Algorithm Engineering

Jonas Peters

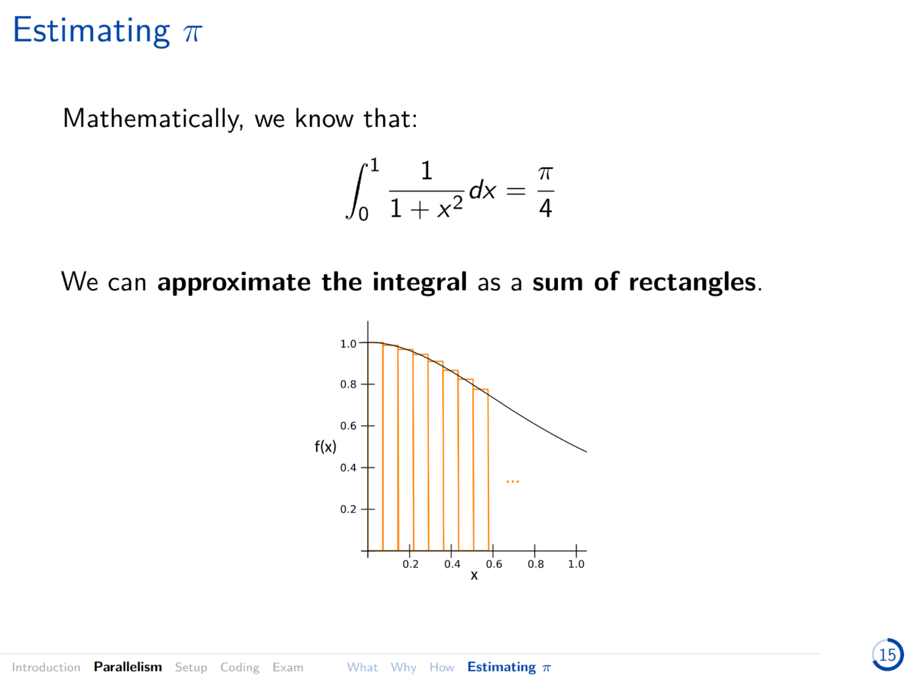
November 18, 2024

Contents

1 Introduction to Algorithm Engineering

1.1 Analyzing slide 15

Here is slide 15 again:

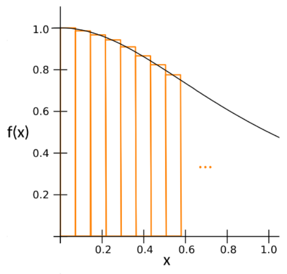


Estimating π

Mathematically, we know that:

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}$$

We can **approximate the integral** as a **sum of rectangles**.



Introduction **Parallelism** Setup Coding Exam What Why How **Estimating π** 15

Let us proof the satement

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}$$

using

$$(f^{-1})'(x_0) = \frac{1}{f'(f^{-}1(x_0))} \qquad :$$

Proof. Let $\tan : (-\frac{\pi}{2}, \frac{\pi}{2}) \to \mathbb{R}$ be the usual trigonometric function. Clearly, it is a bijection, and hence its inverse function $\arctan : \mathbb{R} \to (-\frac{\pi}{2}, \frac{\pi}{2})$ exists. Now, let us argue with the above identity:

$$\arctan'(x_0) = \frac{1}{\tan'(\arctan(x_0))} = \cos^2(\arctan(x_0)) \qquad ,$$

where we have used that $\tan'(x_0) = \frac{1}{\cos^2(x_0)}$. (This can easily be derived by the chain and product rule, knowing the derivatives of $\sin, \cos$ and $\frac{1}{x}$, as well as the fact that $\tan(x) = \frac{\sin(x)}{\cos(x)}$.)

