Threaded programming coursework 3

B177252

December 1, 2020

1 Introduction

This paper implements and discusses a new loop schedule algorithm which is affinity schedule. Section 2 includes the introduction and implementation of affinity schedule. Section 3 discusses the performance of affinity schedule and compares it with the optimal schedule discussed in previous coursework. The last section makes a conclusion briefly.

2 Affinity schedule and implementation

2.1 Affinity schedule

Affinity schedule is a flexible schedule algorithm. Initially, each thread is allocated a contiguous local set of iterations. For example, suppose there are n iterations in a loop and p threads, each thread has a local set with n/p iterations. Then each thread executes chunks of iterations in which the chunksize is 1/p of the iterations left in its local set until there is no more iterations in its local set. If a thread has completed all the iterations in its local set, it automatically locates the thread with most iterations and helps it execute iterations with a chunksize of 1/p of the remaining iterations in the local set of the thread that is being helped. In this process, many threads might compete with each other to 'help' the most-loaded thread. Hence there must be synchronisation to avoid race condition. Affinity schedule stops when there is no more remaining iterations in the local set of any thread.

2.2 Implementation

In this part, we discuss the detailed implementation of affinity schedule. In terms of the expression of local set, each thread has a local set consisting of n/p iterations. However, if n can not be divided by p, the last thread can not get n/p iterations. Therefore, in this case, we allocate iterations whose chunksize equals the ceiling integer of n/p to each thread and make sure the last thread has the remaining iterations as its local set. Then we come to the expression of remaining iterations in the local set. One easy way to think about is to record the index of the latest iteration and use the index of the last iteration of the local set to subtract it to get the number of remaining iterations. However, the problem is how to preserve the starting and ending index of remaining iterations for all the threads. We can allocate two arrays to store them. The core part of affinity schedule is how to allocate iterations and how to use synchronisation to avoid race condition.

Firstly, we initialize the starting and ending index of each thread to be the starting and ending index of its local set respectively. Then we define a function to update the iterations to be executed. In this function, we first judge whether the thread has finished the iterations of its local set. If not, it continues to execute the next iterations. Otherwise it needs to find the most-loaded thread first, then execute the next iterations of it and update the starting and ending index of its remaining iterations. To avoid race condition, we use critical section for this function, which means only one thread can execute the function at a time. However, this does not mean we have successfully prevented race condition. Actually, there still exists another potential race condition that happens in the initialization part of starting and ending index of the remaining iterations for each thread. The solution is adding a barrier at the end of the initialization part. Finally, when all the iterations have been executed, the main program stops. Pseudo code is presented in algorithm 1, algorithm 2 and algorithm 3.

Algorithm 1 find_most_overloaded_thread

```
Input: remaining_start;remaining_end;nthreads
index ← -1;
maxidle ← 0;
for i← 1 to nthreads do
    remaining ← remaining_end[i]-remaining_start[i]
    if remaining>maxidle then
        index ← i
        maxidle ← remaining
return index
```

Algorithm 2 get_next_iterations

```
Input: remaining_start;remaining_end;nthreads;myid;low;high
index ← 0; maxidle ← 0
if remaining_end[myid] > remaining_start[myid] then
    index ← myid
else
    index ← find_most_overloaded_thread(remaining_start,remaining_end,nthreads)
    low← remaining_start[index];high←remaining_start[index]+chunksize
    remaining_start[index] ←remaining_start[index]+chunksize
if index=-1 then return -1
remaining← remaining_end[index]-remaining_start[index]
chunksize ← (int) ceil(remaining * 1/nthreads)
return 0
```

Algorithm 3 Affinity schedule

```
Given: loopid;N;
Initialize: nthreads; remaining_start[nthreads];remaining_end[nthreads];
#pragma omp parallel default(none) shared(loopid,nthreads,remaining_start,
remaining_end)
{
myid ← omp_get_thread_num().
ipt ← (int) ceil(N/nthreads).
lo ← myid*ipt; hi ← min ((myid+1)*ipt , N).
low ← 0; high ← 0.
remaining_start[myid] ← lo; remaining_end[myid] ← hi;
flag ← 0
#pragma omp barrier
while flag!=-1 do
    if loopid=1 then
        loop1chunk(low,high)
    else
        loop2chunk(low,high)
    #pragma omp critical
    {
    flag ← get_next_iterations(remaining_start,remaining_end,nthreads,myid,low,high);
    }
}
```

3 Experiment and discussion

In this section, we perform several experiments to test the performance of affinity schedule and compare the results with the optimal schedule that we chose in coursework 2.

3.1 Experimental environment

Experimental environment is exactly the same as that in coursework 2.

