1 Quicksort and Concurrency

1.1 Comparing quicksort implementations

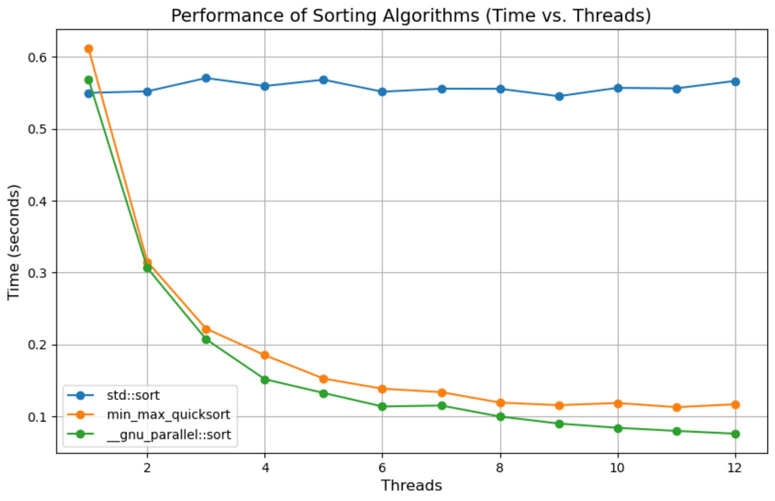
1.1.1 Benchmarking environment

The benchmarks were performed on the following system:

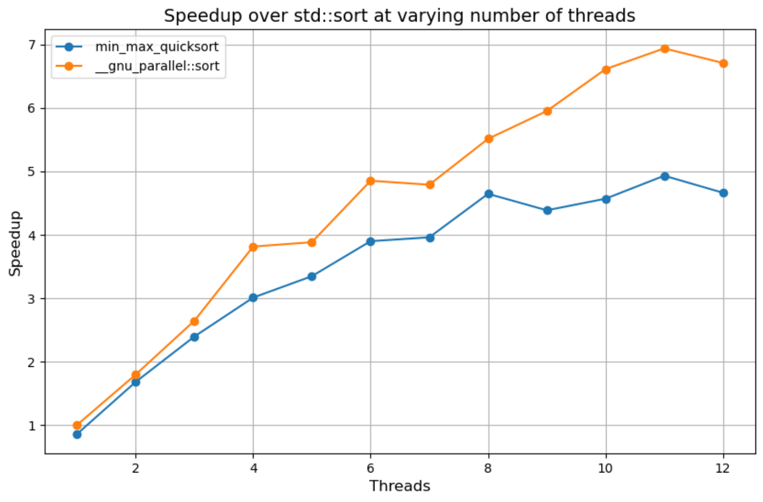
Operating System:	Ubuntu 24.04.1 LTS
RAM:	32 GB
CPU:	AMD Ryzen 5 5600G with Radeon Graphics (6 Cores, 2 Threads per Core)
Compiler:	gcc version 13.2.0

1.1.2 Running time comparison with different numbers of threads

Here are the results when running the sorting algorithms with varying numbers of threads:

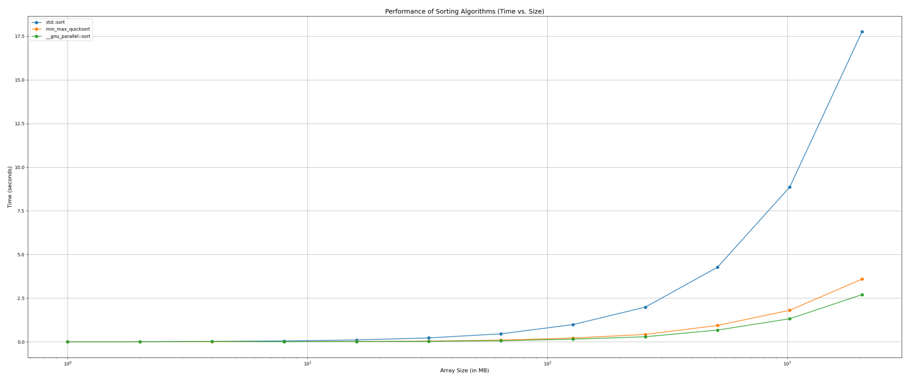


And here are the performance gains compared to std::sort(). Note the different coloring:

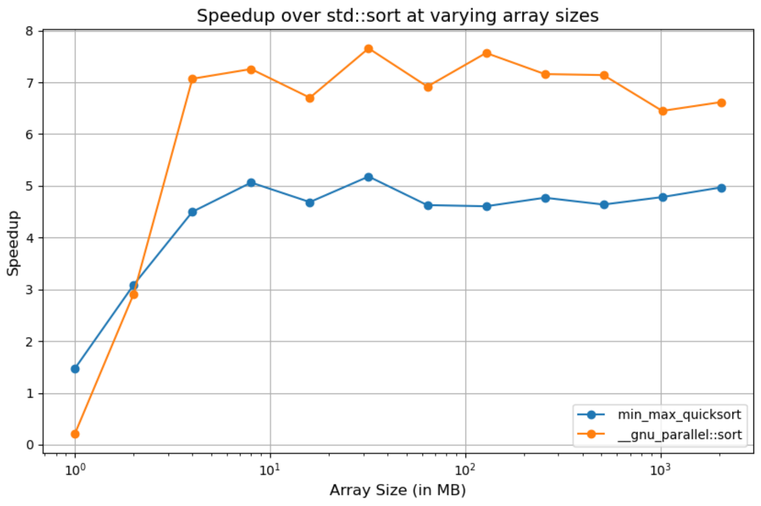


1.1.3 Running time comparison with different array sizes

Here are the running times with varying array sizes:



These are the speedups compared to std::sort():



1.1.4 Explaining the results

In the first two graphs when measuring the running times and speed ups when varying the threads, we can clearly see that the serial std::sort has a constant running time, whereas both parallel sorting algorithms run quicker with increasing thread number, thus overtaking std::sort already with two threads significantly (almost halving the running time of std::sort). With further increasing thread number, the running times of the parallel algorithms further decreases. They are proportional to $\frac{1}{\text{Number of Threads}}$, as one might expect, but the proportionality factor is roughly at 2, meaning there is quite some overhead compared to the optimal running time of $\frac{\text{Serial Running Time}}{\text{Number of Threads}}$.

Let us now analyze the second two pictures, showing the running times at different array sizes. The graphs are as expected, the running times of the parallel algorithms is better, even more so for very large array sizes, as in those instances we can fully employ the parallel power of the CPU cores without much overhead relative to the running time, thus decreasing the running time significantly. This is why the speedup gets better with increasing array size initially. The speedup, however, start to plateau quickly at an array size of roughly 1 MB, as at this point the overhead of parallelization with respect to the running time stays constant (i.e. $\frac{\text{Time of Overhead}}{\text{Parallel Running Time}} = \text{const.}$):

Since

$$\text{speedup} = \frac{\text{Serial Running Time}}{\text{Parallel Running Time}} \qquad ,$$

and

$$\begin{aligned}\text{Parallel Running Time} &= \frac{\text{Serial Running Time}}{\text{Number of Threads}} + \text{Time of Overhead}\\ &= \frac{\text{Serial Running Time}}{\text{Number of Threads}} + \alpha \cdot \text{Parallel Running Time}\\ &= \frac{\frac{\text{Serial Running Time}}{NumberofThreads}}{1-\alpha}\\ &= \alpha' \cdot \text{Serial Running Time} \qquad ,\end{aligned}$$

we have

$$\text{speedup} = \frac{1}{\alpha'} = \text{const.} \qquad .$$

1.2 What every systems programmer should know about concurrency

1.2.1 Why it is not enough to declare shared variables as volatile

By declaring variables as volatile in C and C++, we make the compiler read and write the variable from and to memory every time we access it. Cache coherency protocols then ensure that every local cache of the CPU cores has the same value of the shared variable. So everything should be good then, right?

Well, there is one problem: Modifying the shared variable isn't an atomic operation. Imagine two threads wanting to increment a shared counter. But as executing this increment operation takes multiple clock cycles, thread one starts reading the counter from memory, and a few ticks later thread two reads the same value of memory. Now, both threads increment this value and write it back to memory, so thread one does it first, which causes the cache of thread two to now have the updated counter variable in its cache, but the value it is operating on, which lies in its registers, is unchanged, so it will get the same result and write it back to memory. In the end, the counter increased by one, even though it got incremented two times, which is (in most cases) undesired and incorrect behavior.

1.2.2 But what if my ISA doesn't support atomic access to my custom data structure / code section

We now learned that we have to use atomic variables to ensure correct parallel code execution. But what if the shared variable has no atomic variable data type like bool has with std::atomic_bool? Or we want to synchronize an entire code section? As it turns out, we only need a few of these atomic operations and data types to implement generic synchronization functionality, as those can be used to built so called *locks*.

One well known example is a *semaphore*. It basically consists of an atomic counter, which represents how many threads may still access the shared resource (it is mostly either 0 or 1). If two threads want to access a shared resource, they have to first check if the semaphore has a value of one, and only if it does, decrement the value immediately and progress in their execution. If the semaphore is zero, the thread has to wait (which is mostly done by making it sleep). As you can see, "check if semaphore is one, and if so decrement it" must be an atomic operation for the semaphore to work. And this is exactly why modern ISA have instructions like *compare-and-swap*, which implement such operations atomically in hardware. There are also some other common instructions of this kind. They are combined under the term *read-modify-write*.