Now, we would like to transform $\cos^2(\bullet)$ into something like foo(tan($\bullet$)), such that tan and arctan would cancel. Luckily, we can do this using the trigonometric identity

$$\begin{aligned} & \sin^2 + \cos^2 = 1 \\ \Leftrightarrow\ & \tan^2 \cos^2 + \cos^2 = 1 \\ \Leftrightarrow\ & (\tan^2 + 1) \cos^2 = 1 \\ \Leftrightarrow\ & \cos^2 = \frac{1}{\tan^2 + 1} \end{aligned}$$

Plugging this into our equation yields

$$\arctan'(x_0) = \frac{1}{\tan^2(\arctan(x_0)) + 1} \qquad .$$

But by the definition of arctan, we get

$$\arctan'(x_0) = \frac{1}{x_0^2 + 1} \qquad .$$

Thus, we have established that arctan is an antiderivative of $\frac{1}{x^2+1}$.
By the fundamental theorem of calculus we get

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(1) - \arctan(0) \qquad .$$

Now note that $0, \frac{\pi}{4} \in (-\frac{\pi}{2}, \frac{\pi}{2})$, and $\tan(0) = 0$, $\tan(\frac{\pi}{4}) = 1$. In other words, we have $\arctan(0) = 0$, $\arctan(1) = \frac{\pi}{4}$. Thus,

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4} \qquad .$$

□

The reason we established this equation in the first place is because it allows for a simple approximation for π. We do this by using numerical methods for approximating integrals.

One way to do this is to calculate the Riemann sum, as shown in the lecture. The Riemann sum tries to approximate little intervals of the function $\frac{1}{1+x^2}$ by constant functions. A better way to approximate the function on these intervals would be a linear function connecting the end points, resulting in the *Trapezoidal Rule*.

Even better is the approximation by parabolas, which pass through the end points of the interval as well as the mid point. This is called the *Simpson Rule*. It reads as follows:

$$\int_t^{t+h} f(x)dx \;\approx\; \frac{h}{6}\Big[f(t) + 4f(t+\frac{h}{2}) + f(t+h)\Big] \qquad .$$

When approximating an integral by splitting it into smaller intervals, we use the following formula. To this end, let P be such a partition. Then

$$\int_a^b f(x)dx \quad \approx \sum_{(t,t+h)\in P} I_{(t,t+h)}(f) \qquad ,$$

where $I_{(t,t+h)}(f)$ tries to approximate $\int_t^{t+h} f(x)dx$.

Combining these results, one might get even faster convergence. But note that machines might have trouble dealing with very small numbers, thus this process should be used with caution.

1.2 Chapter 1 from Computer Systems: A Programmer's Perspective

The book gave a quick overview about operating systems and computer hardware, and the flow of execution. I'd like to discuss the memory hierarchy as well as virtual memory.

1.2.1 Memory Hierarchy

The inverse correlation between speed and volume of different memory types war presented in the book. This lead to the memory hierarchy, where we try exploit the benefits of both worlds be having big, slow memory, and a smaller amount of fast memory, which we the computer tries to employ as much as possible.

Two reasons for this are

1. Increased physical memory cells: More logical memory requires more physical memory cells. These cells need to be connected to control circuits, and larger memory arrays result in longer signal paths and higher capacitance. This increases signal propagation delay, making the memory slower.
2. Heat production: Larger memory volumes consume more power, and the switching of many transistors generates more heat. Excessive heat can degrade performance and force memory to operate at lower speeds to prevent overheating.

1.2.2 Virtual Memory

One of the core principles in mathematics and computer science is abstraction. The operating system provides such an abstraction in form of I/O communication, running multiple programs concurrently using processes, and translating virtual addresses into physical ones.

This is very interesting, as by doing so, two distinct processes are unable to access the other ones data. To illustrate this, imagine you write a simple C program, create a pointer, print the pointer location, write an integer there based on a command line argument, and print the content periodically. If one now starts two instances of the program, one where it writes a 1, another one with 0, they won't interfere with one another, the first process will always read a 1, the second one 0.

Behind the scenes, the operating system works together with the *memory management unit*, which translates every address the program uses to its real address before accessing. And exactly this translation process is program specific. Hence, the operating system successfully abstracted the memory.

