Algorithm Engineering Lab Assignment 2

Brian Zahoransky (brian.zahoransky@uni-jena.de)

February 22, 2021

1. What causes false sharing?

False sharing may appear if two processes are going to write to the same cache line at once. For further explanation, consider the following example. A program should show that floating-point operations are not associative by summing up a large number of floating-point values. First, an array is initialized containing the values to be summed. Second, the variables sum1 and sum2 are initialized with zero. Both steps happen while the program runs serially. Assume that both variables, sum1 and sum2, are stored in the same cache line. sum1 will store the result of all values summed up in the right order. sum2, on the other hand, will hold the sum of the float array summed up in inverse order. To speed up the program, each summation runs on an own thread, on a different processor core. Actually, there is no speedup. More likely is a slow down because of false sharing. After every update of sum1 and sum2, the cache line, stored in the cache of the other processor, becomes invalid. Thus, the updated cache line has to be sent to the other processor before it can proceed. Since a huge array of floats is assumed, most of the time, the processors will wait for a valid cache line.

2. How do mutual exclusion constructs prevent race conditions?

Race conditions occur if multiple threads try to access one resource at once. In the context of storage, a race condition occurs if various threads write to the same memory address. For example, multiple threads performing an operation on one memory address. Fist, thread A loads the value of the global variable, performs its operation. Now, tread A stops for some reason, maybe because the system decides to give B computational time, but other scenarios are plausible, too. However, B also loads an old copy of the global variable, performs its operation, and writes the update back into storage. Afterwards, thread A gets computational time again. It stores its result of the global variable and overwrites the changes B has made.

To avoid such behaviour **mutual exclusions** (mutex) can be used. They allow only one thread to access a critical resource. Consider the example above. Thread A wants to update the global variable. Therefore, it receives the access right by locking the mutex. Note that locking and freeing the mutex is an atomic operation; thus, it cannot be interrupted. If the computational time stops now, no other thread can change the value of the global variable. Even if B proceeds, it stops before updating the global variable. However, sometimes tread A gets computational time again. It can perform its operation, write back the result, and free the mutex. Now, B can lock the mutex, gets the new version of the global variable, performs its operation, and stores the result.

3. Explain the differences between static and dynamic schedules in OpenMP.

static: At compile time is the schedule fix. Thus, each chunk is assigned to a specific thread after the compile procedure. It is applied when each chunk holds nearly the same amount of work. The software engineer may consider using a static schedule if the workload differs a lot and can be estimated well.

dynamic: The chunks are assigned to specific threads at run-time. Apply this schedule if the workload for the chunks is unknown or the chunks cannot be divided similarly. However, performing the scheduling task at run-time causes performance penalties. Consider that when choosing a schedule.

4. What can we do if we have found a solution while running a parallel for loop in OpenMP, but still have many iterations left?

One possible solution uses two atomic variables. One boolean initialized with false and a solution variable. The boolean will be changed to true once a solution has been found. The first line of the parallel for loop is an if-clause. Its condition is the atomic boolean. Once the boolean is true, each iteration happens nothing but the evaluation of the if-clause and the following "continue" instruction. The search proceeds while the boolean is false. Hence, the complete code of the if-clause is executed. An if-clause, later on, checks if a solution has been found. Is that the case, the atomic boolean is set to true, and the result is written into the atomic solution variable.

5. Explain in your own words how std::atomic::compare_exchange_weak work.

value.compare_exchange_weak(expected_value, desired_value);

Given the code line above, the atomic variable *value* should become overwritten. Therefore, *value* is compared with *expected_value*. If *value* and *expected_value* are equal, the *desired_value* is written into *value* and the function returns true. If *value* and *expected_value* are not equal, the *desired_value* is written into *expected_value* and the function returns false. The second scenario may occur when another thread has changed the *value* after the *expected_value* was set, but before the compare_exchange_weak function has been invoked.