Threaded Programming coursework II: the affinity schedule for scheduling the OpenMP loop construct

Abstract

Keywords: Scientific programming, parallelization, performance optimization, OpenMP

1. Introduction

OpenMP version 4.5 supports various scheduling options for its loop construct, for example `static`, `dynamic` or `guided` (see OpenMP Architecture Review Board, 2015, Chapter 2). This paper presents an alternative schedule, called the affinity schedule. The affinity schedule combines some properties of the three above mentioned scheduling options into one schedule.

This paper is a follow-up of a benchmark presented in B160509 (2019). B160509 (2019) presents a scientific program written in the Fortran programming language, containing two loops performing matrix and vector operations. These two loops were parallelized using built-in scheduling options of OpenMP version 4.5 and then benchmarked in order to determine the best schedule for the two loops. The here presented affinity schedule is benchmarked the same way, as are the built-in schedules in B160509 (2019).

This paper begins by describing two versions of the affinity schedule. Afterwards the benchmark is described and its results are presented. The benchmark of the affinity schedule is then compared to the best schedule for both loops determined in B160509 (2019). At last the results are discussed and a conclusion is drawn.

2. Method

Like stated in the previous chapter, the affinity schedule combines properties from the already built-in schedules `static`, `dynamic` and `guided`. Let n be the amount of iterations of the loop the instance of the affinity schedule is applied to and let p be the amount of OpenMP threads. The affinity schedule splits the n iterations of the loop into p splits, all having approximately $\frac{n}{p}$ iterations. This is the same way the `static` schedule splits the iterations, if no chunk size is provided (see OpenMP Architecture Review Board, 2015, Chapter 2).

Every split is owned by one OpenMP thread. But, unlike the `static` schedule, the split is not executed as a whole. Rather, it is split again into smaller chunks, which are executed after another. The size of the chunks gets smaller, the more iterations of the split are already executed:

$$chunk\ size := \lceil remaining\ iterations \cdot p^{-1} \rceil.$$

The decreasing chunk sizes are a property of the `guided` schedule. While the `guided` schedule takes all n iterations and dynamically assigns the splits in a first come first serve order to the OpenMP threads (like the `dynamic` schedule), the affinity schedule uses the same property of decreasing chunk size, just local to each split (see OpenMP Architecture Review Board, 2015, Chapter 2).

Once a thread has finished all chunks of the split it owns, it is not simply idle until all threads have finished their splits. Instead, the thread determines the split that has still the most iterations left and takes the next chunk from it. Therefore, once the owned split of a thread is finished, the thread changes from behaving like it is part of the `guided` scheduling strategy on the local split to being `dynamic` in the fact that the split with the most iterations left is dynamically executed by all threads that have finished their split, plus the owner thread.

The affinity schedule therefore combines the strengths, concerning the execution speed, of the aforementioned built-in schedules over the phases of its execution. It starts executing like the `static` schedule, which produces very low overhead of synchronization (in fact none, but the affinity schedule must synchronize access to the split), moving over to a `dynamic` scheduling strategy, which has a higher overhead of synchronization, but produces no idle threads. This effectively reduces the synchronization overhead of the `dynamic` schedule while also removing the idleness of threads during the execution with the `static` schedule.

The affinity schedule is a shared object among each thread. It contains the amount of OpenMP threads p, determined by `omp_get_num_threads`, an array of size p containing the splits and an array of the same size containing OpenMP locks (integers of the kind `omp_lock_kind`). The locks are used for the synchronization of the access to a split (see OpenMP Architecture Review Board, 2015, Chapter 3). Every OpenMP thread has a unique id p_i from the sequence $1, 2, \ldots, p$ and owns the split from the split array at the index p_i.[1] A split is nothing but a tuple of two non-negative integers. The two elements are the remaining iterations of the split and the index where the next chunk taken from the split starts. A chunk is a synonym for a closed interval $[i, j]$, which is a subset of the whole loop executed by the affinity schedule, which—in this case—can be thought of as the interval $[1, n]$. The thread that gets the chunk $[i, j]$ executes the loop from i to j, inclusively, before taking the next chunk, either from its split, or, if the split is already finished, from the split with the most iterations left.

The affinity schedule provides two methods as its interface, called by each thread. The methods are `init` and `take`. `init` initializes everything and returns the id to every thread. It sets p, allocates the split and the lock array and initializes the content of both. While allocation and the setting of p are done by a single thread, the initialization of the splits and

1. OpenMP's `omp_get_thread_num` function is used for determining the id. `omp_get_thread_num` returns ids from the sequence $0, 1, \ldots, p-1$. Since the id is used to access splits from the splits array of the affinity schedule and since Fortran arrays by default start indexing at 1, the returned id by `omp_get_thread_num` is incremented by 1 (see OpenMP Architecture Review Board, 2015, Chapter 3).

locks are done by every thread concurrently. That means, the thread with id p_i initializes the split in the split array at index p_i. The same is done with the lock in the lock array, simply by calling `omp_init_lock` (see OpenMP Architecture Review Board, 2015, Chapter 3). Every thread must leave `init`, only after all threads have finished initializing their split and lock. Otherwise, if a very fast thread has already finished his split, while other threads are still not finished with initializing, the already finished thread could possibly access an uninitialized split, which would result in a segmentation fault. Therefore, at the end of `init`, all threads are synchronized with a `barrier` (see OpenMP Architecture Review Board, 2015, Chapter 2).

TODO: take

This paper describes two different versions of the affinity schedule. The versions differ in their strategy for determining the split, which has the most iterations still to do. The two versions are called `naive` and `queue`. The `naive` affinity schedule simply uses brute force, iterating every split and find the one with the most iterations left, while the `queue` affinity schedule uses a priority queue based on the max-heap data structure, which is constantly updated, every time a chunk is taken from the split (for the max-heap data structure see e.g. Cormen et al., 2009, Chapter 6).

3. Results

4. Discussion

5. Conclusion

References

B160509. Threaded Programming coursework I: benchmarking OpenMP schedules, 2019.

Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, Cambridge, Mass., 3rd ed.. edition, 2009. ISBN 9780262033848.

OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.