Algorithm Engineering

Jonas Peters

December 10, 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**.
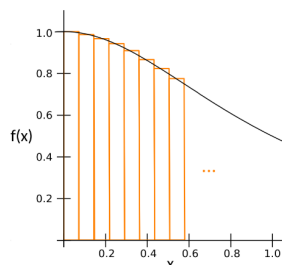
Let us proof the statement

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

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 $\tan(x) = \frac{\sin(x)}{\cos(x)}$.)

3

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

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

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,

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

$\square$

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:

$$\int_t^{t+h} f(x)dx \approx \frac{h}{6}\left[f(t) + 4f(t + \frac{h}{2}) + f(t + h)\right] \quad .$$

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

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

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:

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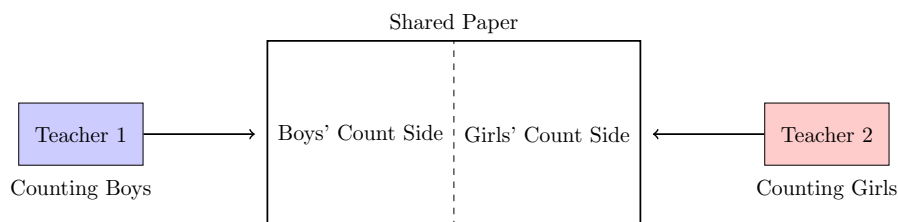
Figure 1: False Sharing Analogy: Two teachers repeatedly passing a single paper to count students separately.

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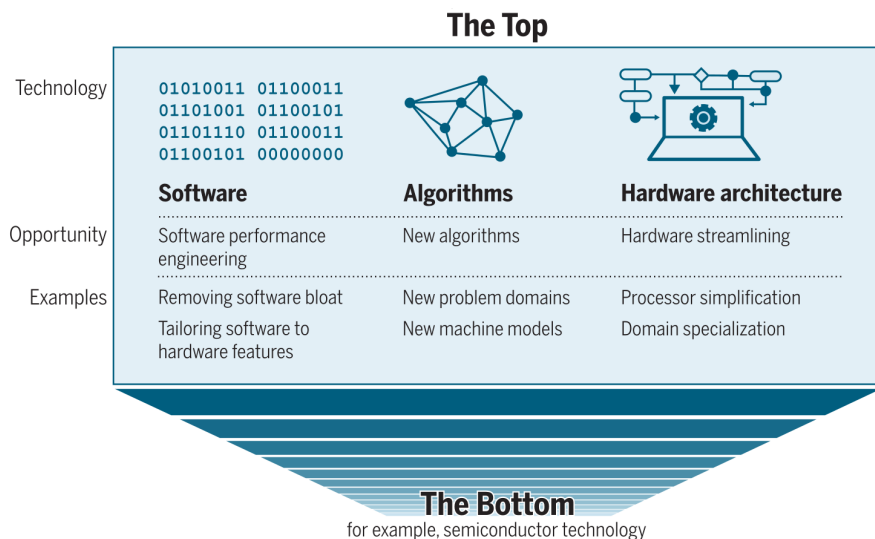
Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

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:

```cpp
#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:

```c
#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:

```c
#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 necessary. 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 comparison 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

```cpp
#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:
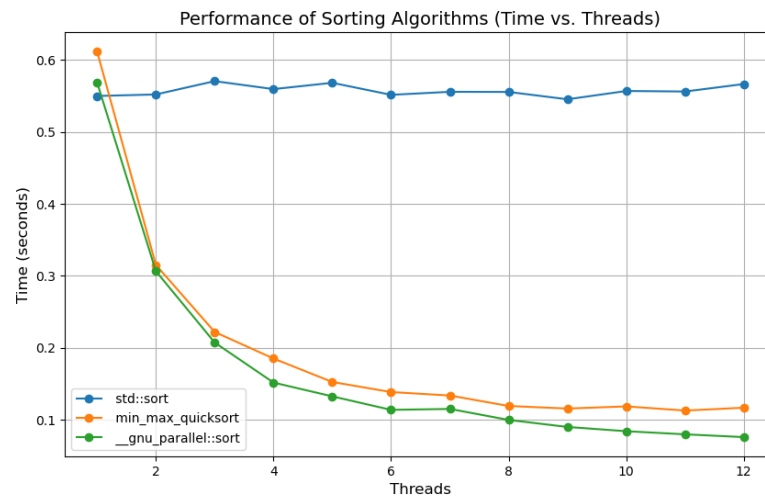
And here are the performance gains compared to std::sort():
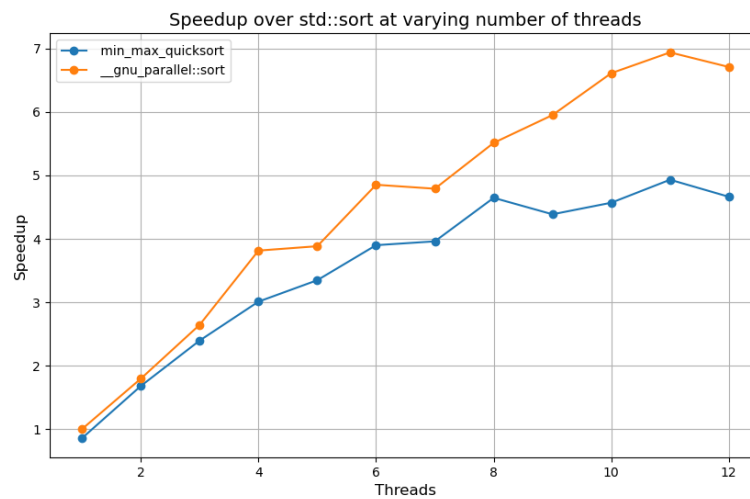
4.1.3   Running time comparison with different array sizes

