Performance comparison between single-threaded and multi-threaded merge sort implementations

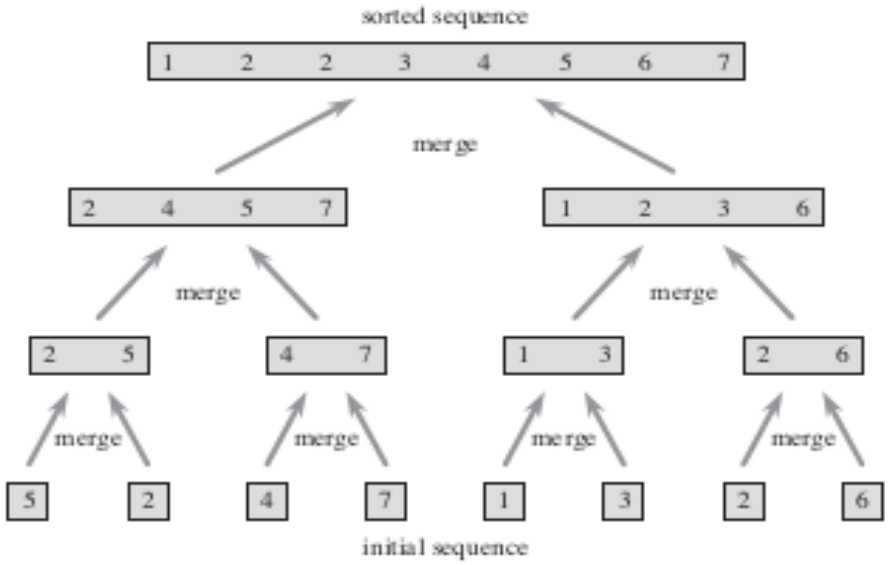
Marco Carlo Feliciano, Università degli Studi di Napoli Federico II, 2020/2021

Abstract

The aim is to make a performance comparison among three different merge sort implementations: the standard one, with a single thread; the multi-threaded one, exploiting a number of running threads equal to the number of available cores, but with a serial merge procedure; the multi-threaded one with parallel merge. Implementation details and figures are taken from [1].

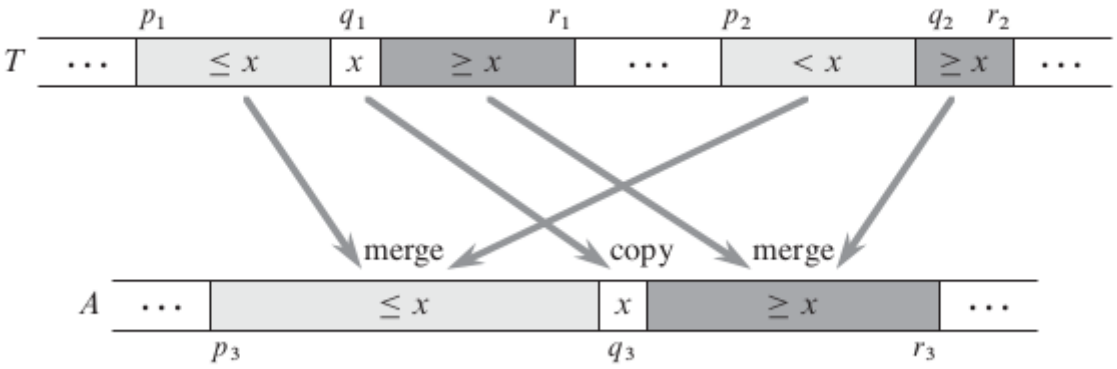
Implementation

Merge sort is a classic sorting algorithm, based upon divide-and-conquer approach. When you run merge sort, the function makes recursive calls to split the problem in smaller problems, until reaching a base case, i.e. a vector composed of a single element. Then, since the problem is split by half on each recursive call, you will come up with two base cases. You put them together by ordering them, and ordering two base cases only requires a single comparison and maybe a swap. Going back in the recursion tree, you won't have base cases anymore, but you will have pairs of ordered vectors from which you have to obtain a single ordered vector. The procedure to obtain this result is called *merge*. It's easy to merge two ordered vectors because you just have to make a linear run on both vectors, taking the minimum at each step, with two iterators. At the root, you will merge two ordered halves of the whole vector, obtaining the ordered vector. This is the standard algorithm, and its time complexity is equal to Θ(n*log(n)).



