# Crash course introduction to parallelism: Multithreading

April 10, 2022Multithreaded Performance, Parallelization, PerformanceLeave a Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

This is the third post in the *crash course introduction to parallelism* series . The first post was about parallel algorithms, more specifically, what kind of algorithms are possible to speed up using parallelization. The second post is about SIMD parallelism, a specific type of parallelism that uses special CPU instructions to process more than one data in a single instructions.

In this post, we talk about multithreading as another form of parallelism. So what is multithreading? **Multithreading is the ability of the software and the operating system to take advantage of additional CPU cores, by splitting the workload into several independent parts and performing calculations on them independently on each core**.

In this post we are interested in how we can use multithreading to speed up our software. We will first introduce multithreading, then talk about hardware support for multithreading, and finally we will talk about the programming application interface (API) you can use to take advantage of additional CPU cores. So let's go!

## The basics of multithreading

We introduce here the concept of *software thread* and *thread synchronization*. If you are familiar with them, skip to the next section.

### The example

Imagine the following problem. You have an unsorted array of 10M elements and you are counting all the elements with value zero (0). The simplest algorithm would look like this:

```
int find_zeros(int input[], int n) {
   int found_zeros = 0;
   for (int i = 0; i < n; i++) {
       if (input[i] == 0) {
           found_zeros++;
       }
    }
    return found_zeros;
}
```

The algorithm is trivial. It goes from 0 to `n` (length of the array) and checks if the value of `input` is zero (line 5). If it is zero, it increases the value of variable `found_zeros`. The function returns the number of elements with the value zero.

To count the number of zero elements on a 10M element array, we would call the function like this:

```
int num_zeros = find_zeros(input_array, 10 * M);
```

Now let's say we have in our system, not one, but two CPUs that are able to execute programs. What we could do, we could split the lookup process on 10M elements into two lookup processes each of which would work on 5M elements. The two independent lookup processes can run simultaneously (in parallel), and since they run on smaller inputs, they would finish faster. When both are done, we can get the final number of zero elements by summing the values of zero elements for the left and the right half. Here is the pseudocode:

```
thread_t left_search = spawn_thread(find_zeros(input_array, 5 * M));
thread_t right_search = spawn_thread(find_zeros(input_array + 5 * M, 5 * M));
// Both threads are now executing simultaneously (in parallel)
left_search.wait_to_complete();
right_search.wait_to_complete()
int num_zeros = left_search.result + right_search.result;
```

The function `spawn_thread` (lines 1 and 2) creates a *software thread:* an independent thread of execution that executes our function `find_zeros`. Notice that we created two threads, and each thread is working independently on its own part of the workload.

In the next step we need to wait for both the `left_search` and the `right_search` threads to finish their calculations. We do this by calling the `wait_to_complete` functions for both of them (lines 6 and 7). Finally, we calculate the result by adding together the numbers of zero elements in the left side and the number of zero elements in the right side.

## Fundamental building blocks of multithreading

With this, we introduced the two fundamental building blocks of multithreading: *(software) threads* and *synchronization*. Software threads run independently; speedup comes from splitting the workload among them. Programs can create threads and assign them work.

Synchronization means the need for the threads to somehow communicate or wait for one another to achieve a certain goal. In our example, calling `wait_to_complete`[1] creates a synchronization point in the program. The program sleeps until the threads completes, then it can resume. There are a few reasons why one would want to create a synchronization point in a program:

- Wait for a thread to finish.
- Wait for a signal from some other thread that something has happened.
- Make sure that the data is modified atomically. If one thread starts modifying an object, it must not be interrupted by another thread, otherwise the object can be left in undefined state (half of the object modifed by one thread, and half of the object modified by another thread).

