+1 \leq i \leq r-2$, all the values in $A[i]$ are greater than or equal to all the values in $A[i-1]$, and smaller than or equal to all the values in $A[i+1]$. And since $\forall\ i = p, p+1, ..., r-1$, $A[i]$ is always sorted by construction, then, the sequence $A[p]A[p+1]\cdots A[r-1]$ represents the input sequence totally sorted, and therefore no more work is needed.

Thus the hyperquicksort algorithm on $P = 2^k$ processors requires exactly $\log_2 P = k$ steps. The divide-and-conquer approach described above is depicted in fig. 1 and as it can be seen, the hyperquicksort implements the descend paradigm of the hypercube, hence the name hyperquicksort.
It can be easily seen that if we have $P = 2^k$ processors, the minimum number of steps required to combine P equal length sorted sub-sequences through parallel merge operations (assuming to merge only two lists at a time) is $\log_2 P = k$. The naive approach consist of merging the sub-sequences in a balanced way, such that, after $\log_2 P = k$ steps we have the input sequence totally sorted. With such approach, however, the number of active processors is halved at every iteration. Therefore, with respect to this naive approach, the hyperquicksort algorithm aims to decrease the overall execution time by reducing the number of inactive processors during the sorting procedure.

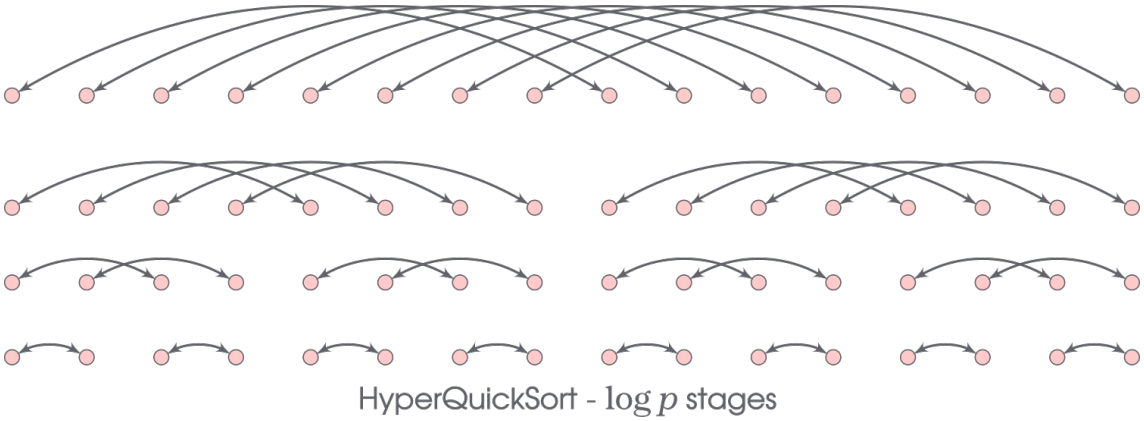

Figure 1: Descend paradigm of the hypercube with 16 processors. Credit to Professor Hari Sundar, University of Utah.

2.1 Implementation Details

As described previously, during the initialization phase each processor sorts its sequence individually. In my implementation I decided to simply use the sequential quicksort algorithm. In this way, when only one processor is employed, it is directly executed the sequential quicksort algorithm. The sequential quicksort algorithm has been implemented following the divide-and-conquer procedure described in 1. The only critical aspect not discussed previously is how to choose the pivot. In my implementation the pivot is chosen as the median of the three values that are at the beginning, at the middle, end at the end of the sequence.
As for the sequential quicksort, the selection of the pivot is crucial also for the hyperquicksort algorithm. Every time a group of processors is halved, a pivot must be chosen. To choose the pivot, each group of processors does the following operations:

1. Each processor sends its median value to the master node.
2. The master node computes the median value of the list of medians just received, and it broadcasts this value to all the nodes in its partition.