Here are the running times with varying array sizes:
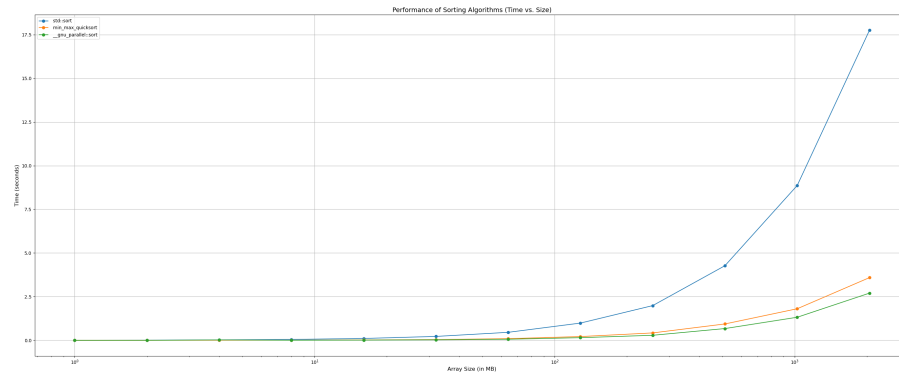
These are the speedups compared to std::sort():
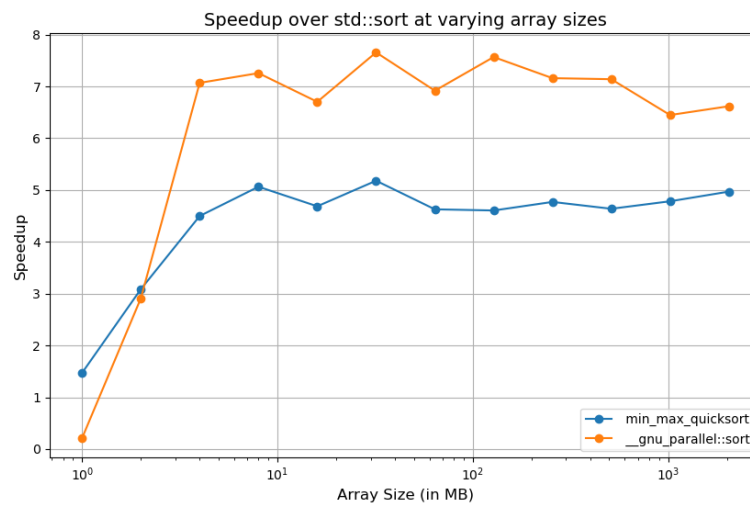
4.1.4 Explaining the results

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

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

and

$$\text{Parallel Running Time} = \frac{\text{Serial Running Time}}{\text{Number of Threads}} + \text{Time of Overhead}$$

$$= \frac{\text{Serial Running Time}}{\text{Number of Threads}} + \alpha \cdot \text{Parallel Running Time}$$

$$= \frac{\frac{\text{Serial Running Time}}{Number of Threads}}{1 - \alpha}$$

$$= \alpha' \cdot \text{Serial Running Time} \quad ,$$

we have

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

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*.

5 Auto Vectorization

5.1 Characteristics of SSE, AVX(2) and AVX-512

SSE (Streaming SIMD Extensions):
> Introduced with x86, supports 128-bit vector registers, enabling SIMD operations on 4 single-precision floating-point or 2 double-precision values in parallel. Limited in width and functionality compared to later extensions.

AVX/AVX2 (Advanced Vector Extensions):
> Extends SIMD width to 256 bits, doubling parallelism compared to SSE. AVX introduced floating-point operations, while AVX2 added integer operations and support for gather instructions.

AVX-512:
> Widens SIMD to 512 bits, allowing operations on 16 single-precision floats or 8 double-precision values simultaneously. It includes more registers, mask registers for conditional operations, and broader instruction sets but increases power consumption and requires hardware support.