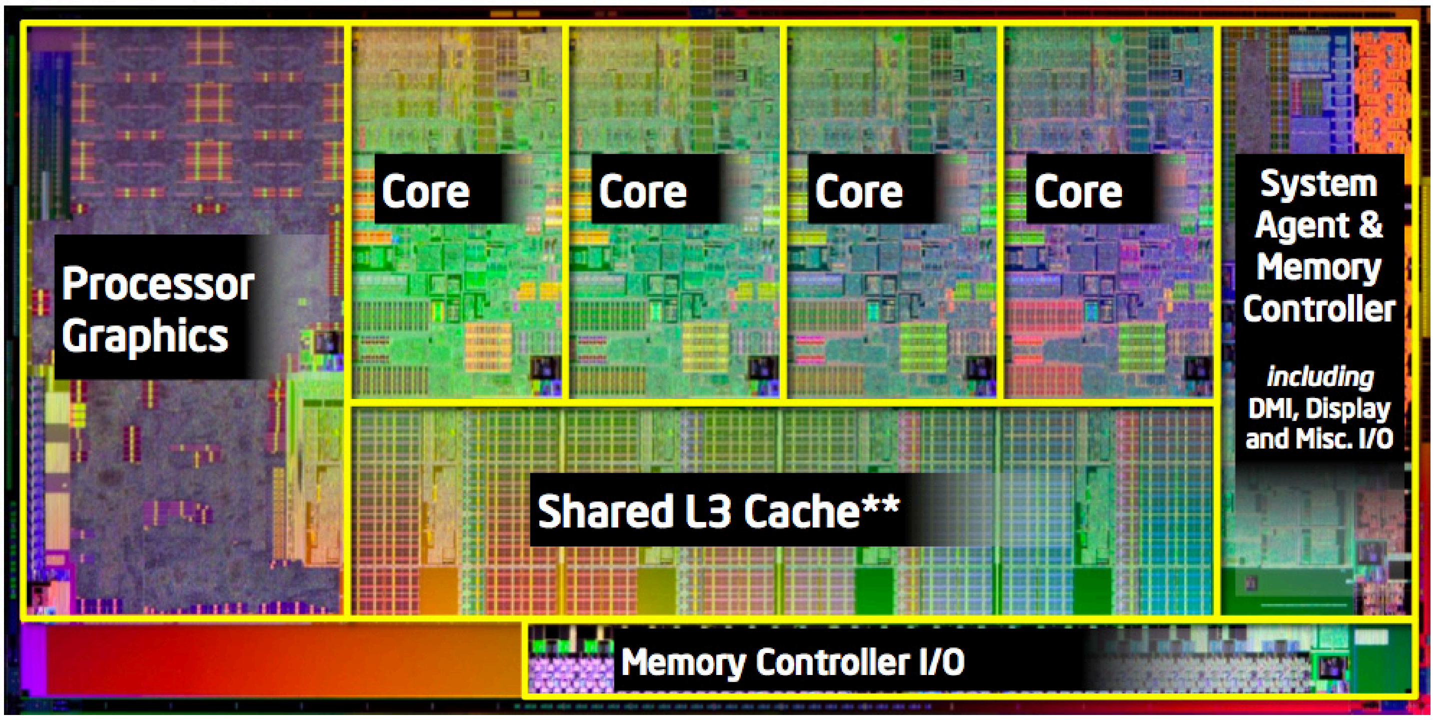
These were the basic building blocks of multithreading, now let's get onto how multithreading is implemented in hardware.

## Multithreading in hardware

If you look up the specification of your computer's CPU, you will come across terms like *socket, cores* and *threads.* Here we explain what these terms mean and what is their importance for parallel programming.

### Socket and core

A *socket* is what we would call a physical chip. It is often called a *package*. You can take it and hold it in your hand. It is called like this is because you can take a CPU package and plug it into a socket in a motherboard. Systems with several sockets are not typically found in home desktops, laptops and small embedded systems. Servers, on the other hand, often feature several sockets.



*CPU Package with 4 cores (source)*

A CPU package can have one or more cores. In the image above we see a CPU package with four cores. Each core has most of its own resources (integer units, floating point units, dividers, L1 cache, L2 cache), but some resources are shared (notably L3 cache). Inside a single package, CPU cores can be all identical, or some of them can be different. In the above image all the cores are identical, but, e.g. Qualcomm Snapdragon 808 CPU used in Android phones and tablets has 2 fast cores (optimized for speed) and 4 slow cores (optimized for power consumption).

### Hardware threads

Each core consists of one or more (typically two) hardware threads (often called only “thread”). A hardware thread executes a software thread created by the program. All the hardware threads in the core share most of the core’s resources (integer units, floating point units, L1 cache, etc). Each thread has its own register set, which is necessary for mapping of a software thread to a hardware thread.