1.3 Parallelizing "Estimating π using Monte Carlo"

Here is the parallelized version of the code:

```
#include <iostream>
#include <omp.h>
#include <random>

using namespace std;

int main() {

    int n = 100000000; // number of points to generate
    int counter = 0; // counter for points lying in the first quadrant
        of a unit circle
    auto start_time = omp_get_wtime(); // omp_get_wtime() is an OpenMP
        library routine

    // compute n points and test if they lie within the first quadrant
        of a unit circle
    #pragma omp parallel
    {
        default_random_engine re{(size_t) omp_get_thread_num()};
        uniform_real_distribution<double> zero_to_one{0.0, 1.0};

        int local_counter = 0;
        int local_n = (n / omp_get_num_threads()) + ((n %
            omp_get_num_threads() > omp_get_thread_num()) ? 1 : 0);
        for (int i = 0; i < local_n; ++i) {
            auto x = zero_to_one(re); // generate random number between
                0.0 and 1.0
            auto y = zero_to_one(re); // generate random number between
                0.0 and 1.0
            if (x * x + y * y <= 1.0) { // if the point lies in the first
                quadrant of a unit circle
                ++local_counter;
            }
        }
        #pragma omp atomic
        counter += local_counter;
    }

    auto run_time = omp_get_wtime() - start_time;
    auto pi = 4 * (double(counter) / n);

    cout << "pi: " << pi << endl;
    cout << "run_time: " << run_time << " s" << endl;
    cout << "n: " << n << endl;
}
```

2 False Sharing, Race Conditions, and Schedules

2.1 False Sharing

Consider two teachers who need to count their students. To divide the work, one teacher decides to count the boys, while the other counts the girls. They use a single sheet of paper to keep track, marking a stroke on their respective sides of the page for each student they count.

However, even though they each have their own pen, they're still working with only one piece of paper. This setup means that every time one teacher wants to make a mark, they must pass the paper to the other, as both are using different parts of the same sheet.

This constant handoff slows them down significantly. Had they used separate sheets, they could each count freely without interrupting one another.

In the same way, false sharing occurs in computing when multiple threads modify different parts of the same cache line. Even though each thread may be working on separate variables, they're forced to constantly invalidate and reload the cache line they "share," leading to substantial inefficiencies.

Here is an illustration of the sceario:

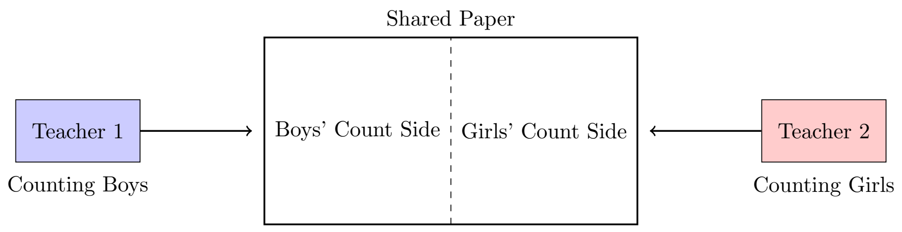
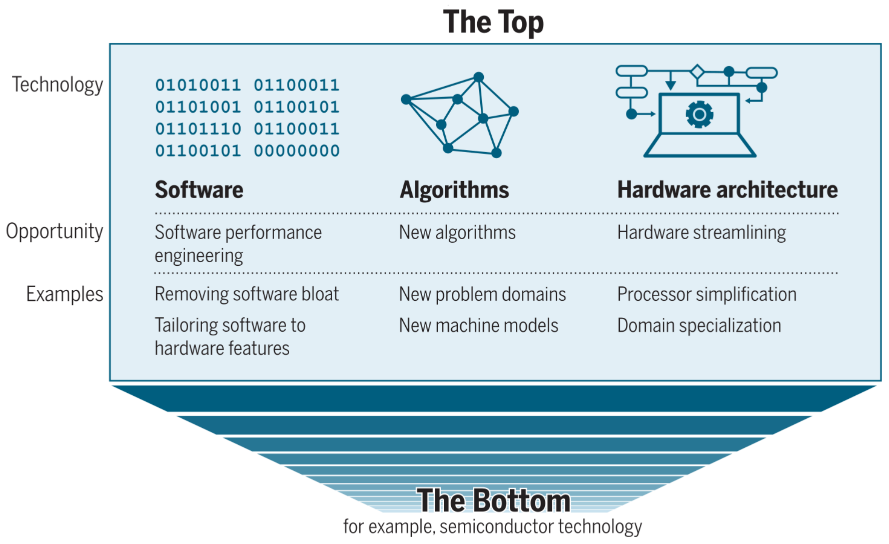


Figure 1: False Sharing Analogy: Two teachers repeatedly passing a single paper to count students separately.