Now, to implement a parallel merge sort, just note that the sub-arrays obtained by splitting the problem in smaller problems are mutually exclusive. In fact, if we look

at the recursion tree, we'll see that nodes at the same height of the tree work on different sub-arrays. Furthermore, a node contains all the sub-arrays contained by its children. So, if you want, for example, two threads to run in parallel, you can just start a thread which executes merge sort on the left half on the vector and another thread which executes merge sort on the right half, join both threads (wait for completion) and make the *manager* call the merge procedure. The manager is the thread who spawns other threads, called *workers*. This is a perfect parallelism, because workers don't need any synchronization: they work on different parts of the array. The only synchronization is made by the manager, but if you think at that, even with a single thread you first call merge sort on the left half, than on the second half and finally you call the merge procedure. We would expect a linear speed-up. But the last merge operation's time complexity is Θ(n), so we know we can't actually expect a linear speed-up. Anyway, this version is the multi-threaded one with serial merge. If we want to exploit parallelism on the merge operation, we need a clever solution, because apparently there is no way to parallelize that procedure. The approach is the following: if we have two sorted arrays to merge, we can take the median of the longest one and make a binary search in the other with that median value, to find the index that splits the shortest array according to the median value of the longest array; then, all elements in both left sides (one is the left with respect to the median, the other one is found with binary search) are strictly less than the median: therefore, the sum of the lengths of these two sides, plus a base index, gives us the index of the position where the median has to be written in the merged vector. Now, we can re-apply this procedure using the left sides as vectors to merge with the same base index (note that the vectors to merge don't need to be adjacent anymore, but mutual exclusion is still preserved), and again using the right sides as vectors to merge with base index equal to the index where the median value was written to.



Now, you can see the picture: this merge procedure makes two recursive calls, like merge sort. It's not a strict limit, but we can safely say that this procedure has a time complexity equal to O(n*log(n)) in the worst case. In addition to that, it has a lowest bound of Ω(n), because it has to fill a sorted array of n elements, so it can't be faster than linear. Its time complexity is then worse than the standard merge. But it is suitable for a parallel implementation: you just make the procedure spawn new threads which execute the recursive calls. Of course, as the parallel merge sort, you

can’t spawn as many threads as recursive calls: it’s pointless to spawn more threads than available cores. It’s a good idea to make the parallel algorithm adaptive to the number of CPU cores, to have a maximum number of running threads at the same time equal to the number of cores. That’s how to build the multi-threaded merge sort with parallel merge. Now we’re going to see the actual performance tests.

Performance tests

Implementation of the described algorithms were made in Python. Tests were executed on a system with Fedora 33 as operating system, with CPU “Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz”, which is equipped with 4 cores, and with 8 GB of RAM. Data was obtained according to the following test cases, in which the value indicates both the dimension of the array to sort and the range of values in that array, which are chosen from a uniform distribution of random values.