5.2 Impact of Memory Aliasing on Performance

Memory aliasing occurs when two or more pointers reference overlapping memory regions, making it difficult for the compiler or CPU to optimize memory accesses. This uncertainty can prevent reordering or parallelizing memory operations, resulting in stalls and reduced performance due to additional dependency checks and cache conflicts.

5.3 Advantages of Unit Stride Memory Access

Unit stride (stride-1) memory access sequentially accesses adjacent memory locations, maximizing cache utilization and enabling efficient prefetching. Larger strides, like stride-8, result in less efficient cache usage, increased memory latency, and potential cache line thrashing due to non-contiguous data access patterns.

5.4   When to Prefer Structure of Arrays

Structure of Arrays (SoA) is preferred when working with SIMD/vectorized operations or when the workload requires processing individual fields of a dataset independently. SoA provides better memory alignment and avoids unnecessary memory fetches, leading to improved cache efficiency and parallelism compared to Array of Structures (AoS).

6 Guided Vectorization and Data Types for Vector Intrinsics

6.1 Vectorization Clauses

aligned

This clause tells the compiler that every element of an array a is n-bit aligned if one uses `#pragma opm simd aligned(a, n)`. The compiler employs this information to effectively vectorize the code section.

safelen

Imagine we had the following code:

```
a[0] = b[0];
for (int i = 1; i < n; i++) {
    a[i] = a[i-1] * (i % 2) + b[i];
}
```

We have a data dependency, but only every second iteration: iteration i, where i is odd, depends on $i - 1$. Iteration i and $i + 1$ are completely independent, however. Hence, we can vectorize the loop with a vector length of 2. To specify this, we use the `safelen` clause like so:

```
a[0] = b[0];
#pragma omp simd safelen(2)
for (int i = 1; i < n; i++) {
    a[i] = a[i-1] * (i % 2) + b[i];
}
```

reduction

This clause has the same functionality when paired with `simd` as when paired with `paraller for`: It will reduce a variable by an associative operation. But instead of reducing it over multiple threads, we use vectorization for reducing. I imagine that the reduced variable itself will be vectorized, so that the in loop we can employ vector operations, and at the end the elements in the resulting vector will get reduced.

6.1.1 AVX512 data types

Type	Layout	Description
__m512	F F F F F F F F F F F F F F F F	16x 32-bit float
__m512d	D D D D D D D D	8x 64-bit double
__m512i	(64 cells)	64x 8-bit byte
__m512i	s s s s s s s s s s s s s s s s s s s s s s s s s s s s s s s s	32x 16-bit short
__m512i	i i i i i i i i i i i i i i i i	16x 32-bit integer
__m512i	ll ll ll ll ll ll ll ll	8x 64-bit long
__m512i	dqw dqw dqw dqw	4x 128-bit quad
__m512i	qqw qqw	2x 256-bit integer
__m512i	dqqw	1x 512-bit integer

6.1.2 Intel MMX

In the paper Intel MMX for Multimedia PCs the authors introduced the new MMX technology, which they integrated into their new cpus. It allows the cpu to employ vector instructions to increase throughput and hence optimize running time of common computing tasks, such as rendering pixels for example. They illustrated the use cases in several examples, two of them being:

Vectorizing Branches Say we want overlay sprite onto a scene, i.e. we have a pixel array representing the sprite, and another one representing the scene (so far). The sprite array consists of color values. However, there is one constant value which represents a *clear color*, which represents the sprite not being visible at that point, and hence it should not get rendered onto the scene. The overlaying operation can the be performed by:

```
for i = 1 to Sprite_Line_size
if A[i] = clear_color then
        Out_frame[i] = C[i] else
        Out_frame[i] = A[i]
```

This is a typical example of branching inside a loop, where assignments have the following form:

$$a[i] = expr\_1 \text{ if cond else } expr\_2$$

The idea is that we can vectorize this using vector instructions by:

$$a[i] = (expr\_1 \wedge cond) \vee (expr\_2 \wedge \neg cond)$$

Where `cond` is evaluated to `111...111` if true, and to `000...000` if false. The expressions should be easily vectorizable themselves, such that ultimately the entire statement can be vectorized efficiently.

