|epcc|

Threaded Programming coursework2

B156924

November 29, 2019

Contents

1 Introduction

Affinity scheduling is an optimized algorithm that can balance the workload between threads to achieve high efficiency in execution. This report will take an original two-loop program with static scheduling as an example, illustrate the methods of developing the affinity scheduling, and then analyze the execution results compared with the best scheduling obtained in the last coursework. In this way, we could evaluate the impact of affinity scheduling on performance improvement.

2 Program design

In this program, the affinity scheduling means that when the program is allocated with P threads, each thread executes 1/P of its remaining iterations every time. When some thread completes all its work, it can steal a chunk of 1/P remaining iterations of the most loaded thread. When developing this algorithm, we need to consider the variables which should be stored in the shared memory and ensure that all threads perform their work synchronously. In this chapter, this report will illustrate the methods of developing affinity scheduling in detail.

2.1 Shared variables

At the beginning of designing this algorithm, shared variables and their structures are worth noting. To target the most loaded thread, each thread should know other threads' on-the-fly work. Therefore, the shared variable should include the start and end iteration of each thread, the start, end iteration and the size of the chunk being executed in each thread. Besides, the remaining iterations in each thread should also be shared so that the idle thread could steal work from them. After full consideration, 8 shared variables are designed. Among them, 6 variables are array type: `int remain_array[p]`, `int thread_start[p]`, `int thread_finish[p]`, `int chunk_start[p]`, `int chunk_finish[p]` and `int chunk[p]`. Another two variables are `int p` - the number of threads and `int each_sum` - all initial iterations assigned for each thread.

As for each array, the size p is equal to the number of threads and the position of each element in the array corresponds to the rank of the thread. In this way, it is convenient for threads to share their work information.

2.2 Work initialization

After declaring these shared variables, we could move to the parallel region. For every thread, the first thing we need to do is to know its rank. In each thread, the thread rank is

called `myid`. Then, we can start initializing each array. Here are the steps for threads' work initialization:

First, each thread's start and finish iteration depends on its rank. The workload for every thread should be the same as the value of `each_sum`, however, it may happen that the total number of iteration for a loop cannot be divided equally by the number of threads. Therefore, the last thread may have relatively less workload. By calculating the actual work, the `remain_array[myid]` can be initialized.

Second, we need to initialize the first chunk for every thread. As required, the first chunk is 1/p of the remaining iterations. Correspondingly, the `remain_array[myid]` should drop this chunk. The start iteration of this chunk equals to that of the thread, and the end iteration equals to the start iteration plus the chunk size.

In this way, the work initialization is completed.

2.3 Work assignment process

In this part, we need to consider how to move chunks and steal work from other threads if possible. Before we start to design the code, we should set a private flag - `int complete` as the loop condition to judge whether the thread has finished all its workload. If the flag equals 0, it means that the thread is performing its initial allocated work or the work stolen from other threads. When the flag turns 1, it means that this thread has finished all its work and there are no other existing remaining iterations.

When there are still iterations remained in the thread, the thread will focus on executing its chunk, whose size is 1/p of the remaining iterations. In this case, the next chunk's start iteration should be the start iteration of the last chunk plus the last chunk size. Then it will be dropped from the remaining iterations array. It is worth noting that when the thread executes the last chunk, the chunk size may be bigger than the remaining iterations because of the chunk size calculation based on the round-up principle. Therefore, the end of the chunk should be either the end iteration of the thread or the number which is smaller than the thread end.

If the start of the chunk is equal to the end of the chunk, it means that this thread has finished all work allocated. Then, this thread is available to steal work from the thread that has the heaviest workload. Figure 1 illustrates the principle of work-stealing when three threads are allocated for this program. As we can see from the graph, when thread 2 completes its initially assigned work, it steals 1/3 iterations from thread 0's remaining work and executes 1/3 of the remaining gained iterations every time. As for thread 0, when thread 2 is stealing its remaining work, it is executing its current chunk without interruption. When the chunk has been done, it will automatically skip the stolen iterations and execute the 1/3 of the rest work. In this way, all threads can work synchronously. The implementation of this algorithm can be seen in algorithm 1.

To explain this process more clearly, a flow chart(Figure 2) is drawn. As we can see from the chart, when a thread completes all its work, the flag will turn to 1 and break

the loop. When all threads become idle, the program is completed.

