Exam Assignments 2

False Sharing

False sharing is a phenomenon of parallel computing. It happens when multiple threads of program access different data that lie close to each other in memory. More accurately, the data lie on the same cache line. (A cache line is the smallest unit of memory that can be transferred to the cache.)

Each thread loads the cache line into their cpu cache, but as soon as one of them modifies the contents, the other needs to update it's copy so as to maintain cache consistency, even though the data that was modified isn't even used by the second thread. This overhead of updating cache contents can cause a significant performance decrease.

Race Conditions and Mutual Exclusion

A race condition is a phenomenon of parallel computing. It describes a situation in which the order of execution of threads (which the programmer usually can't control) has an effect on the output of the program.

For example, this will happen if multiple threads try to access a memory address at the same time, with at least one of those accesses being a `write`. This kind of race condition is also called a **data race**, because the outcome of the execution is determined by which thread gets access to the data first.

Mutual exclusion structures are designed to prevent race conditions. They are used in programming to designate a region of code as a critical region that can only be accessed by one thread at a time. This allows the programmer to make sure that all actions that would be potentially dangerous if performed concurrently (such as access to shared data) are instead performed sequentially.

Static and Dynamic Schedules in OpenMP

When one writes `#pragma omp for`, OpenMP will execute the following for loop on multiple threads by splitting it up and handing each thread some iterations. Schedules determine the way OpenMP maps loop iterations onto threads.

Using `schedule(static)` (the default choice), OpenMP will map onto each thread the same predetermined number of iterations. The mapping is already known at compile-time. One

can influence the number of threads that get dealt out. `schedule(static, 4)` will cause every thread to receive four iterations. If, after giving out four iterations to each thread, there are still iterations left, the first thread will get another four iterations, then second, and so on.

Using `schedule(dynamic)`, OpenMP will do the mapping dynamically at run-time. Once a thread is done with it's previous work, it will get the next chunk of iterations. The size of the chunks can be chosen aswell. `schedule(dynamic, 8)` will make it so that chunks of 8 iterations are dealt out together.

Dynamic scheduling can improve performance by increasing parallelism if the workloads in different iterations are very different. The runtime nature of dynamic scheduling does however result in a significant overhead, leading to performance decreases if workloads are fairly even across iterations.

Getting out of an OpenMP For Loop

For loops that have been parallelized by OpenMP can't be prematurely exited. To speed up the remaining iterations, one can try to minimize the time spent in each of them. A boolean variable `is_finished` could be checked in each iteration. If is is true, then the program can simply continue to the next iteration; otherwise, more work needs to be done.

Coding Warmup: What Schedule produces the bad pattern?

The pattern shows that 22 iterations were allocated to `thread 0`, 10 iterations were allocated to `thread 1` and none to the other two threads. This pattern can be caused both by static and dynamic schedules, when the chunk size is set too large. In this case, the chunk size was set to 22, forcing `thread 0` to take 22 iterations at once, leading to an uneven distribution of iterations and less parallelism.

Coding Warmup: How does `std::atomic::compare_exchange_weak` work?

When trying to update an atomic based on some condition in a parallel program, one runs into a problem:

> What if, after I checked that the condition is met in one thread, some other thread modifies the atomic, invalidating my conditional check? The other thread could

> have changed the value of the atomic in a way that it doesn't satisfy my condition anymore; but I will update it regardless.

The methods `compare_exchange_weak` and `compare_exchange_strong` provide a solution to this problem. This is their signature:

```
atomic.compare_exchange_weak(T& expected, T desired);
atomic.compare_exchange_strong(T& expected, T desired);
```

They both compare `expected` to the actual value of the atomic. If they're equal, the atomic is updated to contain `desired` and `true` is returned. Otherwise, `expected` is updated to contain the actual value and `false` is returned.

This allows the programmer to constantly check wether the condition still holds using the updated value of `expected` and only continue with the code once the method returns `true`.

The only difference between the weak and strong versions is that the weak version has superior performance but can spuriously fail even if the expected and actual values are the same. This isn't a problem in looped applications of the method.