Università degli Studi di Padova
Department of Information Engineering

Parallel Computing
Final Project

Parallel quicksort: OpenMPI implementation and analysis

Author
Luca Badin
1242060
luca.badin.1@studenti.unipd.it

Teachers
Gianfranco Bilardi
Paolo Emilio Mazzon

August 2022

Introduction and sequential algorithm

This project's objective is to devise and implement a parallelization strategy for quicksort, a popular sorting algorithm. In quicksort, we seek to sort an input sequence by recursively picking a pivot element, dividing the input between two subinstances to be recursively sorted; then, the output sequence is simply the concatenation of the lower half, the pivot and the upper half. The base cases correspond to instances of size 1 (trivially a sorted sequence) and 2 (sorted with a single comparison).

The "best choice" of pivot element is clearly the median (or, more precisely, the $n/2$th order statistic), as it ensures that the two halves hold exactly the same number of elements, which produces a balanced recursion tree with the smallest possible number of levels.
However, it's not computationally feasible to actually compute the median: the best way to do so has linear complexity, but huge constants that make it unusable in practice (and is additionally impossible to implement in-place). Picking an arbitrary element instead yields a worst-case quadratic algorithm, which is no better than insertion sort's worst-case.
Therefore, quicksort is typically implemented with a random choice of pivot element: this ensures that, with high probability, the balancing between halves will be weaker, but still good enough to ensure the algorithm behaves much more similarly to the best case than to the worst case (polylogarithmic average running time).

Implementation

The implementation of sequential quicksort was fairly standard, though as a point of proper programming, efforts were made to make it use constant additional memory (yielding an in-place solution, or almost so); the partitioning happens by scanning the input for all elements smaller than the pivot and, if such an element is found:

- swap it to the boundary position of a prefix composed of only elements smaller than the pivot;
- incrementing the index of the boundary position;
- updating the pivot's index, if the pivot was the previous boundary element that has now been swapped out of the prefix.

The pivot is then swapped after the boundary, and by construction the input is now composed of a prefix of elements smaller than the pivot element, the pivot element itself, and a suffix of remaining, greater elements. The recursion then proceeds separately on the prefix and suffix.

Algorithm 1 Pseudocode for sequential quicksort.

procedure QUICKSORT(S, lb, ub)
  if $|S| = 1$ **then**
    return S ▷ Base case (1)
  end if
  if $|S| = 2$ **then**
    return $\{\min S, \max S\}$ ▷ Base case (2)
  end if
  $pivot\_index \leftarrow$ RANDOM(lb, ub) ▷ Divide: Pick a pivot
  PARTITION($S, pivot\_index, lb, ub$) ▷ Divide: Partition in place
  QUICKSORT($S, lb, pivot\_index - 1$) ▷ Recurse: Recursively sort lower half
  QUICKSORT($S, pivot\_index + 1, ub$) ▷ Recurse: Recursively sort upper half
end procedure ▷ No conquer needed: solution is already sorted as returned

Parallel algorithm

Algorithmic approach

The strategy chosen to parallelize quicksort is as follows:

- Equally split and scatter the instance among the p computing nodes;
- Have each process locally sort its own subinstance, with the same sequential quicksort algorithm developed previously;
- Gather the sorted sequences at processor 0;
- Efficiently merge the sorted sequences.

Algorithm 2 Algorithmic approach for parallel quicksort.

if rank $= 0$ **then**
  $S \leftarrow$ input
  MPI_BCAST($|S|$) ▷ Ensure all nodes know the input size
end if
MPI_SCATTER(S, s)
$s \leftarrow$QUICKSORT(s) ▷ Locally sort subinstance of size $|S|/p$
MPI_GATHER(S, s)
if rank $= 0$ **then**
  output $\leftarrow$ MERGE($S = \{s_1 \dots s_p\}$)
end if

Efficient merging

Efficient merging is key to not squandering the benefits of subinstance parallel sort. Indeed, this strategy has an additional complication with regards to, say, the parallel mergesort that was seen in class: we have to merge as many sorted sequences as there are computing nodes, p, instead of just 2.