2.2 There's plenty of room at the Top

Let us analyze the following graphic from the paper *There's plenty of room at the Top: What will drive computer performance after Moore's law?* :



Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

The entire blue area basically represents the foundation of running software: We need hardware ("The Bottom") as well as good software ("The Top"). The width represents the potential of gains we could make in these areas. In particular, the width of "The Bottom" shrank over time and is now becoming less significant.

The paper assumes that the most potential lies in the three categories *Software, Algorithms, and Hardware Architecture*, all of which are located at The Top. They describe their potentials in the *Opportunity* and *Examples* fields. Let us analyze them individually:

2.2.1 Software

The software that is ultimately run on the hardware can be optimized itself. In Software Engineering, there is a phenomenon called *bloat*. It describes the accumulation of unnecessary functionality and features, which makes programs bigger and hence slower. By carefully writing code to avoid these traps, we can improve the running time of our programs.

Tailoring software to hardware features allows developers to leverage specific capabilities of the underlying hardware. For instance, optimizing software to use SIMD (Single Instruction, Multiple Data) instructions can dramatically increase performance for tasks that involve large data sets, such as graphics processing or numerical simulations. Additionally, understanding cache architectures can help developers design algorithms that reduce cache misses, leading to faster data access and processing. By aligning software architecture with hardware capabilities, significant improvements in execution speed and resource utilization can be achieved.

2.2.2 Algorithms

The choice of algorithms plays a critical role in determining software performance. Implementing new algorithms that are more efficient in terms of time and space complexity can lead to significant performance enhancements. For example, employing faster sorting algorithms or utilizing data structures that provide quicker access times can reduce the computational overhead.

New problem domains often introduce unique challenges that require innovative algorithmic solutions. For instance, the advent of big data has prompted the development of algorithms specifically designed for distributed computing environments, which can process large datasets across multiple machines. By addressing the specific needs of these new domains, developers can create algorithms that are optimized for performance and scalability.

2.2.3 Hardware Architecture

Hardware streamlining involves optimizing hardware components to enhance overall system performance. This can include reducing the complexity of circuits, minimizing power consumption, and improving thermal management. Streamlined hardware can lead to faster processing speeds and better energy efficiency, directly impacting software performance by reducing bottlenecks.

Processor simplification is another approach to improving performance. By reducing the number of transistors or focusing on a reduced instruction set, processors can operate at higher speeds and with lower power consumption. Simplified processors are often easier to design, implement, and manufacture, which can also reduce costs while enhancing performance metrics.

Domain specialization allows hardware to be optimized for specific applications or problem domains. For example, graphics processing units (GPUs) are designed to handle complex graphics calculations more efficiently than general-purpose CPUs. Similarly, application-specific integrated circuits (ASICs) can be tailored for particular tasks, such as cryptocurrency mining or machine learning inference, resulting in substantial performance gains over generic hardware solutions.

2.3 Parallelizing pi_numerical_integration.cpp

Here is the parallelized version using the *#pragma omp for* construct:

```
#include <iomanip>
#include <iostream>
#include <omp.h>

using namespace std;

int main() {
    int num_steps = 100000000; // number of rectangles
    double width = 1.0 / double(num_steps); // width of a rectangle
    double sum = 0.0; // for summing up all heights of rectangles

    double start_time = omp_get_wtime(); // wall clock time in seconds
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < num_steps; i++) {
        double x = (i + 0.5) * width; // midpoint
        sum = sum + (1.0 / (1.0 + x * x)); // add new height of a
            rectangle
    }
    double pi = sum * 4 * width; // compute pi
    double run_time = omp_get_wtime() - start_time;

    cout << "pi with " << num_steps << " steps is " << setprecision(17)
        << pi << " in " << setprecision(6) << run_time << " seconds\n";
}
```

3 Parallelizing Code with OpenMP

3.1 Chapter 3 of Introduction to Parallel Computing

3.1.1 Comparing code snippets

The code in listing 3.1 reads as follows:

```
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Hello, world:");
    #pragma omp parallel
        printf(" %d", omp_get_thread_num());
    printf("\n");
    return 0;
}
```

In this code, a team of thread get created my the *#pragma omp parallel* clause. Their task is to print their respective thread numbers. But since no order is defined, the order of their execution are "random", i.e. determined at runtime by the operating system.

