B160509

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

Abstract

This paper describes an alternative schedule for scheduling OpenMP's loop construct, the affinity schedule. The affinity schedule tries to add positive properties of different built-in schedules together into one schedule. This paper is a follow-up on a benchmark presented in B160509 (2019).

Two versions of the affinity schedule are presented and compared against each other. This paper describes in detail how the two versions of the affinity schedule are implemented. Afterwards they are benchmarked and compared against well performing built-in schedules determined in B160509 (2019).

While the two versions of the affinity schedule perform well for the most parts and sometimes even outperforming the built-in schedules, this paper comes to the conclusion that the affinity schedule simply is too complex and contains too much synchronization, which hurts its scalability. An idea for a better performing affinity schedule is presented.

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, trying to combine the strengths of the schedules and canceling out their weaknesses.

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:

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

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 (a weakness of the `static` schedule). 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, freeing the owner thread to become `dynamic` as well. This is continued until all splits are executed.

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 $1, 2, \dots, p$ 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.

Algorithm 1 : executing a loop with the affinity schedule

```
 1: start OpenMP parallel region with schedule being shared
 2: initialize schedule with init
 3: while true do
 4:    get interval from take(schedule, $p_i$)
 5:    if the interval is valid then
 6:       execute subset of loop defined by the interval
 7:    else
 8:       exit while loop
 9:    end if
10: end while
11: end OpenMP parallel region
```

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[2], the initialization of the splits and 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). Algorithm 2 shows the `init` method.

1. OpenMP's `omp_get_thread_num` function is used for determining the id. `omp_get_thread_num` returns ids from the sequence $0, 1, \dots, p{-}1$. 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).
2. The `single` construct is used for this (see OpenMP Architecture Review Board, 2015, Chapter 2).

Algorithm 2 : `init`

```
1: set the $p_i$ of the calling thread to omp_get_thread_num + 1
2: enter OpenMP single block
3: set $p$ with omp_get_num_threads
4: allocate the arrays for the splits and for the locks
5: leave OpenMP single block
6: initialize split and lock in the arrays at the index $p_i$
7: if the schedule is a queue affinity schedule then
8:     initialize the priority queue and its lock as well
9: end if
10: wait here till all threads have finished (realized with the OpenMP barrier structure)
```

After a thread leaves `init`, it starts executing chunks. The thread does not know or care, where the chunks are coming from, as long as they are valid. It has no concept of owning a split. The ownership relationship is only implemented in the affinity schedule and a thread is reduced to its id and the basic task of executing chunks. Algorithm 1 shows how the threads execute when the affinity schedule is used.

In order for a thread to get the next chunk for execution, it calls the `take` method of the affinity schedule instance. `take` takes the id of the thread as an argument. It first looks at the split owned by the calling thread. If it has already finished, it instead tries to get a chunk of the split with the most iterations left. Before accessing any split, the corresponding lock in the lock array must be set with `omp_set_lock`, so the current thread has exclusive access to the split and race conditions are avoided. After a split is accessed, the lock is unset with `omp_unset_lock` (see OpenMP Architecture Review Board, 2015, Chapter 3).

The result of `take`, on an abstract level, can be thought of as an instance of an algebraic sum type, which returns either a valid interval or nothing (for sum types see e.g. Chad Austin, 2015).[3] If nothing is returned (in this case the boolean field `is_none` would be true), then all splits have finished and the thread can stop its execution (see Algorithm 1, lines 5ff). Otherwise it executes the returned interval and calls `take` again, until it returns nothing (see Algorithm 1, line 4). Algorithm 3 displays `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

3. In languages like Rust and Swift this type is called the `Option` type (see The Rust Team, 2019; Apple Inc., 2019). Fortran unfortunately does not support sum types. So the actual implementation is a wrapper around an interval with a boolean field indicating, whether the returned interval is valid or not.