A straightforward solution might be to extend the "classic" binary sorted merging algorithm, so that it keeps track of p indices, finds the minimum across all p sequences, copies it and advances the correct index, until all sequences are exhausted. This, as I will go on to describe, is not the best way to do it sequentially: if in each iteration we scan p arrays and the instance has length n (hence, n iterations), the merging has complexity $\Theta(pn)$.

The approach the project ultimately adopted was to adopt a divide and conquer merging strategy:

Algorithm 3 Divide and conquer merging strategy. BinaryMerge(S,T) is a standard $O(|S| + |T|)$ merging algorithm.

procedure Merge($\{S_1 \dots S_n\}$)
  if $n = 1$ **then**
    return S_1
  end if
  if $n = 2$ **then**
    return BinaryMerge(S_1, S_2)
  end if
  S_A = Merge($\{S_1 \dots S_{n/2}\}$)
  S_B = Merge($\{S_{n/2+1} \dots S_n\}$)
  return BinaryMerge(S_A, S_B)
end procedure

In practice, this is much faster, as it requires fewer comparisons.

Parallel merging

It's evident that further speedups might be achievable if only we could parallelise the merging operation. The calls to Merge form a recursion tree where siblings do not share any data, and can therefore be processed in parallel. Much like in our analysis of mergesort, it may be possible to parallelise it so that each level of the recursion tree is solved in parallel.

Unfortunately, though, I ran into dynamic memory issues that I was unable to debug, and as such my code for parallel merging is still included but is not functional as it fails at runtime. The code for parallel quicksort includes an `ADVANCED_MERGE` definition that can select between naive merging (0), divide and conquer merging (1) and parallel merging (2).

Results

All final tests were conducted on the CAPRI platform, on self-generated (but identical) datasets of different sizes: 16M, 32M, 64M and 128M (respectively, 2^{24}, 2^{25}, 2^{26} and 2^{27} elements). Some tests involved even greater datasets of sizes 256M and 512M.

The parallel code was locally debugged on 4 processors, and the CAPRI tests involved 2, 4, 8, 16 and 32 nodes. All tests were conducted with 512M local memory for each node, except for 128M on 2 and 4 nodes (1G) and for more massive datasets (1.5G).

Sequential quicksort benchmarks

For the sequential code, the only measurement taken was the start-to-end time taken by the algorithm to complete, t_b (base time). The time grows a bit more than linearly with respect to the size of the instance, which is consistent with the well-known average case complexity of randomized quicksort, $O(n \log n)$.

Sort base time (t_b) [ms]					
16M	32M	64M	128M	256M	512M
2384	4876	9914	20792	42833	87967

Table 1: Benchmark results for sequential quicksort.

Parallel quicksort benchmarks

For the parallel code, three time measurements were taken: t_s, the time to locally sort the instance (in practice, identical across processes), t_m, the time to merge the sorted sequences, and t, the whole scatter-to-merge running time.

Node	Local sort time (t_s) [ms]				Merging time (t_m) [ms]			
count	16M	32M	64M	128M	16M	32M	64M	128M
2	1168	2418	4988	10355	127	252	499	994
4	560	1164	2435	5042	255	503	1004	1991
8	269	559	1179	2433	406	759	1541	2984
16	129	269	562	1179	578	1031	2031	4087
32	62	130	272	574	703	1334	3252	9090

Table 2: Local subinstance sort time and merging time for different instance size-node count combinations. Lower is better.

