Threaded Programming Coursework

B066600

November 30, 2018

1 Introduction

The objective of this piece of work is to implement the affinity scheduling algorithm using the OpenMP framework. Assuming n iterations for a loop and p threads available, the affinity scheduling algorithm can be described as follows:

- Each thread is first assigned a local set of n/p iterations.
- Each thread independently executes a chunk of iterations whose size is the number of remaining iterations in its local set divided by p.
- Each thread continuously performs the step above until there are no more iterations left in its local set.
- Once an individual thread has finished all the iterations in its local set, it determines which of the other threads has the highest number of remaining iterations in its respective local set and it executes a chunk of iterations whose size is the remaining iterations in the local set of the most loaded thread divided by p.
- As each thread finishes the iterations in its local set, it repeats the step above until there are no more iterations remaining in any thread's local set.

2 Design and Implementation

As a starting point, two parallel loops are provided along with a verification test for each loop and a measurement of the execution for 1000 repetitions of each loop on 729 iterations. However, instead of using an explicit work sharing directive, the loops are scheduled "by hand" in such a way that the implementation is equivalent to the OpenMP STATIC schedule kind.

The basic data structures used for the implementation of the affinity algorithm proposed here are two arrays that store the number of remaining iterations and the last iteration performed by each thread. These arrays are shared as they are continuously updated by each thread throughout the execution of the algorithm. These arrays are also statically allocated to contain 16 elements, which satisfies the programme requirement of being able to be executed by 16 threads.

After having been assigned a local set of iterations, each thread calculates its very first chunk of iterations by calculating values for the inner lower and inner higher bounds. The general rule is that the inner lower bound is equal to the inner higher bound of the previous chunk, while the inner higher bound is equal to the inner lower bound that has just been calculated summed with the quotient of the number of remaining iterations present in the thread's local set and the total number of threads available. The only exception to this is that for the very first chunk of iterations to be executed by each thread, the inner lower bound is equal to the lower bound of the entire local set of each thread.

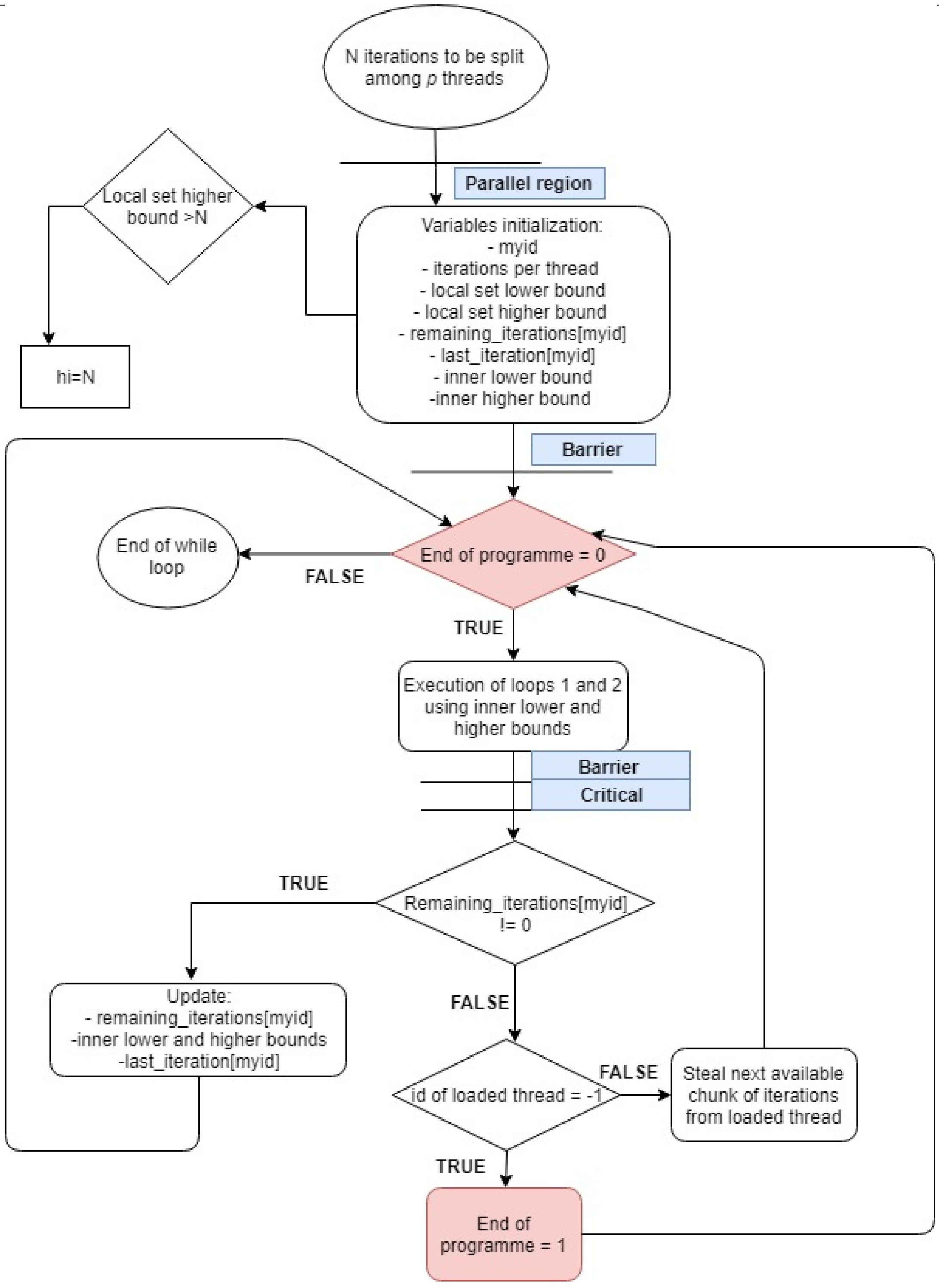
N iterations to be split among p threads
Parallel region
Local set higher bound >N
hi=N
Variables initialization:
- myid
- iterations per thread
- local set lower bound
- local set higher bound
- remaining_iterations[myid]
- last_iteration[myid]
- inner lower bound
-inner higher bound
Barrier
End of while loop
FALSE
End of programme = 0
TRUE
Execution of loops 1 and 2 using inner lower and higher bounds
Barrier
Critical
TRUE
Remaining_iterations[myid] != 0
Update:
- remaining_iterations[myid]
-inner lower and higher bounds
-last_iteration[myid]
FALSE
FALSE
id of loaded thread = -1
Steal next available chunk of iterations from loaded thread
TRUE
End of programme = 1