Algorithm 3 : `take`

```
1: set the lock in the lock array at index $p_i$
2: if the split in the split array at index $p_i$ is not finished then
3:     take chunk from the split at index $p_i$
4:     unset the lock at index $p_i$
5: else
6:     unset the lock at index $p_i$
7:     get id of the thread owning the split with the most iterations left $p_{most}$ (either naive
       or with the queue)
8:     set the lock in the lock array at index $p_{most}$
9:     take chunk from the split at index $p_{most}$
10:    unset the lock at index $p_{most}$
11: end if
12: return the taken chunk (can be nothing if the split with the most remaining iterations
    is finished)
```

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).

Both implementations have weaknesses. While the `naive` version is, in theory slow, because the lookup requires the calling thread to iterate all splits, the priority queue used by the `queue` affinity schedule is not thread-safe and therefore requires a lot of synchronization, which reduces its in theory faster lookup of the split with the most iterations left. For this reason, the access to the priority queue must be locked with an additional lock, so only a single thread can access the priority queue at any given moment. The performance of both versions is compared in the following chapters.

The priority queue provides two main methods as its interface (and some more minor methods for constructing it), `max_element` and `decrease_key`. It consists of three arrays, all the size of p. One array is the heap, containing the remaining iterations of each split (see Cormen et al., 2009, Chapter 6). The second array—the lookup array—contains the index of the split, corresponding to its key in the heap. For example, if the split owned by the thread p_3 is the split with the most remaining iterations of 15, the first element of the heap would be 15 and the first element of the lookup array would be 3. The lookup array is for fast access to the split that has the most iterations left. The `max_element` method simply returns the first element of the lookup array.

After a thread takes a chunk from a split, the remaining iterations of the split are decreased. The decreased key must be updated in the priority queue. The third array of the priority queue is the reverse lookup array for fast access during the `decrease_key` operations. Lets continue the example from above: a thread takes the next chunk of size 3 from the split owned by p_3, respectively decreasing the remaining iterations of that split to 12. `decrease_key` takes p_3 and 12 as its arguments. Now, in order to find the heap element which corresponds to p_3, we could iterate over the elements of the lookup array until we find the element that is equal to p_3. This is not very elegant or efficient, so the priority queue maintains the reverse lookup array. So, instead of iterating over the lookup arrays, instead the reverse lookup array at p_3 returns the position of p_3 in the heap, in this example 1 (see Algorithm 4, line 1). Afterwards the 15 in the heap at index 1 is decreased to 12 and the heap property is restored (if 12 is smaller than any of its children, they are recursively swapped until no children are bigger than 12 (see Algorithm 4, lines 8ff)). Finished splits (where remaining iterations are zero), are automatically removed from the heap (see Algorithm 4, lines 2ff).

Algorithm 4 : `decrease_key`(p_i, key)

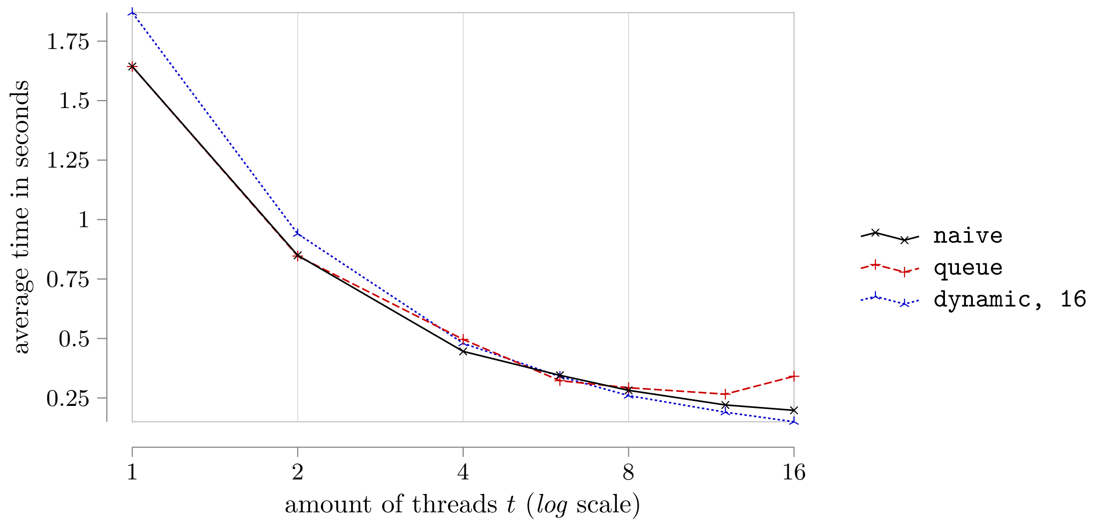
```
1: get heap and lookup index $i$ from the reverse lookup array at index $p_i$
2: **if** $key = 0$ **then**
3:    swap places with the last element of the heap
4:    decrease the heap size
5:    restore heap property (see Cormen et al., 2009, Chapter 6)
6: **else**
7:    decrease value in heap at index $i$ to $key$
8:    **while** the decreased element has children that are greater **do**
9:       swap the element with its biggest child
10:   **end while**
11: **end if**
```