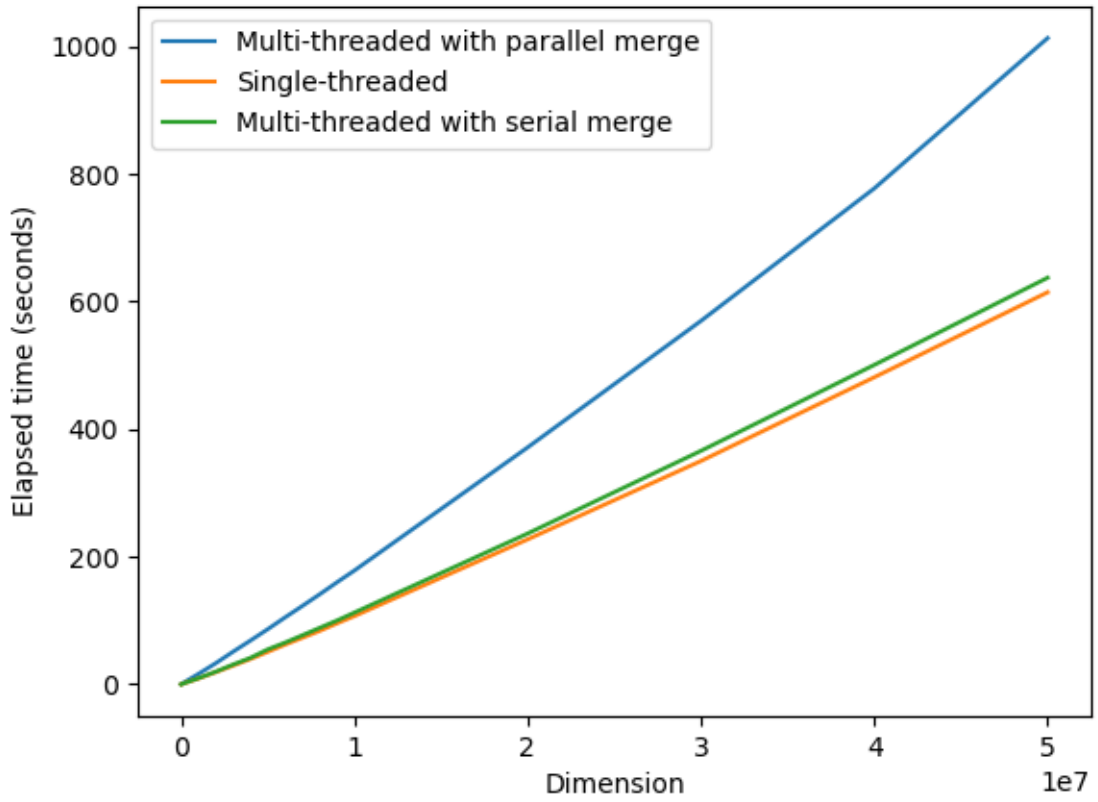
Test case	Start	Step	End
Base	0	100	1100
Low	10000	10000	110000
Mid	1000000	1000000	10000000
High	10000000	10000000	60000000

The aim of this organization of the tests is to reduce the number of tests to execute, by defining ranges of values, not inclusive at right. Note the test with 0 elements, which belongs to a different equivalence class with respect to the other tests. The limit on the high test case is dictated by RAM capacity: the array of dimension 5*1e7 with values in range [0, 5*1e7] created with numpy, occupies alone 2 GB of RAM, and we know that merge sort doesn’t sort in place.
Here are the results:

	Elapsed time (seconds)		
Dimension	Single thread	Multi-threaded with serial merge	Multi-threaded with parallel merge
0	0,0001	0,0003	0,0003
100	0,0034	0,0022	0,0045
200	0,0013	0,0026	0,0055
300	0,0020	0,0055	0,0067
400	0,0025	0,0048	0,0079
500	0,0031	0,0046	0,0094
600	0,0039	0,0056	0,0109
700	0,0046	0,0065	0,0119

800	0,0054	0,0074	0,0129
900	0,0062	0,0092	0,0147
1000	0,0067	0,0091	0,0155
10000	0,0867	0,0751	0,1416
20000	0,1639	0,1629	0,3050
30000	0,2284	0,2336	0,4320
40000	0,3024	0,3054	0,5755
50000	0,3849	0,3858	0,7121
60000	0,4434	0,4594	0,8520
70000	0,5236	0,5442	1,0010
80000	0,6082	0,6341	1,1453
90000	0,7329	0,7130	1,2986
100000	0,7692	0,7954	1,4511
1000000	8,8566	9,3121	16,0178
2000000	18,7517	19,5710	32,8437
3000000	29,0480	30,7456	50,8866
4000000	39,4078	41,2599	68,3409
5000000	50,5629	54,5249	86,1856
6000000	61,9527	65,1500	104,6598
7000000	72,7011	76,6998	122,6709
8000000	83,5247	88,3135	140,9533
9000000	95,3571	99,8377	159,8267
10000000	106,7343	112,2385	178,1530
20000000	227,1190	236,2690	371,3445
30000000	350,0104	365,9257	569,7366
40000000	481,2802	500,6838	777,4758
50000000	614,5291	637,3590	1013,1676