Figure 1: Control flow diagram for the proposed affinity algorithm implementation using the OpenMP framework.

After entering the parallel region, each thread passes the inner lower and higher bounds of their first chunk of their respective local sets as arguments to the *loop1chunk* and *loop2chunk* functions. In turn, these functions use these values as actual low and high loop bounds in order to calculate two independent values that are essentially counters for the number of iterations that are actually performed by loops 1 and 2. This is helpful as it allows for a straightforward evaluation of the correctness of the execution of the affinity algorithm on different number of threads. Moreover, this part of the algorithm is surrounded by a while loop (see Figure 1), which ensures that the programme executes until essentially all the threads have finished their respective local sets of iterations. Before entering the while loop, all the variable initializations and writing to shared arrays are synchronized by using a OpenMP barrier.

After each thread has entered the while loop and passed the inner lower and higher bounds to the *loop1chunk* and *loop2chunk* functions, threads are synchronized using an OpenMP barrier, such that there are no threads lagging behind. After the barrier, each thread in turn enters a critical region and faces two scenarios based on its number of remaining iterations left:

- If the number of remaining iterations is different from 0 (i.e the thread has to still carry out some iterations from its local set), the thread will then update its number of remaining iterations (by taking into account the inner lower and higher bounds of the chunk of iterations it has just passed to the *loop1chunk* and *loop2chunk* functions before entering the critical region), its last iteration (which is equal to the inner high bound it has just passed) and it will finally calculate the inner lower and higher bounds of the chunk it will execute next based on the general rule described above.
- If the number of remaining iterations is 0 (i.e the thread has finished all the iterations present in its local set), the thread will instead identify which is the most heavily "loaded" thread by calling a function named *largest*. This function was purposely built such that it returns the thread id after having traversed the shared array holding the values of the remaining iterations for all threads.

In the case in which a heavily loaded thread cannot be identified (i.e. all threads have finished their local set of iterations), the *largest* function will return -1, which will be interpreted as a flag to exit the while loop and terminating the program. However, if there is a thread that has some iterations left to process, the "free" thread will access those iterations, calculate a chunk of iterations based on the general rule outlined above and update the remaining iterations for the thread it is stealing from such that the "loaded" thread will not attempt to process these iterations once the "loaded" thread has subsequently entered the critical region.