- Intel-Compilers-19
- Linux 4.18.0-147.8.1.el8_1.x86_64
- Bash 4.4
- OpenMP 4.5

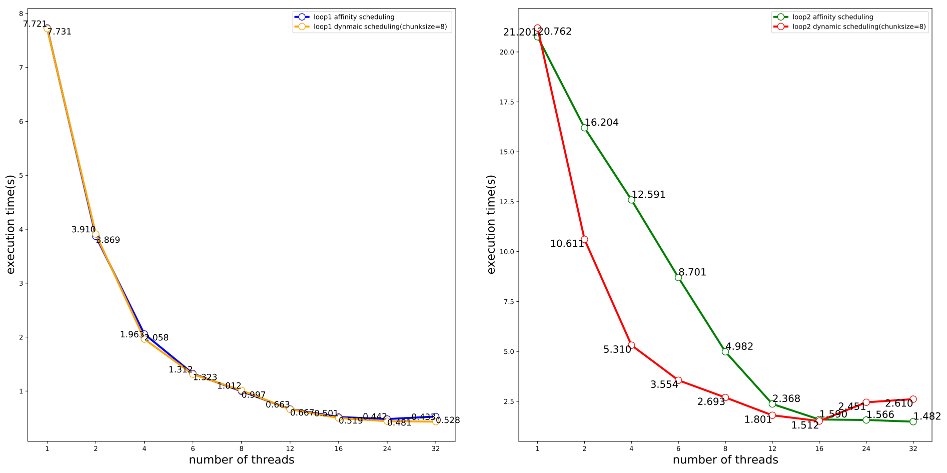
- Python 3.8.3
- Matplotlib 3.2.2

3.2 Details of experiment

In order to test how varying number of threads affect the performance of affinity schedule and how it compares with the optimal schedule that we selected in coursework 2, we run the code on 1, 2, 4, 6, 8, 12, 16, 24, 32 and record the execution time respectively. In order to make accurate timing, we use compute nodes rather than public nodes to run the code by submitting a SLURM script that includes a command "export OMP_NUM_THREADs=N", in which N is to be changed. To make timing more accurate, we record the execution time for 10 times and use the average time to be the final result for each case. Afterwards, it comes to the part of comparison between affinity schedule and the optimal schedule which was determined to be dynamic schedule with chunksize 8 for both loop1 and loop2 in coursework 2.

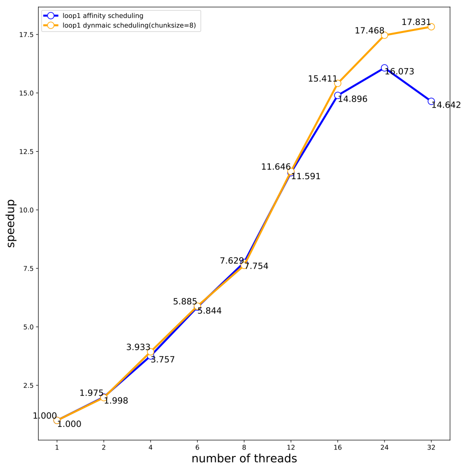
3.3 Results

Comparison of execution time of loop1 and loop2 is presented in figure 1 while comparison of speedup is presented in figure 2. Table 1 and 2 in appendix A present specific data.

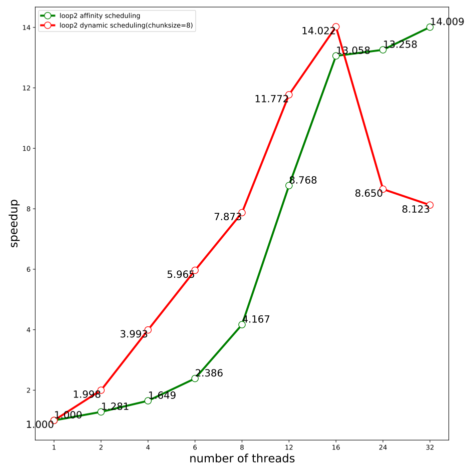


(a) Time of loop1 versus number of threads (b) Time of loop2 versus number of threads

Figure 1: Execution time of loop1 and loop2



(a) Speedup of loop1 versus number of threads



(b) Speedup of loop2 versus number of threads

Figure 2: Speedup of loop1 and loop2

3.4 Description

As is shown in the two figures, for loop1, execution time of the two schedules is very close for all the number of threads, which makes it hard to choose the winner. However, when it comes to the speedup, we can see clearly that the turning point of affinity schedule appears at 24 threads, when the execution time is 0.481 seconds, while for dynamic schedule, the performance keeps increasing within 32 threads. Generally, dynamic schedule is slightly better than affinity schedule with few exceptions in loop1 For loop2, the situation is different. When number of threads is smaller than 24, dynamic schedule performs better than affinity schedule. However, affinity schedule surpasses dynamic schedule when there are more than 24 threads. What's more, as for speedup, the turning point of dynamic schedule appears when number of threads is 16 while the performance of affinity schedule keeps improving within 32 threads.