The idea behind hardware threads is that many workloads use only some of the CPU core resources, leaving most other core resources idle. A workload might only be using floating-point unit leaving integer unit available for other workloads. Running a floating-point workload on hardware thread 1 and integer workload on hardware thread 2 will not result in a slowdown. But, running two identical workloads on two hardware threads belonging to the same core will result in a slowdown.

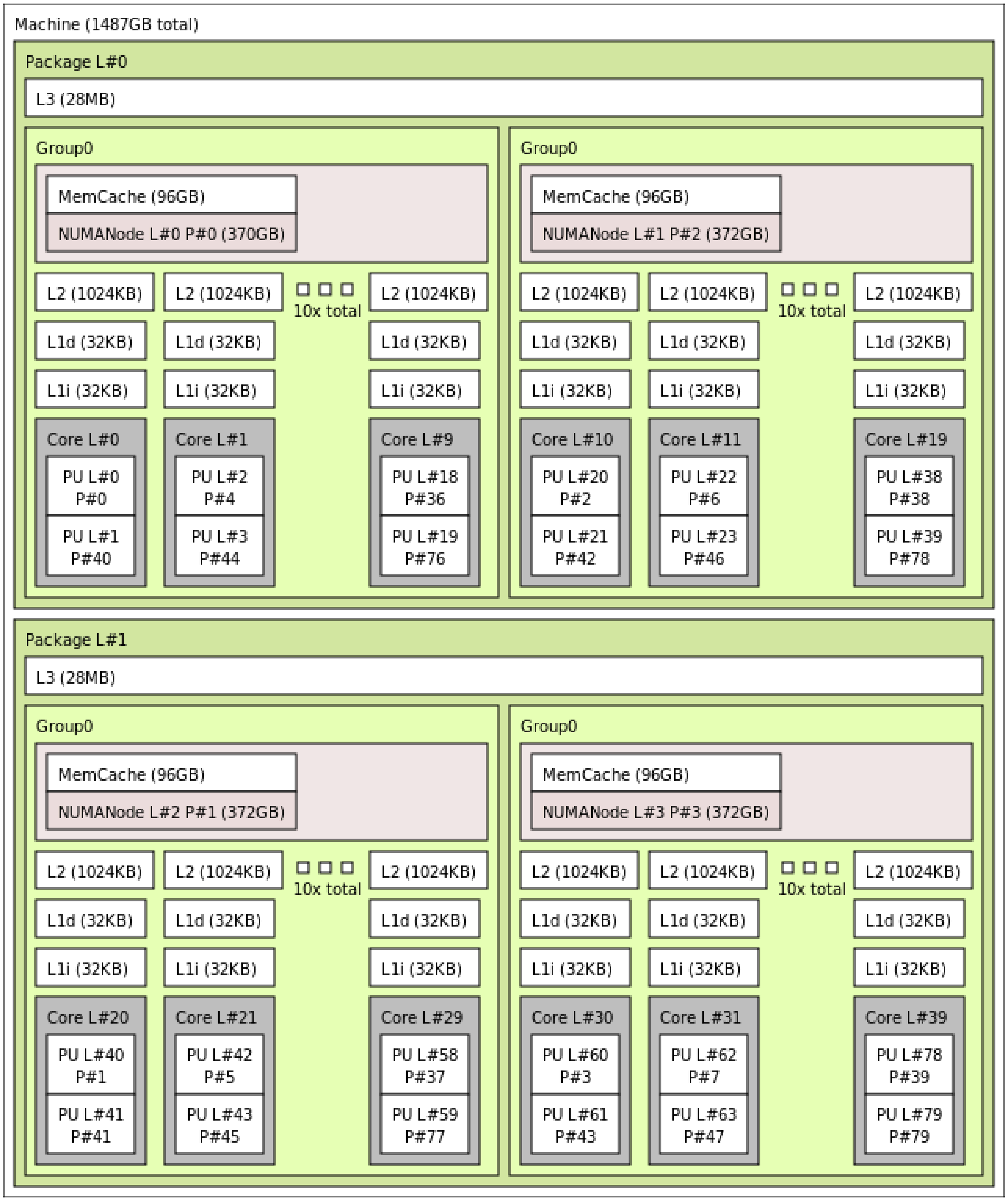
From the aspect of software and the operating system, a hardware thread runs a software thread. A system that can run 8 software threads in parallel can be e.g a CPU with 8 cores and 1 HW thread on each core, or 4 cores and 2 HW threads on each core, or something else. From the programmer’s point of view, there is no difference.