The benchmark presented in the following chapters is the same as the second part of the benchmark presented in B160509 (2019). The two loops of the scientific program are executed by affinity schedule instances on a back end node of the Cirrus supercomputer with exclusive access (see EPCC, 2019). The program is run with 1, 2, 4, 6, 8, 12 and 16 OpenMP threads, five times per amount of threads. The timing of the execution is done with the `omp_get_wtime` routine (see OpenMP Architecture Review Board, 2015, Chapter 3).

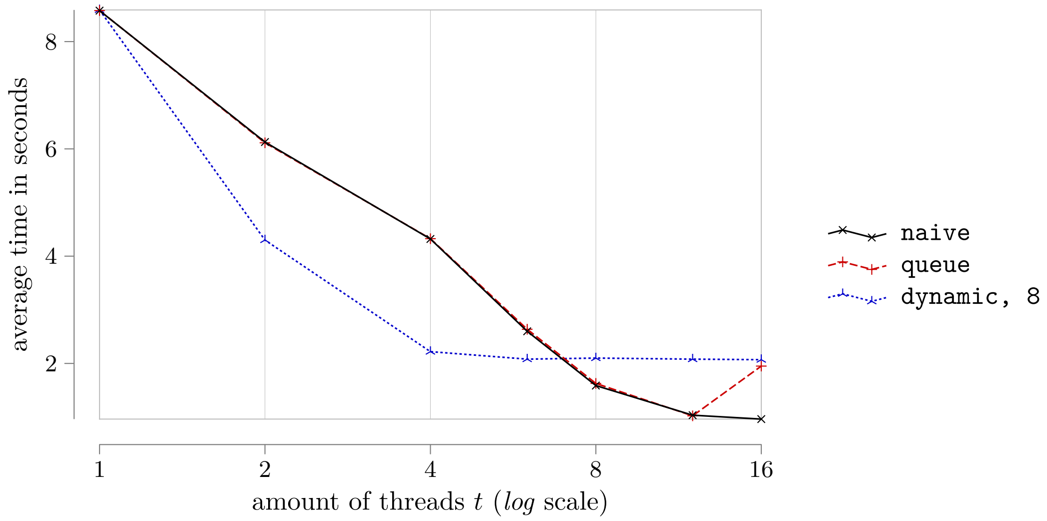
Both versions of the affinity schedule are tested. They are compared against each other and the results of the best built-in schedules for the two loops of the program, determined by the benchmark from B160509 (2019).

3. Results

The results here presented are grouped by each of the two parallelized loops of the benchmarked scientific program. The loops are thoroughly described and analyzed in B160509 (2019). Both loops are asymmetric when it comes to the distribution of the computational complexity. Both loops are more complex at the beginning, which means chunks closer to 1 take longer to finish then chunks closer to n. $n = 729$ for the benchmark (see



(a) Loop 1.



(b) Loop 2.

Figure 1: Average execution time in seconds, taken per tested amount of threads.

B160509, 2019). While for the first loop complexity decreases linearly, for the second loop the complexity at the beginning is much higher and decreases more extreme (see B160509, 2019).