3.5 Explanation

This part is the explanation of the experimental results. Firstly, we recap dynamic schedule and affinity schedule briefly. As for dynamic schedule with chunksize being 8, the iterations are divided into many chunks in which each chunk has 8 iterations. Then the chunks are assigned to all the threads in a first-come-first-served order. As for affinity schedule, Suppose there are p threads and n iterations, each thread is assigned n/p iterations as its local set initially then executes 1/p of its remaining iterations each time until its local set is finished. If any thread finishes its local set, it continues to help the most loaded threads. The whole schedule ends when all the iterations are finished.

Firstly, we come to loop1. As we have discussed in coursework 2, loop1 is not load-balanced because the number of iterations in the inner loop decreases in a subsequent order. Dynamic schedule basically makes sure that each thread has something to do at any time in this case, which makes it an efficient schedule in loop1. For affinity schedule, although each thread has a large local set, which is similar to static schedule, threads can also cooperate efficiently by all the vacant threads trying to help the most loaded thread. That's the reason why affinity schedule is also a good schedule in this case. However, to avoid race condition between threads, synchronization is necessary in affinity schedule so that some threads have to wait when a thread is executing the synchronized code, which lowers the efficiency potentially. This is the reason why dynamic schedule is slightly better than affinity schedule in loop1. When there are too many threads, the time for waiting also increase due to the synchronization part. This is the reason why turning point appears when number of threads is 24. In terms of loop2, the number of iterations in its inner loop is either 1 or 864, which is extremely load-imbalanced as we discussed in coursework 2. For dynamic schedule, it performs well due to its flexibility. However, using more threads in dynamic schedule does not guarantee better performance because creating threads and assigning work to them also takes time and more threads consume more CPU, which lowers the efficiency. For affinity schedule, if number of threads is small, each thread has a large local set as the iterations are approximately evenly distributed. As a result, even though affinity schedule can dynamically assign vacant threads to help the most loaded threads, the workload is still too heavy and help is not enough. This is the reason why affinity schedule performs worse than dynamic schedule when number of threads is smaller than 24. However, affinity schedule is always catching up with dynamic schedule as number of threads increases and is close to it when there are 16 threads. With more than 16 threads, affinity schedule successfully surpasses dynamic schedule. Although affinity schedule is not efficient with only few threads, it shows its advantage as number of threads increase, which means more threads can get involved in helping the most loaded thread. For load-imbalanced loops like loop2, the performance of schedule is normally determined by how it deals with the most-loaded threads because these threads take most time to finish their work. Compared with dynamic schedule, although it adopts first-come-first-served strategy to dynamically assign work to threads, it also means that each thread must finish its work on its own and no other threads would come to its help even if it is allocated the heaviest work. This explains why affinity schedule performs better with more than 16 threads. However, it does not mean performance of affinity schedule would always improve with more threads. Actually, too many threads would necessarily lower the efficiency for any schedule, no exception for affinity schedule. Therefore, turning point does exist at a specific threads number for affinity schedule.

We might wonder why in loop1, affinity does not show any advantage over dynamic one. That is because number of the inner iterations decreases subsequently, which means loop1 is mildly load-imbalanced. In contrast, loop2 is extremely load-imbalanced and affinity schedule is particularly useful in solving this kind of loops due to its 'help most-loaded thread' strategy.

4 Conclusions

In conclusion, affinity schedule is a very flexible schedule which dynamically assigns vacant threads to help the most-loaded thread. The implementation requires synchronization to avoid race condition. Affinity schedule is a very useful schedule in solving extremely load-imbalanced loops due to its strategy of allocating all vacant threads to help the most-loaded thread. In contrast, dynamic schedule is good at solving mild load-imbalanced loops. Moreover, turning point of speedup exists for any schedule. When using too many threads, the speedup would necessarily decrease.

A Tables

Execution time(s) of loop1									
Schedule	1	2	4	6	8	12	16	24	32
affinity	7.731	3.869	2.058	1.323	0.997	0.667	0.519	0.481	0.528
(dynamic,8)	7.721	3.910	1.963	1.312	1.012	0.663	0.501	0.442	0.433

Table 1: Execution time of loop1

Execution time(s) of loop2									
Schedule	1	2	4	6	8	12	16	24	32
affinity	20.762	16.204	12.591	8.701	4.982	2.368	1.590	1.566	1.482
(dynamic,8)	21.201	10.611	5.310	3.554	2.693	1.801	1.512	2.451	2.61

Table 2: Execution time of loop2