3 Analysis of Performance Data

In order to gain an understanding of the performance of the newly developed affinity algorithm, the program was executed on 1, 2, 4, 6, 8, 12 and 16 threads on the back-end on Cirrus and its execution time was measured. In order to also benchmark the algorithm, its performance was compared against the best built-in OpenMP schedules determined in part 1 of the coursework, namely GUIDED with chunksize 4 and DYNAMIC with chunksize 16 for loop 1 and 2, respectively. Although the affinity algorithm produced the correct validation values for both loops 1 and 2 on 1, 2, 4 and 12 threads, it appeared to unexpectedly deadlock when it was run on 6, 8 and 16 threads. Due to time constraints, this bug could not be fixed and in order to make up for the lack of data points, both algorithms were additionally executed on 3, 9 and 10 threads. The execution times recorded for all these experiments are shown in Table 1. As it can be easily seen, the best built-in OpenMP schedules for either of the two loops always perform better than the affinity algorithm, regardless of the number of threads the programme is run on.

In Figure 1, the performance speedup achieved with increasing number of threads (hence due to increasing parallelization) is plotted for loops 1 and 2 when they are executed with the affinity algorithm or each respective best built-in OpenMP schedule clause. Consistent with the execution times reported in Table 1, the maximum speedup achieved on 12 threads by either loop when executed using the affinity algorithm is approximately 3 times smaller than the one achieved with the respective best built-in OpenMP schedule. This suggests that the stealing mechanism encoded in the affinity algorithm could potentially be introducing a consistent overhead to the total execution time.

Table 1: Execution time (sec) for loops 1 and 2 using affinity algorithm and best built-in OpenMP schedules based on analysis performed in part 1 of coursework.

	Loop 1		Loop 2	
Threads	Affinity algorithm	GUIDED 4	Affinity algorithm	DYNAMIC 16
1	1.69	1.68	8.64	8.66
2	1.28	0.85	7.18	4.40
3	0.97	0.57	6.68	2.98
4	0.79	0.45	6.43	2.20
9	0.48	0.26	5.86	2.07
10	0.45	0.21	5.60	2.07
12	0.48	0.17	5.48	2.07

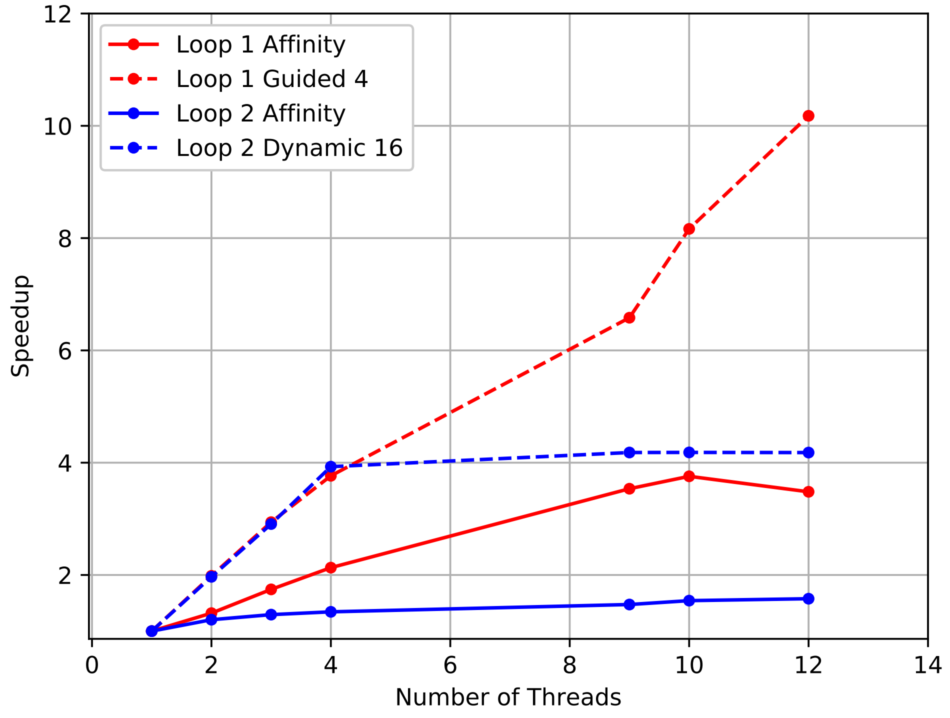

Figure 2: Performance speedup (calculated as T1/Tp, where p is the number of threads) for loops 1 and 2 when run on either affinity algorithm or best built-in OpenMP schedule on 1, 2, 3, 4, 9, 10 and 12 threads.

4 Conclusions

In this piece of work, an alternative algorithm for scheduling loop executions has been developed using the OpenMP framework. The performance of the algorithm has been evaluated against some built-in OpenMP loop scheduling clauses and the algorithm was found to consistently exhibit lower performance. Further tests and optimization of the work stealing control flow could lead to further performance improvements.