Algorithm Engineering Lab Assignment 4

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

January 15, 2021

1. Explain how divide and conquer algorithms can be parallelized with tasks in OpenMP.

The goal of divide-and-conquer algorithms is to generate a ordered list out of a randomly sorted one. The algorithm can be described through a recursive function. This function receives a not ordered list of elements. If the length of this list is one, it is ordered and is returned. Otherwise the list of elements is divided into parts. Now the function calls itself for each sub list. After this phase, the function holds ordered sub lists. The sub lists become merged to generate a fully ordered list. The result is returned. The algorithm can be parallelised through the help of tasks. A new task starts if the function calls itself as long as the length of the lists is above a set value. This mechanism ensures that the overhead for starting new tasks does not exceed the work for sorting a list.

2. Describe some ways to speed up merge sort.

Alternative sorting algorithms for small sub-lists: For example, insertion-sort is faster as merge-sort for small lists. Due to this, it was used in the lecture for lists with less than 32 elements.

Storing in stack instead of heap: The storage for stacks is faster and limited. Therefore, lists are stored in stacks when their length was below 8192 in the lecture.

Multiple Task algorithms: To enable a multiple task version of merge-sort, a additional function is used. This function opens a parallel region and invokes merge-sort. This and the omp single instruction within the merge-sort function avoid that each piece of work is processed from multiple tasks. For further descriptions of multi-threaded merge-sort, have a look on exercise 1 and exercise 3.

In-place algorithms: The approach is working with two arrays of the same size. One array holds the elements which are to be sorted. The other one is a buffer. Both arrays are declared before the first call of the merge-function. Thus, there is no need for creating new arrays during the execution of merge-sort.

3. What is the idea behind multi-threaded merging?

A recursive function gets two sorted sub-lists A and B, is merging them into a sorted list, and returns it. Through a lookup on the longer sub-list, its median M is determined. Now, the two lists become divided by M in A_smaller_M, A_bigger_M, B_smaller_M, and B_bigger_M. Afterwards, the function calls itself, once with A_smaller_M and B_smaller_M, and once with A_bigger_M and B_bigger_M. The results are two sorted lists M1 and M2. M1 contains only elements smaller as M, and M2 only elements bigger as M. Concatenating M1, M, and M2 generates a fully sorted list. As described in exercise 1, tasks can be started each time the function calls itself.

4. Read What every systems programmer should know about concurrency (https://assets.bitbashing.io/papers/concurrency-primer.pdf). Discuss two things you find particularly interesting.

In the tenth chapter, memory orderings are discussed. A programmer is able to give each atomic operation an optional order-clause. The paper also describes use cases. However, the following only explains the procedures with the help of pseudo-code:

doWork1()
barrier1()
atomicOperation()
barrier2()
doWork2()

memory_order_seq_cst: It stands for sequentially consistent order, and is set per default. Above, a pseudo code example is shown. To improve performance, the compiler and processor are only allowed to reorder the operations within the blocks doWork1() and doWork2(). However, they are not allowed to reorder work through the barriers. First, doWork1() has to be done, then the atomic operation, and last doWork2().

memory_order_release: This clause removes the barrier1(). It allows compiler and processor to reorder doWork1() and the atomic operation.

memory_order_acquire: It is usually the counterpart to the release clause in a second thread. A further explanation can be found in the paper. Anyway, this clause keeps barrier1() and removes barrier2(). Thus, doWork2() and the atomic operation may be shuffled by compiler or processor to increase the performance.

memory_order_acq_rel: Both barriers are kept. The difference compared to the sequential consistent case is described in the paper as follows, "acquire-release provides order relative to the variable being load-acquired and store-released, whereas sequentially consistent operation provides some global order across the entire program."

memory_order_consume: It is the alternative counterpart for release and works similar to acquire. The difference between consume and acquire is the interpretation of the kept barrier. The consume clause let operations pass the barrier as long as they are not dependent to the atomic variable which was affected by the atomicOperation().

memory_order_relaxed: This removes both barriers. One can use it if the operation has just to be atomic but order does not matter. Thus, compiler and processor are allowed to reorder doWork1(), atomicOperation(), and doWork2() to speed up the program.