To see this in action, the authors provided a graphic for the example discussed earlier:
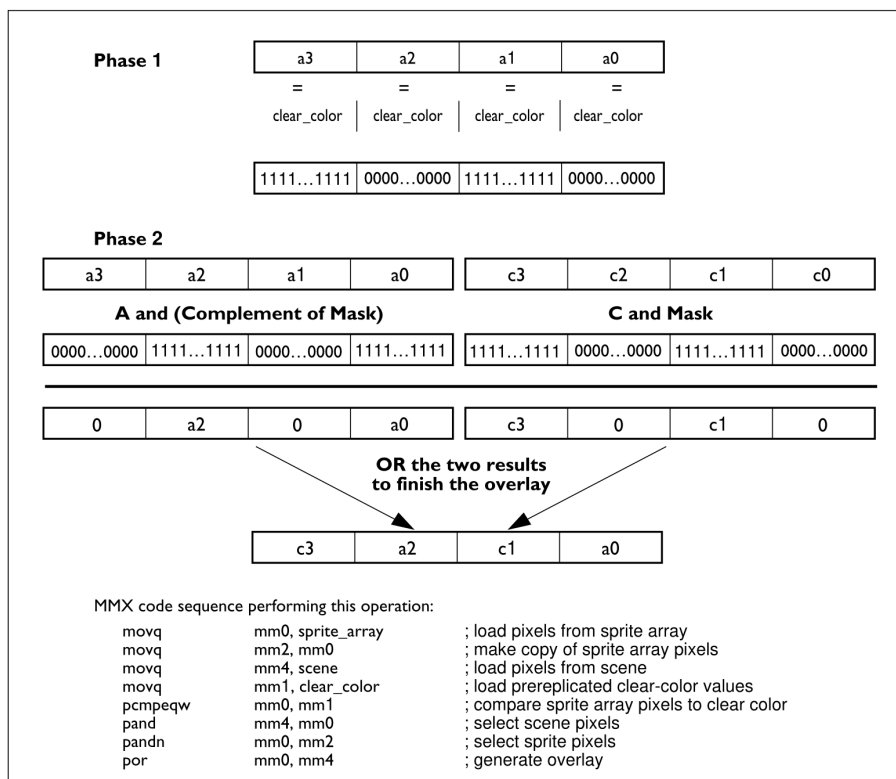
Phase 1

a3	a2	a1	a0
=	=	=	=
clear_color	clear_color	clear_color	clear_color

1111...1111	0000...0000	1111...1111	0000...0000

Phase 2

a3	a2	a1	a0		c3	c2	c1	c0

A and (Complement of Mask)   **C and Mask**

0000...0000	1111...1111	0000...0000	1111...1111		1111...1111	0000...0000	1111...1111	0000...0000

0	a2	0	a0		c3	0	c1	0

**OR the two results
to finish the overlay**

c3	a2	c1	a0

MMX code sequence performing this operation:

```
movq      mm0, sprite_array    ; load pixels from sprite array
movq      mm2, mm0             ; make copy of sprite array pixels
movq      mm4, scene           ; load pixels from scene
movq      mm1, clear_color     ; load prereplicated clear-color values
pcmpeqw   mm0, mm1             ; compare sprite array pixels to clear color
pand      mm4, mm0             ; select scene pixels
pandn     mm0, mm2             ; select sprite pixels
por       mm0, mm4             ; generate overlay
```

Figure 5. Overlay operation using packed compare

In *Phase 1*, the mask is computed, i.e. the condition evaluated (the condition is that the color value equals the `clear_color` constant).

Next, in *Phase 2*, we evaluate the expressions (in this case it is omitted), and mask the results with the appropriate mask (`mask`, and ¬`mask`). Finally, the two results get reduced by a logical or, yielding the final output.

At the bottom of the graphic is a little assembly code section, emphasizing the fact that the operation can entirely be done with vector instructions.

Matrix Transposition Vector instructions can be fully utilized if the data is stored sequentially in memory. Hence, accessing rows of matrices can vectorized. But I we want to work on the columns of the matrix, it might be beneficial to transpose the matrix. This transposing can be vectorized as well by employing the so called *unpack* operation, which interleaves two operands word-wise, as can be seen in the picture:
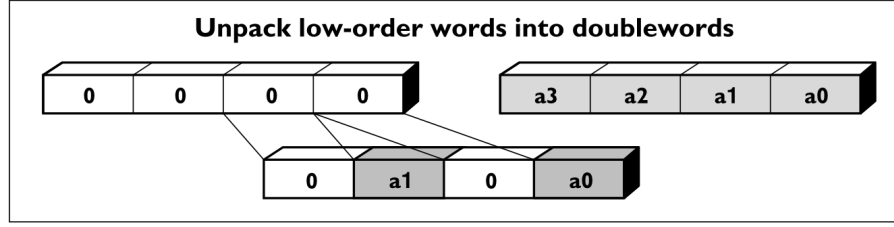
Figure 6. MMX technology Unpack high and low instructions on word data type

To now transpose an entire matrix we go on like this: Imagine for the sake of simplicity that we have the following matrix M:

$$M = \begin{pmatrix} d3 & d2 & d1 & d0 \\ c3 & c2 & c1 & c0 \\ b3 & b2 & b1 & b0 \\ a3 & a2 & a1 & a0 \end{pmatrix}$$

We now compute the last two rows of the matrix by firstly unpacking the last two entries of two adjacent rows, in this case d and c, and b and a, like shown in *Phase 1* in the following graphic:

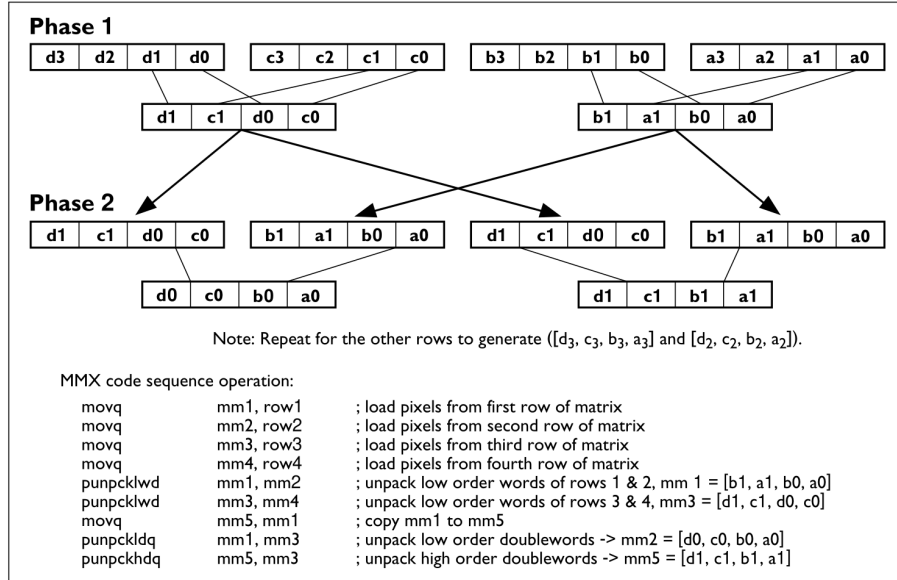**Figure 7.** Matrix transposition using Unpack instructions

We now created two *column slices* of length two in every result of the unpack operations. Thus, we now need to unpack those among themselves, to get longer column slices, or in this case entire columns, as demonstrated in *Phase 2*. Once again, all the operations can be implemented with vector instructions, as hinted by the small assembly program at the bottom.

7 Vector Intrinsics and Instruction-Level Parallelism

7.1 Analyzing slide 14

Slide 14 explains how we can utilize vector instrinsics to transpose 8x8 matrices. Here is the code:

```cpp
// transpose 8 x 8 matrix with intrinsics
inline void transpose_8x8_int_intrinsics(__m256i *vecs) {
    __m256 *v = reinterpret_cast<__m256 *>(vecs);
    __m256 a = _mm256_unpacklo_ps(v[0], v[1]);
    __m256 b = _mm256_unpackhi_ps(v[0], v[1]);
    __m256 c = _mm256_unpacklo_ps(v[2], v[3]);
    __m256 d = _mm256_unpackhi_ps(v[2], v[3]);
    __m256 e = _mm256_unpacklo_ps(v[4], v[5]);
    __m256 f = _mm256_unpackhi_ps(v[4], v[5]);
    __m256 g = _mm256_unpacklo_ps(v[6], v[7]);
    __m256 h = _mm256_unpackhi_ps(v[6], v[7]);
    auto tmp = _mm256_shuffle_ps(a, c, 0x4E);
    a = _mm256_blend_ps(a, tmp, 0xCC);
    c = _mm256_blend_ps(c, tmp, 0x33);
    tmp = _mm256_shuffle_ps(b, d, 0x4E);
    b = _mm256_blend_ps(b, tmp, 0xCC);
    d = _mm256_blend_ps(d, tmp, 0x33);
    tmp = _mm256_shuffle_ps(e, g, 0x4E);
    e = _mm256_blend_ps(e, tmp, 0xCC);
    g = _mm256_blend_ps(g, tmp, 0x33);
    tmp = _mm256_shuffle_ps(f, h, 0x4E);
    f = _mm256_blend_ps(f, tmp, 0xCC);
    h = _mm256_blend_ps(h, tmp, 0x33);
    v[0] = _mm256_permute2f128_ps(a, e, 0x20);
    v[1] = _mm256_permute2f128_ps(c, g, 0x20);
    v[2] = _mm256_permute2f128_ps(b, f, 0x20);
    v[3] = _mm256_permute2f128_ps(d, h, 0x20);
    v[4] = _mm256_permute2f128_ps(a, e, 0x31);
    v[5] = _mm256_permute2f128_ps(c, g, 0x31);
    v[6] = _mm256_permute2f128_ps(b, f, 0x31);
    v[7] = _mm256_permute2f128_ps(d, h, 0x31);
}
```

Let us analyze how we can calculate a specific column, say $v[0]$. It depends on the variables a and e, so let us track them:

We use

```cpp
    __m256 a = _mm256_unpacklo_ps(v[0], v[1]);
```

in the first step (similar for e). Here, we unpack the first two rows, which means we interleave them (compare to last week). *lo* specifies that we focus on the *lower order bits* $(a[\bullet][0], a[\bullet][1], a[\bullet][4], a[\bullet][5])$. Hence, after this operation, a and e contain the following matrix elements:

```
a = {a00, a10, a01, a11, a04, a14, a05, a15};
e = {a40, a50, a41, a51, a44, a54, a45, a55};
```

For the next operation, we calculate *tmp*, for which we need variable c in case of a, so let us quickly do that:

```
c = {a20, a30, a21, a31, a24, a34, a25, a35};
```

Now we can calculate *tmp* with:

```
auto tmp = _mm256_shuffle_ps(a, c, 0x4E);
```

We have

```
0x4E = 0b01001110 (= 1 0 3 2)        ,
```

and hence *tmp* becomes:

```
tmp = {a01, a11, a20, a30, a05, a15, a24, a34};
```

Let us visualize *tmp*:

	1				5
	2				6
3				7	
4				8	

Next, we calculate a using:

```
a = _mm256_blend_ps(a, tmp, 0xCC);
```

Note that

```
0xCC = 0b11001100       .
```

Hence, we get after this blending operation, which selects elements based on the mask:

```
a = {a00, a10, a20, a30, a04, a14, a24, a34};
e = {a40, a50, a60, a70, a44, a54, a64, a74};
```

28

From here, we can see that we are almost done, a and e look like they almost contain one column, in fact the contain two columns "cut in the middle". Hence, after a final permutation we should be done calculating a column. To this end, we use _mm256_permute2f128_ps and specify that we want to use the lower bits of both a and e by setting the mask to CC:

```
v[0] = _mm256_permute2f128_ps(a, e, 0x20);
v[0] = {a00, a10, a20, a30, a40, a50, a60, a70};
```

This explanation was not very detailed, but it gives a guideline to get a *feel* for the algorithm, which might improve one's understanding of this algorithm.

7.1.1 Latency and Throughput

Latency measures the time it takes for an intrinsic function to produce its result once it starts executing, typically expressed in clock cycles.

Throughput indicates the rate at which the function can be executed, showing how many operations can be completed per unit of time (e.g., cycles per instruction) when pipelined.

Obviously, the lower the latency, and the the higher the throughput, the better. In certain situations, however, one metric is much more decisive than the other:

If one writes dependent code, i.e. the next line of code may only be executed after the current one because of a dependency, the latency of the operation will be the main factor for performance, since we want our result as quickly as possible to procede with the execution.

On the other hand, if we write independent code which also uses many repeated operations, throughput will be key, as it determines how many clock cycles we approximately need to finish all calculations. In this case, the latency has a neglectable influence on the performance, especially if the number of operations grows bigger.

7.1.2 Analyzing Vectorized Hash Tables Across CPU Architectures

In this paper the authors explained how to employ SIMD instructions to efficiently implement hash tables. Hash tables are quite diverse, as different implementation approaches can be used. One common implementation employs linear probing to search through the key array to find the specified key, so that the desired value can be returned for reading. Here is how we might implement this using vector instructions:

Vectorized Linear Probing The keys should be integer values, so that they can fit comfortably into the vector registers. Linear probing starts by checking the index specified by the hash value of the current key first, and then keeps going one step at a time until we found our key. The "key" idea is to simply load multiple consecutive keys into one vector register, and to compare them all against the search key. If one comparison is successful, we can calculate the index and return tha value at that index. Otherwise, we keep probing with the next set of keys.

The authors also mentioned that there are subtle challenges, as vector instructions require the data in memory to be correctly aligned (by their register size). When we start probing at the hash index, however, we are almost certainly not at an aligned address. Hence, we should ensure to "align ourselves", for example by loading the aligned memory region containing the first search index into a vector register.

Vectorized Fingerprinting As mentioned, vectorized linear probing require integer keys. So what could we do if wanted to use strings instead? The "key" idea is to calculate "keys" for the keys themself (quite many keys around here). One can think of these keys as hash values (though not necessarily the same as the real hashes) of the keys. To avoid confusion, they are called *fingerprints* instead, and are quite short, in the paper the authors used fingerprints of size 8 or 16 bits. Using this technique, we can use vectorized linear probing on these fingerprints as described earlier, with the only difference being that we might run into fingerprint-conflicts. Thus, after a successful match of fingerprints, we have to then compare the actual keys to determine an actual hit.

8 Cache and Main Memory

8.1 How do bandwidth-bound computations differ from compute-bound computations?

The difference is quite obvious. When running a program, it will run in a finite amount of time (hopefully). But why doesn't it terminate instantly? What a silly question, because a computer has to execute one instruction at a time, and each instruction takes a little bit of time to complete.

