Efficient Parallel C++

Before talking about parallel code, we want to redefine some common vocabulary, just so we are talking about the same things. We call a (whole) processor with its multiple processing units a *socket*. We use this name because it clearly distinguishes the whole entity from its single processing units which we call *cores*.

Each execution path of your programs is called a *thread*. Threads can be scheduled by the operating system and can usually be executed on any of your machines cores.

1 Parallel Programming

1.1 OpenMP

OpenMP is mainly used to parallelize simple tasks that are close to or fully independent. Most OpenMP instructions are pragmas, which look like preprocessor commands. Therefore, you can often make your code run in parallel by just adding some annotations. This is also nice because a lot of the time the same code will compile into sequential code when you remove the OpenMP flag.

OpenMP is a quick and comfortable option, because you often do not have to think about work scheduling and thread specifics (like thread pinning).

1.2 Standard Threads (`std::thread`)

Since C++11 there is standard support for parallel execution. You can manually create threads that will execute a given function. The underlying implementation will usually use your system's libraries to communicate with your operating system and create the threads (on Linux this means threads are usually created with the pthread library).

async vs. thread Aside from manually creating threads (with the `std::thread` constructor) you can also create concurrently running tasks with `std::async`. The `async` function will return a future that you can wait on in order to use the result of your function call. The idea behind this is that background tasks can potentially be done by a concurrently running threadpool. The problem is that this is not guaranteed, in fact there is no guarantee that anything will happen in parallel.

In theory, a valid implementation of `async` could start nothing, and once the main thread waits for the result the calculation would be done by the main thread. This is usually not what we want in our exercises, so stay away from `async`. Since C++17 there are other parameters that can fix this behaviour, but creating new threads manually is simpler and less misunderstandable.

threadpool Oftentimes, your program can quickly switch between parallel sections and sequential sections. This means you spend a lot of time on creating and rejoining new threads. This is problematic, especially if you generate new threads within your time measurements. A common solution to this problem is the use of threadpools. The idea is that you create some threads at the beginning of your program's execution. These threads are then waiting for tasks that will commonly be created by a main thread. Sometimes, the main thread is only used to synchronize between the participating threads.

2 Thread Safety

No matter how you start your threads, you have to make sure that access to shared variables cannot lead to unwanted behavior. Both OpenMP and the STL have possibilities to ensure correct computation.

atomics (`std::atomic<T>`) In some sense, atomics are the simplest and the hardest way to ensure correct computation. Atomics ensure that the object can never be in an undefined or intermediate state during an operation. This means that when thread A is concurrently accessing the protected object, while thread B manipulates it, then thread A will either see the state before the manipulation, or the state after the manipulation.

Not every kind of object can be atomic – essentially, only basic data types can. There are several functions that simplify the use of atomics: `store`, `load`, `fetch_add`, `compare_exchange_strong` (and `weak`), etc. Most operators are overloaded to use these atomic functions – some people advise

against using the operators, since they hide the fact that these functions may not be cheap.

The speed of the atomic functions highly depends on the number of threads changing the same element. The speed also depends on the optional memory order parameter. You can achieve higher performance by using different memory orders, but generally you should stick to the default (i.e., sequentially consistent). This is the safest and messing with the other orders will probably give you more hidden bugs than performance. If you want to learn more about this, I can recommend the talk *Lock-Free Programming (or, Juggling Razor Blades)* by Herb Sutter at CppCon (you can find it on YouTube).

compare and swap Compare-and-swap or `compare_exchange_...` is easily the most flexible and powerful atomic operation. Calling `x.compare_exchange_strong(y, z)` will atomically execute `x = (x==y) ? z : x` and it will even store the current `x` in `y` if the comparison was false. This can be used for many different things, but most notably it can be used to ensure that you only change `x` if it has not changed since you last read it.