To implement the descend paradigm of the hypercube, at every iteration it was used the MPI function *MPI_Comm_split()* to appropriately partition the processors.
To exploit the full-duplex communication capability of CAPRI, each processor sends its list of values to its counterpart asynchronously, in this way, each processor can concurrently receiving the values from its counterpart without waiting the end of the outgoing transmission. Then, since the values in each node are kept sorted, the selection of the values greater than a given pivot is implemented using a binary-search approach instead of a linear search.
The input list that we want to sort must be stored as a file, where in the first row there must be the size of the list, and then, there must be a number of rows equal to the size of the list such that in each row there is exactly one value. Values are assumed to be 64 bit unsigned integer.

3 Results

To study the performances of my implementation of the hyperquicksort algorithm, the following metrics have been evaluated:

- Execution time as the number of cores increase.
- Speedup[1] as the number of cores increase.
- Execution time as the size of the input increases.

The above metrics have been evaluated on instances with 2^{23}, 2^{24}, 2^{25}, 2^{26} and 2^{27} values, and using 1, 4, 16 and 64 cores. In particular, for each input size have been generated 5 random instances, such that each measurement is obtained as the average over 5 trials. All measurements have been repeated twice with different compiler optimization levels, once with the optimization level -O0 and once with the optimization level -O2. The plots generated by the obtained results are shown below. Comments on these results will be discussed in section 4.

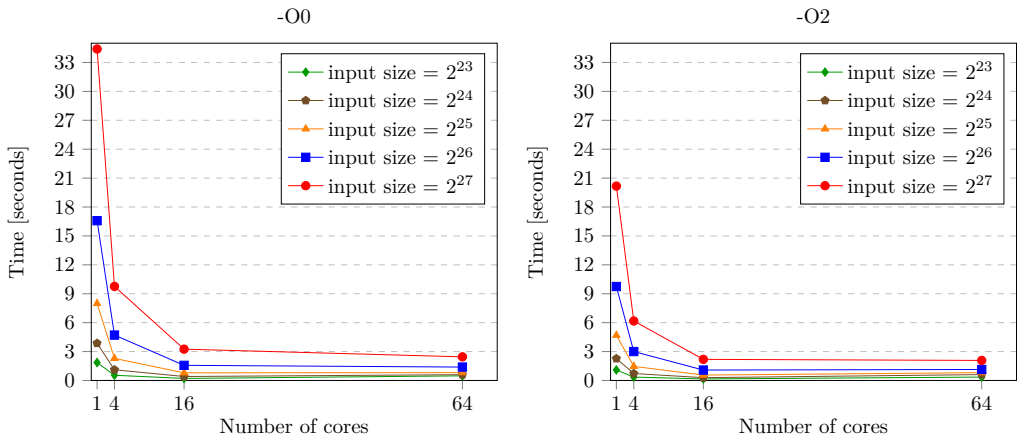

Figure 2: Execution times as the number of cores increase.

[1]Speedup $S(p) = T(1)/T(p)$, where $T(p)$ is the execution time using p processors.

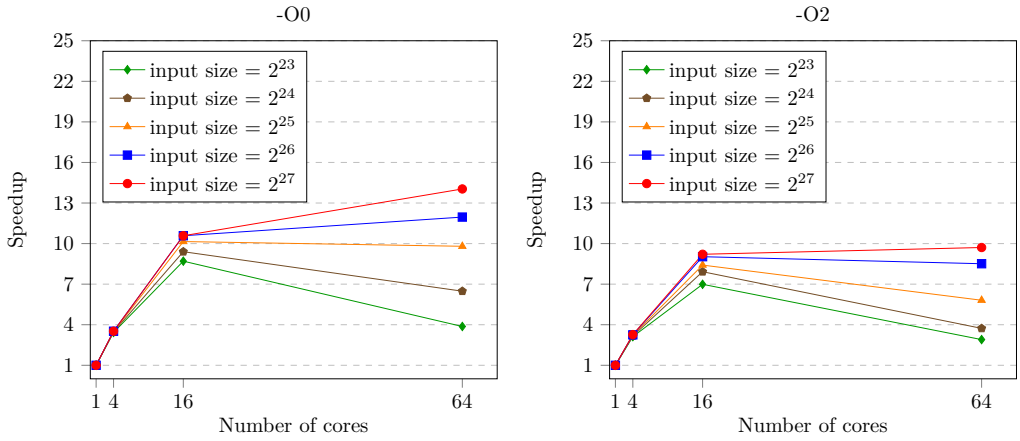

Figure 3: Speedup as the number of cores increase.