Now, we differentiate between two kinds of operations: Usual cpu instructions and memory operations. The reason for this is that memory operations are relatively much more time expensive, and the recent hardware trends make this gap grow bigger. Hence, we would like to know whether our program is bottlenecked by these expensive memory operations, which we very much would like to avoid, as we want to utilize our cpu with useful instructions instead.

To this end, we call computations *memory-bound* iff they are bottlenecked by the latency of memory operations. This, of course, is dependent on many factors, such as hardware and so on.

If our program is not memory-bound, it still executes not instantly, as it is bound by the execution speed of the cpu. We have a *compute-bound*. In a sense, every program or operation experiences a compute-bound. Hence, it is more useful when we quantify this bound, by analyzing the cpu frequency and the execution speed of individual operations, so that we can estimate the program's execution time.

8.2 Influence of temporal locality and spatial locality on program performance

If our program uses a lot of data, and thus faces a memory-bound, we have to intentionally engineer a program that employs good temporal as well spatial locality.

This is, because a lot of data will read from and written to memory, which, as explained, is very time consuming. Luckily, computer engineers introduced caches to this end, which store some data of main memory, and provide it to the program with faster access times. The only problem will be if the caches or, more precisely, their hash bucket are full. Because caches are usually not *fully-associative*, which means that if we have a cache line in memory, we can not write it to any cache line in the cache as we like. In fact, we only have N cache lines available in an *N-way associative* cache.

As we can see, caches are very useful, but we have to understand their behavior in order to specifically tailor a program to effectively utilize them. In fact, a *cache-aware* algorithm employs good temporal as well as spatial locality. But what does that mean?

Temporal locality describes the concept that recently accessed memory has a higher probability to be accessed again.

Spatial locality on the other hand is used to refer to the concept that memory, which is spatially close to recently accessed memory, has a higher probability of being accessed themself.

But why is all that useful? Well, it is due to the *replacement policy* of caches. As explained, caches will run full, and at that instance, they will have to evict a cache line. But how do they decide? Well, one common replacement policy is *LRU (Least Recently Used)*, which means that they choose the cache line whose access is the most far away in time. This is why temporal locality comes in handy, as it increases the chances that our data lies in cache, and thus reduces the risk of a cache miss.

Finally, spatial locality is useful, because, as described, caches hold entire cache lines, which commonly hold 64 bytes of memory. Hence, if the data we access is closely clustered, we reduce the potential number of cache lines required, reducing the space requirement in the cache, and we will have more accesses to our cache lines. All of this contributes to a lower risk of a cache miss, which ultimately reduces the running time of our program.

8.3 Cache Associativity

On the slides 15 and 17 we hear about caches being *N-way associative*. Let's investigate on this topic further. Let's analyze the following picture (from Wikipedia):
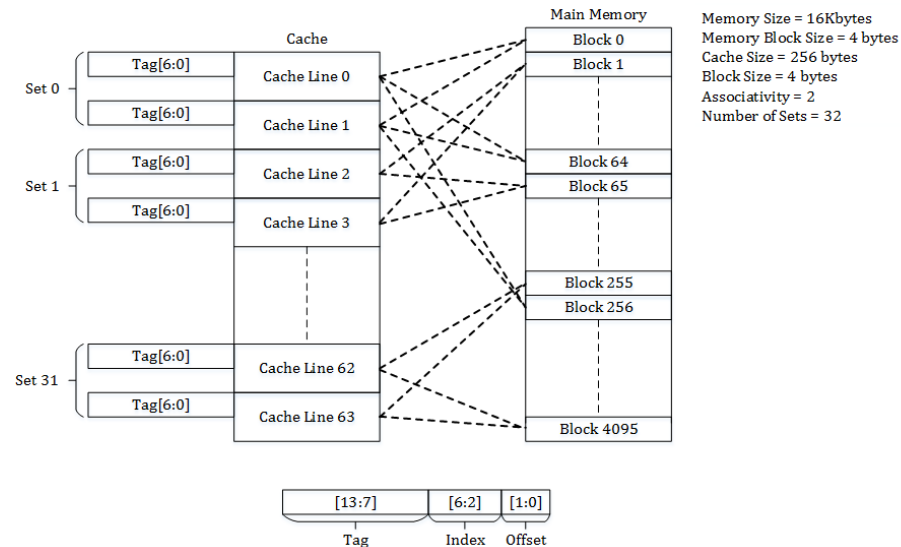

Figure 2:
2-way associative cache [1]

Since this cache is 2-way associative, two consecutive cache lines form a group. For example, cache line 0 and 1 both belong to group 0. If we both know the $cache\_line\_size$ and the $cache\_size$, we can calculate $no\_cache\_lines = \frac{cache\_size}{cache\_line\_size}$. Now, we can calculate the number of sets these cache lines form because of associativity: $no\_sets = \frac{no\_cache\_lines}{N}$.