**Attention!**

In the early days of computing, there was one CPU that had one core that had one thread. In the meantime, the complexity of those systems grew. So there is a lot of confusion about terminology, mostly using CPU to mean either CPU core or CPU thread. For example, a text might say “the loop uses CPU resources efficiently” but the actual meaning is “the loop uses CPU core resources efficiently”. Luckily, the meaning can be usually easily deduced from context if you are familiar with CPU terminology.

## NUMA domain

The last term of multithreading is *NUMA domain*. NUMA stands for *Non-Unified Memory Access* and it means that the system is divided into several domains. Inside a domain, there is a certain amount of memory and some computational resources (e.g. a core, several cores, a package or several packages). If a CPU core is accessing memory in the same domain, this access is faster than accessing memory in other domains. Here is an example of a system with four NUMA domains:



*Hardware Locality Topology for a system with 2x Xeon CascadeLake 6230 (source)*

The image shows a system with two CPU packages. Inside each package there are two NUMA domains, to make a total of four NUMA domain. Each NUMA domain consists of 10 cores, so a single package carries 20 cores. And each core consists of two threads. From the software point of view, this system can run 80 software threads simultaneously.

NUMA domains are not typically seen in embedded systems and workstations, but they can often be seen in servers and other high-performance systems.

In systems with Non-Unified Memory Accesses, the place where the data is stored plays a role in performance. Therefore a special care needs to be taken when allocating memory. The worse case scenario is that all the memory the program is using is allocated in the single NUMA domain. Since all other cores are accessing memory in this domain, the memory bus can quickly get saturated which leads to low performance.

## Software threads and hardware threads

A software thread runs on a hardware thread. If there are 8 hardware threads, this means that at most 8 software threads will be running simultaneously. The program can spawn more than 8 software threads, but all except eight of them are sleeping and waiting for their turn to execute.

To the user, the system can appear to run more than 8 software threads, but what is in fact happening is *preemption*. The operating system is running one software thread for a short amount of time (e.g. a few milliseconds), and then it is passing the hardware thread to another software thread that was sleeping. In this way it achieves the illusion of having more hardware threads that there actually are. Of course, if the number of software threads is much higher than the available hardware threads, the system will become less responsive.

## The use cases for multithreading

There are two basic scenarios for multithreading:

- **A software component will create a software thread for its own data processing**. The main purpose of multithreading in this scenario is component decoupling: several components can run their computations independently without disturbing one another. This keeps the component's source code simpler and easier to maintain.
- **Multithreading is used to speed up computation**. In this case, there is no benefit for code readibility or component decoupling. In fact, multithreading represents a "liability" in the sense it makes software development more complex. But for very computationally-intensive workloads this is often done.

In this post, we focus on the second: how is multithreading used to speed up programs.

## How can we speed up our program with multithreading?

To speed up a computation, a typical scenario would look like this:

- If the system has N hardware threads, split the workload into N smaller independent parts.
- Spawn N software threads and have each thread solve its own part of the workload.
- Wait until all the threads are done with their work.
- (Optional) Sometimes, the final solution is calculated by combining the output of all threads into a single solution.

This workflow is very generic.[2] However, if we look at it carefully, we see a few possible problems:

- **The software needs to spawn software threads**. Spawning software threads is not cheap, and the result is that multithreading doesn't pay off unless the workload is large enough.
- **While waiting until all threads are done with their work**, it can happen for various reasons that **one software thread gets delayed**. Since we need to wait for all software threads to finish, because of this delay, the whole computation will be delayed. This is solved by splitting the workload into more than N pieces. N threads will work on the problem, but the workload pieces are not assigned in advance. Instead, workload pieces are stored in a queue and the first thread that has done its piece takes another piece of work from the queue.
- **Waiting requires some kind of thread synchronization**. How do we know when all the threads are done with all the work?
- **In same cases the workload can be split only when the computation is ongoing**. In that case, the threads are spawned as the workload is being processed. How is this done?