Listing 3.2 reads as follows:

```
#include <stdio.h>
#include <omp.h>

long fib(int n) {
    return (n < 2 ? 1 : fib(n-1) + fib(n-2));
}

int main() {
    int n = 45;
    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        printf("%d: %d\n", t, fib(n+t));
    }
    return 0;
}
```

While still not defining the order of the tasks, they yet seem to execute in order. Interestingly, this is not quite true, as the main reason the seem to do so is because the *terminate* in order. This is simply due to the fact that the work load for a thread with a higher thread number is substantially higher than for another one with a lower number. This is because fib(n+t) is invoked, but it takes very long to calculate because of the big n, and even a singe step by t causes fib(n+t) to take nearly twice as long.

3.1.2 Game of Life

The code in Listing 3.10 looks well designed. It collapses the two nested for loops of size *size* into a single one of size $size^2$. Since each iteration approximately takes the same amount of work, the implicit static scheduling ensures a good distribution of the workload. This way, every thread has to compute $\frac{size^2}{\text{omp\_get\_num\_threads()}}$ iterations.

The outer most loop cannot be parallelized, as their iterations are all strictly dependent on the previous one. This is why the implicit barrier in line 13 is neccessary. It also shouldn't be that big of a bottleneck, as the threads should reach it almost simultaneously.

3.1.3 Optimizing Random Shooting

When trying to approximate π by the method of random shooting we use random numbers. They, however, don't need to be truly random, they just have to simulate it well enough. One way to do this simulation of randomness is the rnd function like it is defined in the book (or see the code below). It calculates a random number quite efficiently. Another benfit is that we do not need multiple instances of random number generators, and we also don't run the risk of misusing those by calling their methods in different threads. The code is shown in the figure below. It runs very quickly and calculates π very precisely. The running time on my particular machine was 0.0313137 s, in comparision to the previous version with a running time of 0.117771 s.

4 Quicksort and Concurrency

4.1 Comparing quicksort implementations

4.1.1 Benchmarking environment

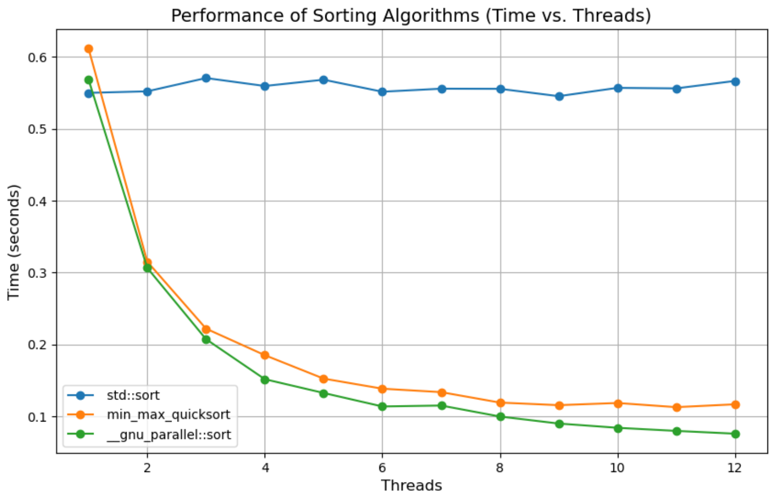
The benchmarks were performed on the following system:

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

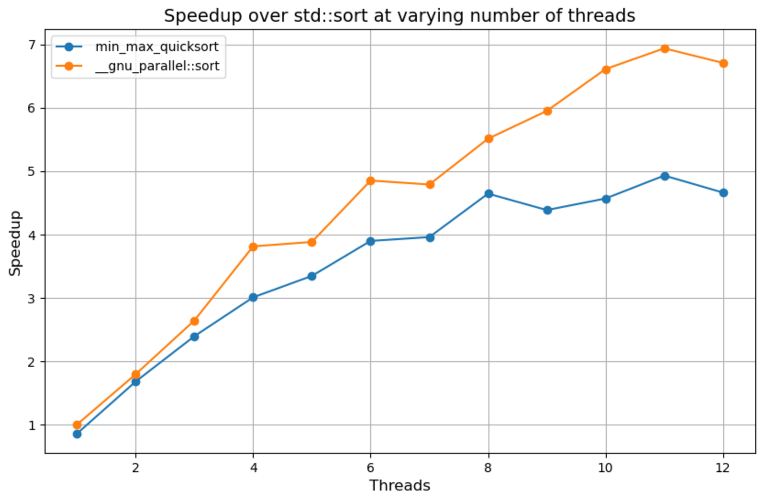
```
#include <iostream>
#include <omp.h>
#include <random>

using namespace std;

double rnd(unsigned int *seed)
{
    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
    return ((double)(*seed)) / (1 << 24);
}

int main()
{
    int n = 100000000;              // number of points to generate
    int counter = 0;                // counter for points lying in the
        first quadrant of a unit circle
    auto start_time = omp_get_wtime(); // omp_get_wtime() is an OpenMP
        library routine

    // compute n points and test if they lie within the first quadrant
        of a unit circle
    #pragma omp parallel reduction(+:counter)
    {
        unsigned int seed = omp_get_thread_num();
        size_t local_n = (n / omp_get_num_threads()) + ((n %
            omp_get_num_threads() > omp_get_thread_num()) ? 1 : 0);

        for (int i = 0; i < local_n; ++i)
        {
            auto x = rnd(&seed); // generate random number between 0.0
                and 1.0
            auto y = rnd(&seed); // generate random number between 0.0
                and 1.0
            if (x * x + y * y <= 1.0)
            { // if the point lies in the first quadrant of a unit circle
                ++counter;
            }
        }
    }

    auto run_time = omp_get_wtime() - start_time;
    auto pi = 4 * (double(counter) / n);

    cout << "pi: " << pi << endl;
    cout << "run_time: " << run_time << " s" << endl;
    cout << "n: " << n << endl;
}
```

4.1.2 Running time comparison with different numbers of threads

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

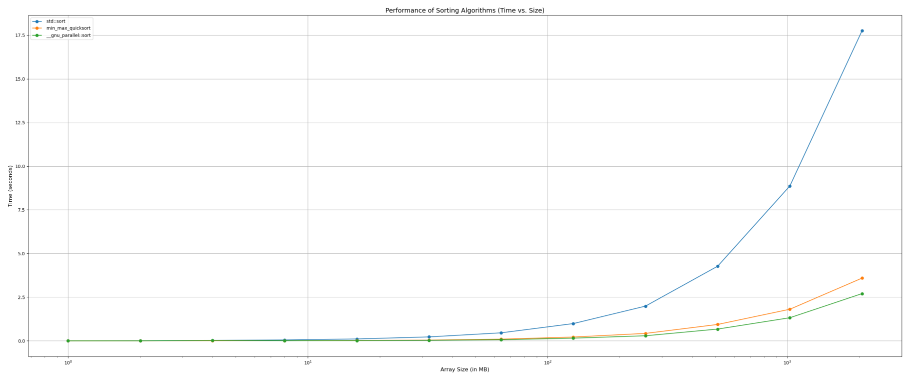


And here are the performance gains compared to std::sort():

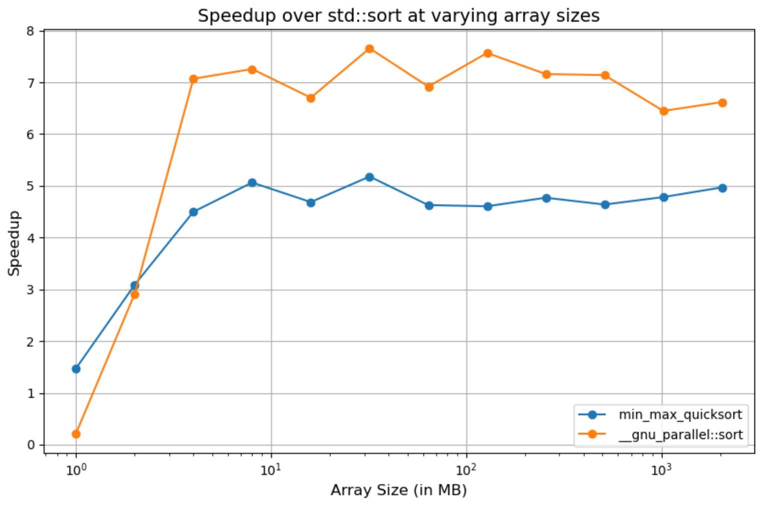


4.1.3 Running time comparison with different array sizes

Here are the running times with varying array sizes:



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



4.1.4 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

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

we have

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

4.2 What every systems programmer should know about concurrency

4.2.1 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.

4.2.2 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*.