Now, here comes the clever part. When determining the memory address of the stored data in the cache, we can save 5 bits in this example, as this information is stored in the index of the block. More precisely, we have $no\_index\_bits = \log_2(no\_sets)$. The offset determines the exact location inside the cache line, and hence $no\_offset\_bits = \log_2(cache\_line\_size)$.

Finally, the remaining bits that are needed to uniquely identify the memory address are grouped into the *Tag*. How many bits do we need exactly? Well, here is one explanation: In total, there are $no\_address\_bits = \log_2(memory\_size)$ bits needed. But we already have *index* and *offset* bits, so there are $no\_tag\_bits = no\_address\_bits - (no\_index\_bits + no\_offset\_bits)$ bits remaining.

8.4 An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms

In this paper the author explains the importance of cache-aware programs, as compilers won't help us much in this regard. Hence, let's investigate on some practical examples where we can improve temporal and / or spatial locality.

Loop Blocking This is best understood by an example. Say we have the following code:

Algorithm 10.3.3 Loop blocking for matrix transposition	
1: // *Original code:*	1: // *Loop blocked code:*
2: **for** $i = 1$ **to** n **do**	2: **for** $ii = 1$ **to** n **by** B **do**
3: **for** $j = 1$ **to** n **do**	3: **for** $jj = 1$ **to** n **by** B **do**
4: $a[i,j] = b[j,i]$;	4: **for** $i = ii$ **to** $min(ii + B - 1, n)$ **do**
5: **end for**	5: **for** $j = jj$ **to** $min(jj + B - 1, n)$ **do**
6: **end for**	6: $a[i,j] = b[j,i]$;
	7: **end for**
	8: **end for**
	9: **end for**
	10: **end for**

Both matrices a and b are stored in row-order fashion. Hence, when running the left algorithm, a has good spatial locality, whereas b's is horrendous, as it leads to a cache miss on every access. The idea is now to find a tradeoff between these two. Let us analyze the access pattern of the two algorithms:

---

[1]Image source: https://en.wikipedia.org/wiki/Cache_placement_policies#/media/File:Set-Associative_Cache_Snehal_Img.png
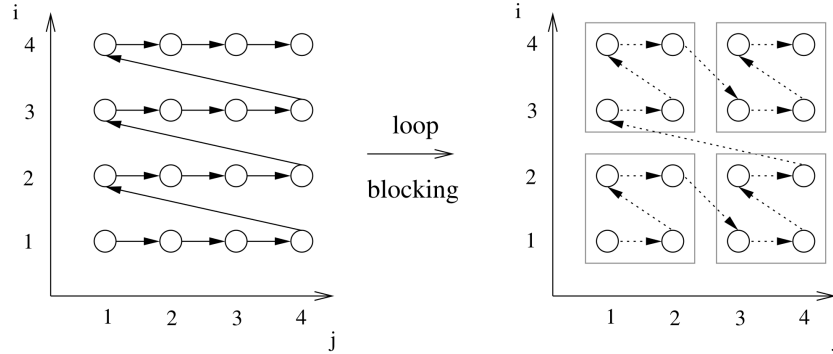
Note that walking along the rows is good for spatial locality, and also note that for b the graphic should be flipped along $y = x$.

So why is the right access pattern better? Note the following: for a, the spatial locality does get a bit worse, but only in $O(1)$ fashion, as now we have to maybe load two more cache lines into the cache, not really a big deal. But what about b? Well, it basically divides its cache misses by a factor of 2 (in case of a block size of 2)! Thus, the overall cache misses will be reduced.

Array Padding This one is rather specific, but it is quite useful to keep it in the back of one's mind. Imagine for the sake of argument that we have cache which can hold 1024 doubles in total, and has a 1-way associative cache (we will now see why this is maybe a bit too aggressive). Next, consider this algorithm:

Algorithm 10.3.4 Inter-array padding.

1: *// Original code:*	1: *// Code after applying inter-array padding:*
2: double $a[1024]$;	2: double $a[1024]$;
3: double $b[1024]$;	3: double $pad[x]$;
4: **for** $i = 0$ **to** 1023 **do**	4: double $b[1024]$;
5: $\quad sum{+}= a[i] * b[i]$;	5: **for** $i = 1$ **to** 1023 **do**
6: **end for**	6: $\quad sum{+}= a[i] * b[i]$;
	7: **end for**

It just so happens that in this instance both array a and b will map to the same cache lines (compare to my previous explanation). But, as there is only one slot in the cache for a specific cache line with a certain index, we will load a cache line of array a, right after that we will evict it for b, and then again for a and so on. In the end, we will have a cache miss on every access, yikes. If only the cache line indices of a and b wouldn't align. Well, exactly this is the idea behind the algorithm on the right: We simply occupy some padding space in-between the two arrays, which the cache will be grateful for.

This example was obviously very specific, but we can see that we might have to be a bit more careful if we had large arrays, maybe multiple of them, which in memory are separated by a large power of 2.