Figure 1a shows the average execution time for both affinity schedule versions and the best built-in schedule determined in B160509 (2019) for loop 1: `dynamic, 16`. Figure 1b shows the same for the second loop.

Both best built-in versions come from a preselection of schedules tested on the two loops of the scientific program. The two best schedules, `dynamic, 16` for the first and `dynamic, 8` for the second loop, were selected based on their performance when run with 4 threads. B160509 (2019) argues, that, especially for the second loop, a `dynamic` schedule with a smaller chunk size, would scale better.

One can see in Figure 1a that the `naive` affinity schedule performs better than the `queue` version. For both loops, the `queue` affinity schedule performs as good as the `naive` affinity schedule for the most parts, except for the tests with 16 OpenMP threads, where the performance of the `queue` affinity schedule very much deteriorates, resulting in timings worse than using it with 8 threads. The `naive` affinity schedule on the other hand has no such scaling problems. It produces a monotonic increasing function over the different amounts of threads tested.

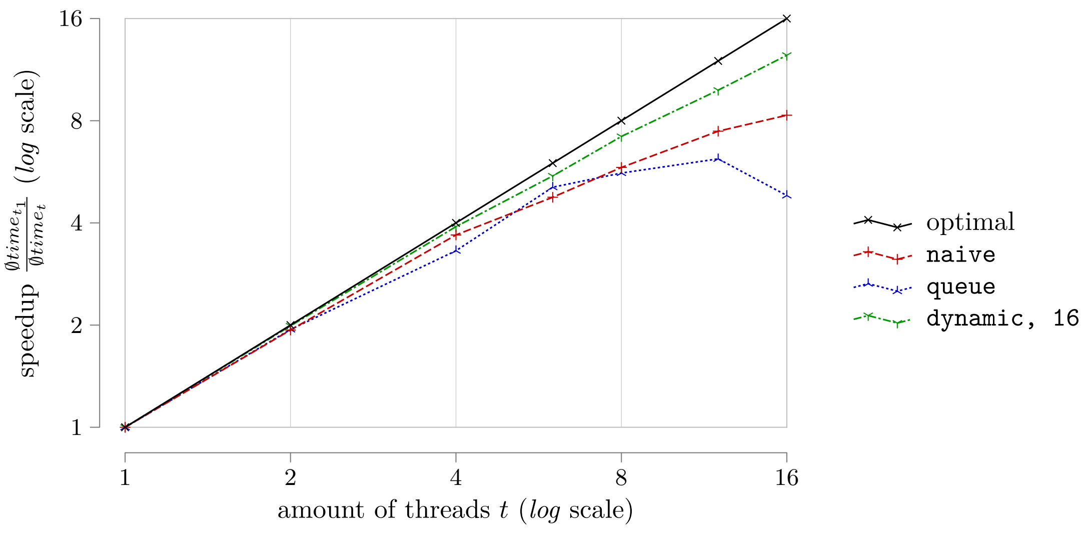
When the results of the affinity schedules are compared to the best built-in schedules for both loops, they behave exactly the opposite for both loops. For the first loop, the two affinity schedule versions are faster than the built-in schedule for lower amounts of threads (1, 2, 4, 6). With 8, 12 and 16 threads `dynamic, 16` performs better than the affinity schedules. For the second loop, this behavior changes. `dynamic, 8` performs better with 1, 2, 4 and 6 threads, while the affinity schedules outperform it with 8, 12 and 16 threads.

Figure 1 shows the performance of the schedules, Figure 2 shows their scaling capabilities by displaying the speedup of each schedule when using more threads. Figure 2a shows that the affinity schedules do not scale as well as `dynamic, 16` for the first loop. Again the deterioration of the `queue` affinity schedule for 16 threads is shown.