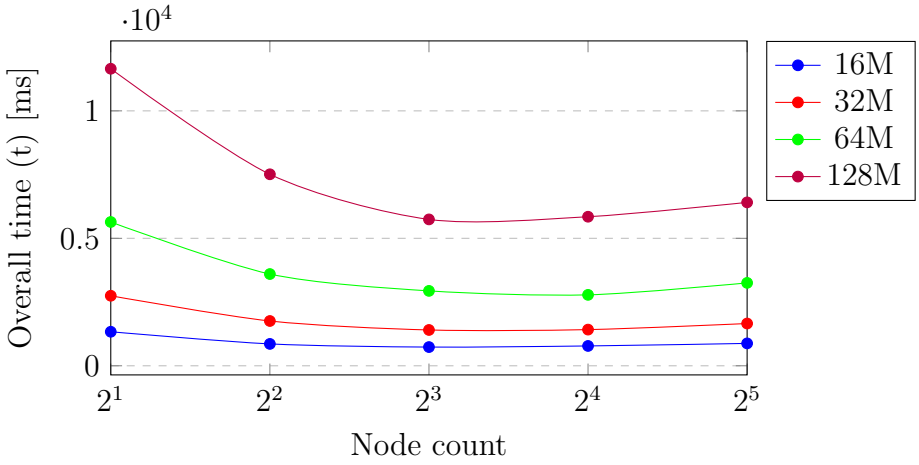
With the same overall instance size, local sort time decreases with more processing nodes, as subinstance sizes get smaller. Conversely, more processing nodes mean more sequences to merge and an increase in merge time. Meanwhile, each combination with matching subinstance size sorts all subinstances in roughly comparable time, as can be seen in each left-to-right diagonal of the local sort time portion (the values are, naturally, analogous to the sequential quicksort case for the respective instance sizes).

Node count	Overall time (t) [ms]				Speedup (S=t_b/t)			
	16M	32M	64M	128M	16M	32M	64M	128M
2	1335	2747	5641	11652	1.78	1.77	1.75	1.78
4	858	1757	3599	7512	2.77	2.77	2.75	2.76
8	**734**	**1405**	2938	**5742**	**3.24**	**3.47**	3.37	**3.62**
16	778	1419	**2782**	5847	3.06	3.43	**3.56**	3.55
32	878	1658	3252	6410	2.71	2.94	3.04	3.24

Table 3: Benchmark results for parallel quicksort. Best result for an instance size in bold. For time, lower is better; for speedup, higher is better.

Scalability

Therefore, this approach does not get indefinitely better as more nodes are added, but a tradeoff needs to be struck between too few nodes (too big subinstances, slower local sort) and too many nodes (too many subinstances, slower merge). It seems like the goldilocks area lies between 8 and 16 nodes, after which the speedup drops off.

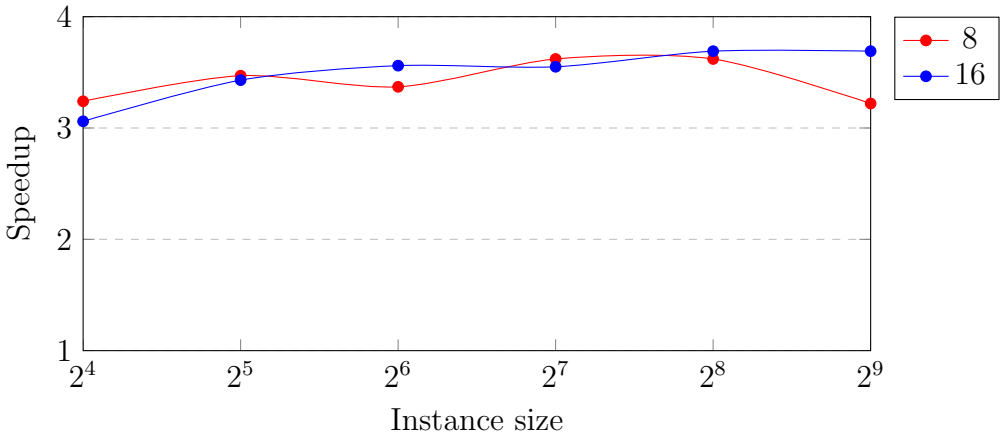


Speedup and instance size

The speedup is not only variable in node count, but also in instance size. Consider the following results on even bigger datasets: the speedup seems to inch upwards, but (even accounting for natural differences between runs) seems to roughly remain on the same values.

Node count	Overall time (t) [ms]		Speedup (S=t_b/t)	
	256M	512M	256M	512M
8	11821	27256	3.62	3.22
16	11594	23812	3.69	3.69

Table 4: Benchmark results for parallel quicksort on a selection of best performing node counts, and on massive datasets. For time, lower is better; for speedup, higher is better.



Communication

The time to gather and scatter was not measured, but we can informally say that the combined communication time is roughly $t_c \simeq t - t_s - t_m$. What follows is again a selection of values of t_c for the best performing node counts, along with the computation over communication ratio. On average, in the best performing cases, communication seems to account for less than 7% of the total time.