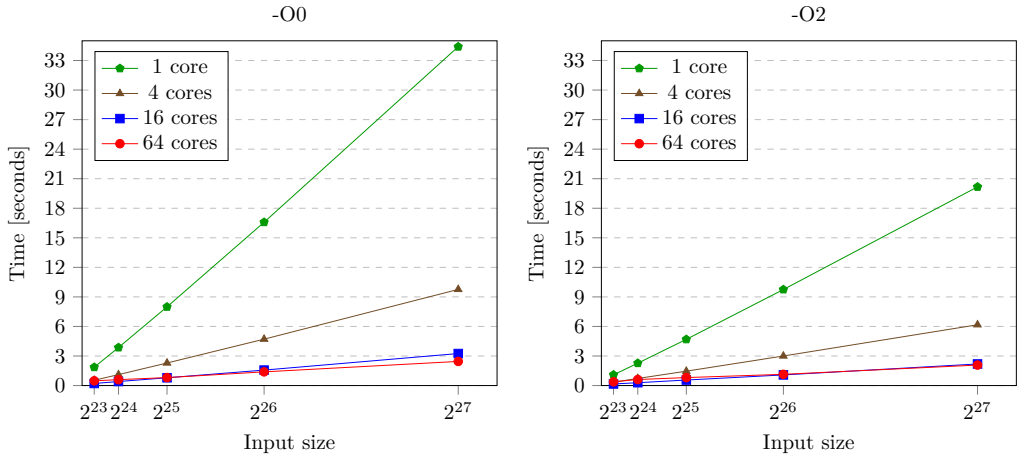

Figure 4: Execution times as input size increases.

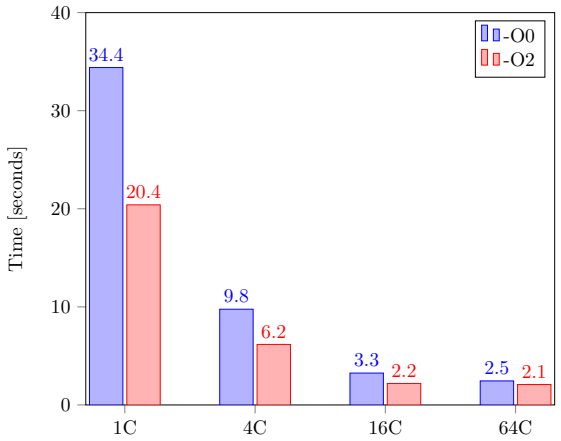

Figure 5: Execution times using different optimizations levels on inputs with 2^{27} values.

4 Conclusions

In figure 2 are represented the execution times in function of the number of cores. As expected, since the instances tested are not very small, increasing the number of cores the execution times decrease, however, to study the scalability of the algorithm, the speedup must be analyzed.
From figure 3 we can see that the speedup grows significantly up to 16 cores. Instead, when passing from 16 to 64 cores, the speedup increases very slowly and only using the largest instances, whereas, with the smallest instances the speedup stops growing or even begins to drop.
The fact that when using 64 cores the speedup values are so different when tested on different input sizes, it suggests to us that the instances tested are too small to fully exploit all 64 cores, and therefore larger instances would be needed to appropriately measure this metric.
Figure 4 shows how execution times are affected by the increase of the input size. How it can be seen, the execution times appear to grow linearly in all configurations tested.
Finally, in figure 5 it can be observed the significant improvement derived from the usage of the optimization level -O2 over the optimization level -O0, using instances with 2^{27} values.
As side note, I also prepared the tests for the 128 cores configuration, unfortunately, due to frequent system's errors on CAPRI, I was not able to successfully run these tests.

Bibliography

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[2] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

[3] C. A. R. Hoare. "Algorithm 64: Quicksort". In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782. DOI: 10.1145/366622.366644. URL: https://doi.org/10.1145/366622.366644.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. ISBN: 0071232656.

[5] *University of Padova Strategic Research Infrastructure Grant 2017: "CAPRI: Calcolo ad Alte Prestazioni per la Ricerca e l'Innovazione"*.