The ways these problems are solved actually depends on the multithreading API we use. Here we list the most useful.

## How is multithreading supported in software?

There are several paradigms multithreaded programming is supported in various programming languages. But since we are talking about C/C++ in this blog, we will focus on the mechanisms available there. Nevertheless, many programming languages figure some or all of these concepts.

### Raw threads

This kind of API is the most low-level and it is the basic block for building other models. In C, the name of the standard is POSIX and is supported on almost all operating systems. Starting from C++11, there is a C++ wrapper for POSIX threads in `std::thread`.

Here is the list of the most important API calls:

| **API Name** | **POSIX Threads** | **`std::thread`** |
|---|---|---|
| Create thread | `pthread_create` | `std::thread::thread` |
| Wait until the thread has finished | `pthread_join` | `std::thread::join` |
| Fire an event for another thread | `pthread_cond_signal` | `std::condition_variable::notify_one` |
| Wait until other thread fires an event | `pthread_cond_wait` | `std::condition_variable::wait` |

| API Name | POSIX Threads | `std::thread` |
|---|---|---|
| Protect the section of the code so no other thread can access it before the thread executing is done | `pthread_mutex_lock` `pthread_mutex_unlock` | `std::mutex::lock` `std::mutex::unlock` |
| Yield the hardware thread to the operating system | `pthred_yield` | `std::this_thread::yield` |

*POSIX Threads API (C and C++) vs C++11 API*

As you can see, C and C++ APIs are very similar and have the same name for basic operations.

This API is very powerful, but it gives the developer the bare minimum. There is nothing related to workload distribution, synchronization between the threads at the end of calculation, etc.

## OpenMP

OpenMP is a multithreading API that utilizes special compiler pragmas (e.g. `#pragma omp parallel`) to tell the compiler on how to spawn threads, split the workload among the threads and combine the independently calculated results into a final result. The API is very declarative in nature: you need to tell it only the bare minimum, and it does all the rest: thread spawning, workload sharing, synchronization, etc.

Before giving a quick introduction to OpenMP, we need to cover an important prerequisite. The critical problem of multithreading is the division between *thread-shared* and *thread-private* variables. Thread-shared variables are common to all the threads; in contrast, each thread has its own copy of the thread-private variables.

Thread-shared variables are cheaper to construct, because there is only one copy of them. But changing them is more expensive, since it can require some kind of exclusion mechanism. Otherwise, two threads might simultaneously modify the value which can result in value corruption. Thread-local are more expensive to construct (this applies mostly to arrays, not simple variables), but changing them is cheap. For good performance, the program should strive to minimize modifications of thread-global variables.

Here is an example of the loop with OpenMP compiler pragma:

```
#pragma omp parallel for shared(a, b, n)
for (int i = 0; i < n; i++) {
    b[i] = 0.1 * a[i];
}
```

The pragma tells the compiler that variables `a`, `b` and `n` are shared among all threads. Variable `i` is special, because the OpenMP runtime uses it to distribute the workload. Each thread will be given a range of values for variable `i` that is independent of other threads. So, writes to `b[i]` do not require any synchronization, since each thread has its own independent values for `i`.

Thread spawning happens when the before the CPU starts executing this loop. So does the workload sharing. The user only has to declare which variables are shared and which ones are private.

OpenMP is supported in almost all compiler. In GCC and CLANG it is enabled with `-fopenmp` compiler switch. It is supported in C, C++ and Fortran. A great OpenMP tutorial can be found here.

*Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.*
*Need help with software performance? Contact us!*

## Transform-Reduce

Another very important multithreading paradigm is called *map-reduce* or *transform-reduce*. Originally coming from the domain of servers and high-throughput data processing, it can be used to speed up workloads through multithreading.

There are two basic operations: *transform* and *reduce* and if you manage to represent your problem using these operations, the problem can be safely and efficiently parallelized. In C++, these two are supported in STL through `std::transform`, `std::reduce` and `std::transform_reduce`. Consider the following loop:

```
for (int i = 0; i < n; i++) {
   b[i] = a[i] + 1.0;
}
```

These types of loops correspond to `transform` operation, because the input data is transformed to output data using some transformation function. The same operation, written using `std::transform` would look like this:

```
std::transform(a.begin(), a.end(), b.begin(), [] (double c) -> double { return c + 1.0; });
```

The iterators `a.begin()` and `a.end()` tell the range of input values it should work on. The iterator `b.begin()` is the place where it should write its outputs, and the lambda expression `[] (double c) -> double { return c + 1.0; };` specifies the type of transformation.