The difference between the weak and strong variants is that the weak one is allowed to fail, even if there have not been any changes. This is oftentimes not so bad, especially since compare-and-swap operations are often called within a loop that will retry the operation until it succeeds.

busy waiting Busy waiting is a technique in which a thread repeatedly checks a variable until it changes, either to protect access to a critical section, or for other synchronization purposes. If used right, busy waiting can be simple to implement and fast. Note that there are cases where busy waiting can lead to bad behavior, especially when one variable is very contended (a lot of threads want access) and the same variable (or something in the same cache line) is changed a lot.

mutexes Whenever a resource needs to be locked and busy waiting (i.e., spin-locks) is not an option – usually when processor time should not be wasted because other threads can do something productive – using a mutex can create a system call that makes the thread sleep until the lock is unlocked again. Usually a mutex implementation will first try to do some busy waiting, before it resorts to this system call.

deadlocks You have to think about deadlocks whenever you have a number of different locked resources. This fact is independent of the locking method. The easiest solution is to always acquire locks in a predefined order. This could also be achieved by sorting the mutexes by their memory-address and acquiring them in this order.

3 Parallel Optimization

3.1 Cache Effects

In exercise 2 you have (hopefully) seen some performance aspects that happened due to cache effects in a sequential program. Even in sequential programs, cache effects can be complicated and somewhat hard to estimate. They only get more complicated with higher numbers of different processors, especially when looking at multi-socket server architectures.

Lets first look at a normal (current Intel) computer with one socket (one processor) but multiple cores on that processor. Each core has its own L1 and L2 cache but all cores together share one L3 cache.

The second, more uncommon scenario for you is multiple sockets (2, 4, or 8 are possible). Here a single socket behaves as described above, but two cores that are not on the same socket do not share any caches. There is also some need for communication between two sockets.

positive cache sharing Whenever cores share a common cache they can profit from data that the other core has loaded during its computation. This is similar to the positive cache effects that happen in sequential programs. The main difference in this scenario is that for two execution threads to profit from this scenario, they have to be scheduled to two cores on the same socket (to share the cache), **and** they cannot change the data, otherwise the following problem might reduce their benefits.

false sharing When two caches have a memory line and one core wants to change that line, then it will first take exclusive control of that line. This is important to ensure that there are no two different copies of that line floating around the system. The other copies of said line are evicted from their positions. This is especially costly in multi-socket scenarios where the communication to evict lines from caches can take a long time.

False sharing leads to two different things. First, you should not have small data elements that are continuously and quickly changed by many threads at once, even if changes do not have to be atomic or locked for some reason. Second, things that change quickly should not be in the same cache line with other data that is read/changed by other threads.

C++11 and newer include the `alignas(`x`)` *`type`* feature, which tells the compiler to treat the created instance of the `type` as if its type had a size of x. This can be used to block one whole cache line by declaring a type `alignas(64)` (usually one cache line has 64 bytes). `alignas` can also be used within arrays.

3.2 NUMA

Another problem that luckily only appears in multi-socket scenarios is Non-Uniform Memory Access (NUMA). Each socket has its own connection to the main memory. In fact each socket has its own part of the main memory! Other parts of the memory are accessible but slower, because read elements have to be communicated between sockets.

Most NUMA systems are configured such that memory is located in the main memory of the thread that first writes to it. Specifically allocating memory on one socket can be done with libnuma, but it is somewhat tedious.

3.3 Hyperthreading

Modern processors often support multiple threads (usually 2) to run on the same core at the same time (without being interrupted by scheduling). Even though only one command is executed at each moment in time, there can be increased throughput since memory latencies of one thread can be used for computations in the other. However, it is clear that executing two things on the same core does not give you the same performance increase as executing them on different cores.

3.4 Thread Pinning

Usually, the system is relatively smart about scheduling your threads. For example, it prefers scheduling two threads on different cores, if possible. But if you want to purposefully use or reduce the above effects, you should always pin your threads to cores – you can then argue where these effects might happen.

On multi-socket systems, pinning threads might even improve sequential performance. If you are not pinning your threads, it is possible that data is read by your program while it was scheduled on one socket, but your program was rescheduled to another socket before the computation. Then each access goes to the slower memory of the first socket (see the section on NUMA).

4 Guidelines

- Always be thread-safe: Use either atomic operations or locks (prefer atomics).
- When using locks, prefer busy waiting for small things, and mutexes for bigger things.
- Think about cache effects in parallel scenarios (positive and negative ones).
- Multi-socket scenarios are complicated, and often require extensive experiments to optimize.