Within a single plot:



Notice that single-threaded implementation proved to be faster than multi-threaded ones, and even multi-threaded with serial merge was faster than multi-threaded with parallel merge: looks like parallelism slowed down the algorithm instead of making it faster. Let's dive into the reasons behind this result, borrowing some concepts about threads, processes and scheduling from [2]:

1. Threads belong to a single process, and the process is the *unit of scheduling* for an operating system: therefore, threads can't be scheduled independently from each other, they're always scheduled all together.
2. Even when the python process executing the sort is scheduled, the operating system doesn't devote all available cores to that process, because there are many other background jobs executing.
3. There is some overhead related to the use of threads, which is "hidden in the asymptotic notation".
4. Results obtained in the theory of parallel algorithms assume to have a perfect parallel underlying computing architecture, with an unlimited degree of parallelism, whereas in our system we only have a finite number of threads which don't even actually work in parallel.
5. In the case of the merge based on binary search, we said that the time complexity can go up to O(n*log(n)) with a single thread, and that the function is recursive; it can easily be executed in parallel, but even with a perfect parallel computing architecture with 8 parallel running threads, this function

would be slower than the linear merge: this is because the term log(n), which has base 2 in this case, dominates the number of running threads as the dimension of the problem increases, i.e. $\log_2(n) / \text{parallel}_{\text{threads}} > 1$. The linear merge, in addition to that, doesn't have hidden constants in the asymptotic notation, so it is w*ay* faster.

6. In the single-threaded case, computations tend to focus around a certain sub-array: if you look at the recursion tree, you first go in-depth, reaching a base case, then you merge it with another base case and you go back, merging vectors that are physically and logically *near* between them, and that you *visited recently*, except when you arrive at the root and you have to visit the right sub-tree. This means that the single-threaded implementation takes advantage of the *principle of locality* and of *cache mechanisms* better than multi-threaded implementations. In fact, in multi-threading, the accesses to the big vector become "more random than sequential", so they are slower because even in RAM sequential accesses are faster than random accesses, and there could be a low cache hit ratio, resulting in a form of thrashing between cache memories and RAM.

Conclusion

We showed an approach to implement a sorting algorithm which exploits multi-threading, starting from the standard single-threaded algorithm, without needing of synchronization. We then showed the results obtained from the tests made on a modest architecture, armed with a 4-core CPU, justifying the fact that the single-threaded implementation proved to be faster with a deeper look into how the operating system handles threads. Now, this doesn't prove that single-threading is better than multi-threading, but the choice between them doesn't need to be blind. Here are some use cases for a parallel algorithm:

- When you have an underlying architecture which enables for a more performing parallelism, such as a GPU: it would be fully dedicated to the parallel execution, providing a cache memory for any thread and the actual parallel scheduling.
- When you are forced to use a parallel algorithm, because the size of the problem is too big to be handled on a single node, and therefore you have to split the work in multiple nodes, each one with its processor; in this case, you would have to design a communication protocol among these nodes, which are part of a distributed system, and the overall performance will also depend on the efficiency of the protocol in terms of number of messages.
- I/O bound applications, such as network ones: it's fine to have many threads waiting for I/O operations at a given time, because you can wake them in response to events and commit the CPU to other tasks meanwhile, obtaining a good overall usage of system resources.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", The MIT Press, 3rd edition, 2009

[2] William Stallings, "Operating Systems. Internals and Design Principles", Prentice Hall, 7th edition, 2012