The second operator is `reduce`, which is called like that because it "reduces" an array of values to a single value. Consider the following example:

```
double sum = 0.0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

This example can be represented using the `reduce` operator, like this:

```
sum = std::reduce(a.begin(), a.end(), static_cast<double>(0.0), std::plus<double>());
```

Again, the `reduce` operator takes the beginning and the end of the input arrays using iterators `a.begin()` and `a.end()`. It also takes the start value for the accumulator variable `sum`. And lastly, it takes the reduction function, in our case this is `std::plus<double>`.

The main advantage of this approach is that both `transform` and `reduce` are trivial to parallelize. Starting from C++17, you can specify their execution policy to parallel, which will make these functions use multithreading paradigm.

There are certain loops that cannot be represented using the transform-reduce paradigm, e.g:

```
for (int i = 1; i < n; i++) {
    b[i] = b[i - 1] + 1.0;
}
```

The processing of values of `b[i]` depends on `b[i - 1]` and this cannot be represented by transform operator. This loop has an implicit ordering, the variable `i` going from 0 to `n`. The transform operator doesn't guarantee ordering and cannot be used here.

## Multithreading enabled standard library

The last building block of multithreading is the standard library with multithreading support. Starting from C++17, many algorithms in the standard library have gotten their "parallel" versions, including `std::sort`, `std::find`, `std::replace`, `std::copy`, `std::count_if`, `std::for_each`, etc. To use the new algorithm, you call it in the same way as the old algorithm, except for a new parameter `execution_policy`. Here is an example of parallel sorting:

```
std::sort(std::execution::par, v.begin(), v.end());
```

The parameter `std::execution::par` generates a call to a multihreaded-enabled version of `std::sort`.

If you need thread-safe and efficient data structures that can benefit from multithreading, you will need to look further than the standard library provided by C++. For example, efficient concurrent hash maps are provided as part of Intel's Thread Building Blocks library, Microsoft's Parallel Pattern Library and HPX C++ Standard Library for Concurrency and Parallelism. They allow thread-safe insertion, deletion and lookup and can be used to speed up working with the hash map in a multithreaded environment.

## A few final notes about performance and multithreading

We introduced the topic of multithreading in this post and in the earlier post we talked in great length about how to make multithreading systems more performant. But in the essence, here are two most important rules if you want to actually see the performance increase in your multithreaded code.

The first important rule is related to the workload size. **Small workloads to not profit from multithreading**. Spawning threads is costly, and so is synchronization. Sorting an array of 100 elements will not be faster with multithreading. If you are looking for ways to speed up small workloads, vectorization is a much safer bet.

The second rule relates to thread synchronization: **avoid thread synchronization as much as possible**. Every time two threads needs to content for a memory location or any other resource, one thread will have to wait, which in the essence means reverting back to serial execution. A shared variable protected by a mutex and accessed by all the threads all the time is a definite performance killer, as it has been very skillfully illustrated by ITHare in a post called "Using Parallel Without a Clue: 90x Performance Loss Instead of 8x Gain".

## Final Words

With the stagnant single core performance and increasing number of cores in a CPU package, multithreading is already an important aspect of software performance. Simply there is no way to avoid multithreading if you want to process large amounts of data in a reasonable time.

Multithreaded programming is not however easy. Threads, mutexes, synchronization, workload size, etc. all add to the complexity of programming. Transform-reduce API or using parallel libraries will make parallel programming much easier, but on some occasions there is no running away from more complex APIs. But the complexity of parallel programming is its beauty and what makes parallel programming fun!

*Like what you are reading? Follow us on LinkedIn , Twitter or Mastodon and get notified as soon as new content becomes available.*
*Need help with software performance? Contact us!*

1. What we called `wait_to_complete` here is typically called `join`. [↩]
2. But it is not the only one. Sometimes, we cannot split the workload at beginning. In that case, we split the workload as we are solving the problem and opportunities for workload splitting appear. The prominent example of this type of workload split is quicksort. [↩]