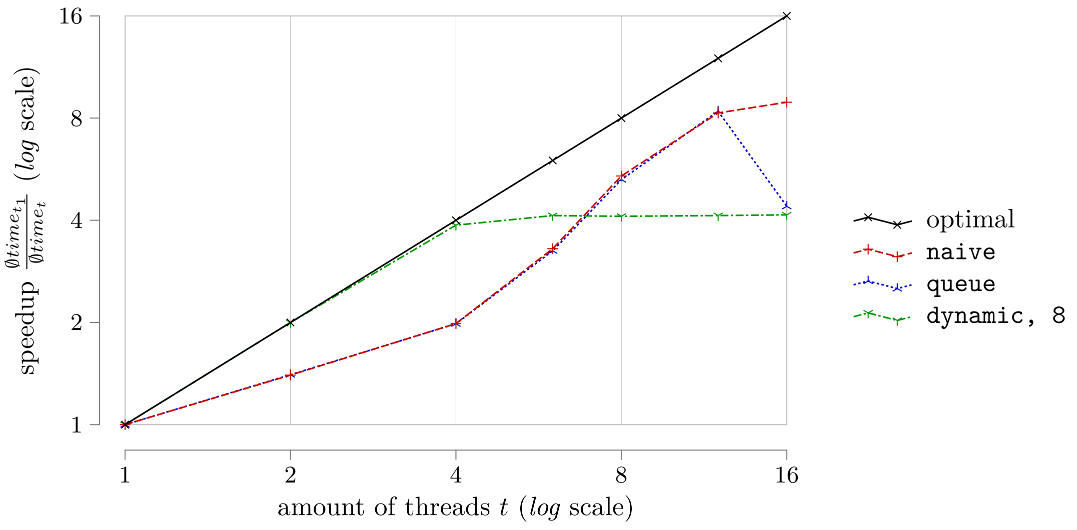
During execution of the second loop, Figure 2b shows that the affinity schedules do not scale even close to optimal speedup for 2, 4 and 6 threads. Their scaling increases rather rapidly from 4 to 12 threads while decreasing again for 16 threads. `dynamic, 8` scales close to optimal until 4 threads and hits a plateau afterwards, which is the reason why the affinity schedules outperform the built-in schedule for more threads.

4. Discussion

While the `queue` version of the affinity schedule has the theoretical better sequential performance than the `naive` affinity schedule, the fact that it is not thread-safe and therefore must be made thread-safe by mutual exclusion, clearly makes the priority queue a bottleneck. The synchronization really hurts the `queue` affinity schedule, making its performance drop with higher amounts of threads, wherefore the `naive` affinity schedule outperforms it.



(a) Loop 1.



(b) Loop 2.

Figure 2: Speedup when using more threads. The speedup was calculated over the average of every measurement, grouped by the amount of threads.

While for the first loop the performance difference between the built-in schedule and both versions of the affinity schedule is minimal, the affinity schedules—especially the `naive` affinity schedule—out-perform the `dynamic, 8` for the second loop with higher threads. This paper therefore underlines the claim made in (B160509, 2019), that a `dynamic` schedule with a lower chunk size than 8 would scale better for the second loop, because the tested affinity schedules provide a lower chunk size for the most parts.

Overall the usefulness of the affinity schedule is questionable, because more inner state must be maintained (the shared splits instead of just the loop) and more synchronization must be performed—once a thread has finished its split—than during a truly `dynamic` schedule.

5. Conclusion

The affinity schedule's performance is comparable to the built-in schedules tested in B160509 (2019), sometimes even outperforming them. But overall, the affinity schedule suffers from its complexity and the necessarily accompanying synchronization, because it tries to add the properties of several built-in schedules into one schedule (see Chapter 2).

If one were to continue looking into a better performing affinity schedule, a good place to start would probably be to build the `queue` affinity schedule on top of a thread-safe priority queue. This could remove its bottleneck and make it scale in a nice way, because its overhead would be compensated by a bigger amount of threads, where probably the `naive` affinity schedule would suffer performance drops.

References

Apple Inc. The Swift Programming Language, 2019. URL `https://swift.org/`.

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

Chad Austin. Sum Types Are Coming: What You Should Know, 2015. URL `https://chadaustin.me/2015/07/sum-types/`.

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

EPCC. Cirrus, 2019. URL `https://www.cirrus.ac.uk`.

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

The Rust Team. Rust Version 1.39.0, 2019. URL `https://www.rust-lang.org/`.