Node count	Comm. time (t_c) [ms]				Comp. ratio ($(t_s + t_m)/t$)				
	16M	32M	64M	128M	16M	32M	64M	128M	AVG
8	59	87	218	325	91.96%	93.81%	92.57%	94.33%	**93.16%**
16	71	119	189	581	90.87%	91.61%	93.20%	90.06%	**91.43%**

Table 5: Communication times (lower is better) and computing over communicating ratio (higher is better)

Conclusions

The developed parallelisation strategy yields a speedup that is not proportional to the number of cores. Despite this, I would not call it a failure: as we learned in class, some algorithms are just not well suited to being parallelised without rethinking the problem entirely (at which point, the resulting algorithm might bear little resemblance to quicksort anyway). I would claim the strategy is sufficiently robust and sound to provide appreciable gains, within our means.

As I hinted to earlier, the only part this strategy can be improved without devising a whole different approach is merging. Had the parallel merging code worked, we could have attained a better speedup. Indeed, I'd like to conclude on a simple numerical experiment.

Let's examine the 8 nodes, 128M size case. Our sequential divide and conquer merging algorithm required to merge 128/8=16M size instances into a sorted 128M solution. This was benchmarked at $t_m^{(8,128M)} = 2984$ ms.

We shall use the 2 node merge times on different sized instances to estimate binary merge times: remember that, for instance, $t_m^{(2,32M)}$ represents the time to merge two 16M instances into one 32M instance. First, let's validate that these times are representative of sequential merging, and then let's use them to estimate the speedup of parallel merging.

Recall that, in our (8,128M) case, we need to merge four pairs of 16M subinstances into two pairs of 32M subinstances, into one pair of 64M subinstances, and finally into a final sorted 128M sequence. This would all need to be done sequentially, so the times all add up:

$$t_m^S = 4t_m^{(2,32M)} + 2t_m^{(2,64M)} + t_m^{(2,128M)} = 4 \times 252 + 2 \times 499 + 994 = 3000 \text{ ms} \simeq t_m^{(8,128M)} = 2984 \text{ ms}$$

If we could merge in parallel, all multiplicative factors would be dropped, as each recursion level is solved in parallel by different nodes, but a small unknown penalty due to communication costs would appear, which we might copy over from the communication time of the (8,128M) case, since that's the amount of data that gets shared:

$$t_m^P = t_m^{(2,32M)} + t_m^{(2,64M)} + t_m^{(2,128M)} + t_c^{(8,128M)} = 252 + 499 + 994 + 325 = 2070 \text{ ms}$$

This is 2/3 of the sequential merge time, and would save almost a second off the overall (8,128M) time of $t = 5742$ ms, $t' = 4828$ ms, for an improved speedup of $S' = 4.30$ compared to $S = 3.62$. The benefits would scale sublinearly with the number of subsets, since the number of rounds of communication required to merge are the base 2 logarithm of the number of nodes.

This would improve our performance, but not grant perfect scalability, as we would need to somehow have a merging process that is independent of the number of subsets. Unless we can obtain that, the merging time will always far outweigh the sorting time at some point, as the sorting time can only get so small, while the merging time will increase indefinitely as more nodes/subsets are considered. On the other hand, there is the possibility that we might have results that are "effectively" or "realistically" scalable - if we only considered the amount of nodes we could realistically allocate.

Notes on the submitted code

Sequential quicksort was compiled with `g++ quicksort.cpp -o quicksort -O3` and requires a path to the input dataset. Output is on standard output, benchmark results on standard error. Advised command line usage is
`./quicksort path/to/dataset > /dev/null`

Parallel quicksort was compiled with `mpicxx parallelqs.cpp -o parallelqs -O3` and requires a path to the input dataset. Output is on standard output, benchmark results on standard error. Advised command line usage is
`mpirun -np # parallelqs path/to/dataset > /dev/null`

Submission includes code for dataset generation. Invoke with `./inputgen count > path/to/output`. Count must be the integer value: eg. 134217728 for 2^{27} (128M).