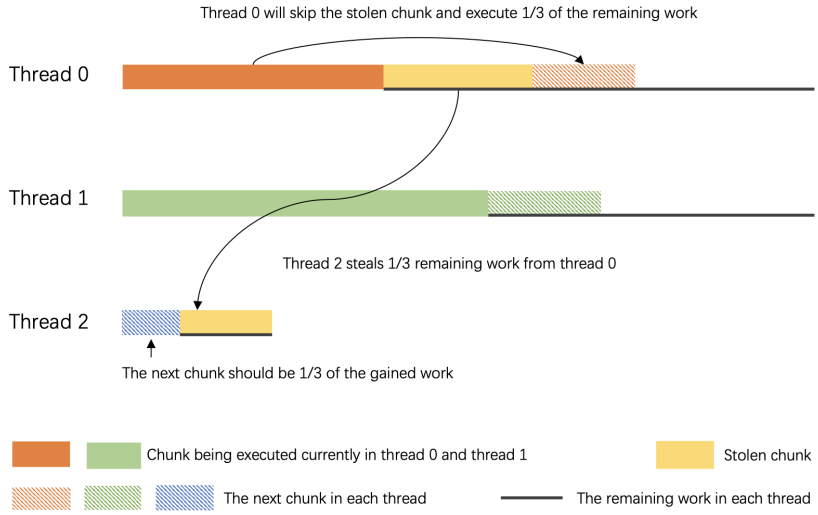

Figure 1: This steal work process takes 3 threads (p = 3) condition as an example. In this case, thread 2 has completed its initially allocated workload and it can steal work from other threads. For thread 0 and thread 1, it is obvious that thread 0 has the most remaining work. Therefore, thread 2 will steal 1/3 iterations from thread 0's remaining work. For thread 0, its next chunk's start iteration will be the end iteration of the stolen chunk. For thread 2, its next chunk will be 1/3 of the gained chunk.

2.4 Parallel region directives

During execution, we need to add several directives to ensure that all threads will synchronously execute their work and the shared variables will not be modified by more than 1 thread at a time.

Before executing chunks, it is essential that every thread has finished its work initialization. Otherwise, it may happen that one thread has completed its initial work and start to steal work, while another thread has not finished initialization, which may cause a segmentation fault. Therefore, we need to set a barrier (#pragma omp barrier) after the work initialization and before the while loop to ensure that every thread' execution has reached the while loop code.

Another issue is that when the shared variables are changed by one thread, other threads should not access to them. In this case, the directive #pragma omp critical is before the work assignment process. In this way, when one thread is executing the code in the critical area, others should wait until it finishes all of the operations.

Algorithm 1 Steal work algorithm

1: **procedure** STEAL_WORK($pointers\_of\_all\_shared\_variables, myid, p$) ▷ myid is the rank of the thread that excutes these code, p is the number of threads
2: $id \leftarrow 0$ ▷ initialize id as the loop variable
3: $max\_remain \leftarrow 0$ ▷ the most loaded thread's remaining work
4: $find \leftarrow myid$ ▷ initialize the most loaded thread's rank
5: $complete \leftarrow 1$ ▷ a flag to judge whether there are still some thread with remaining iterations
6: $steal\_chunk$ ▷ create this variable to store the size of the steal chunk
7: **for** $id = 1, 2, \ldots, p-1$ **do** ▷ look for the most loaded thread
8: **if** $remain\_array[id] > max\_remain$ **then**
9: $max\_remain \leftarrow remain\_array[id]$
10: $find \leftarrow id$
11: **if** $find \neq myid$ **then** ▷ the most loaded thread is found
12: $complete \leftarrow 0$
13: $steal\_chunk \leftarrow 1/p \times max\_remain$
14: $thread\_start[myid] \leftarrow chunk\_start[find] + chunk[find]$
15: $thread\_finish[myid] \leftarrow thread\_start[myid] + steal\_chunk$
16: $remain\_array[find] \leftarrow remain\_array[find] - steal_chunk$
17: $chunk[find] \leftarrow chunk[find] + steal\_chunk$
18: $chunk[myid] \leftarrow 1/p \times steal\_chunk$
19: $chunk\_start[myid] \leftarrow thread\_start[myid]$
20: $remain\_array[myid] \leftarrow remain\_array[myid] + steal\_chunk - chunk[myid]$
21: **if** $chunk\_start[myid] + chunk[myid]) > thread\_finish[myid]$ **then** $chunk\_finish[myid] \leftarrow thread\_finish[myid]$
22: **else**$chunk\_finish[myid] \leftarrow chunk\_start[myid] + chunk[myid]$
23: **return** $complete$

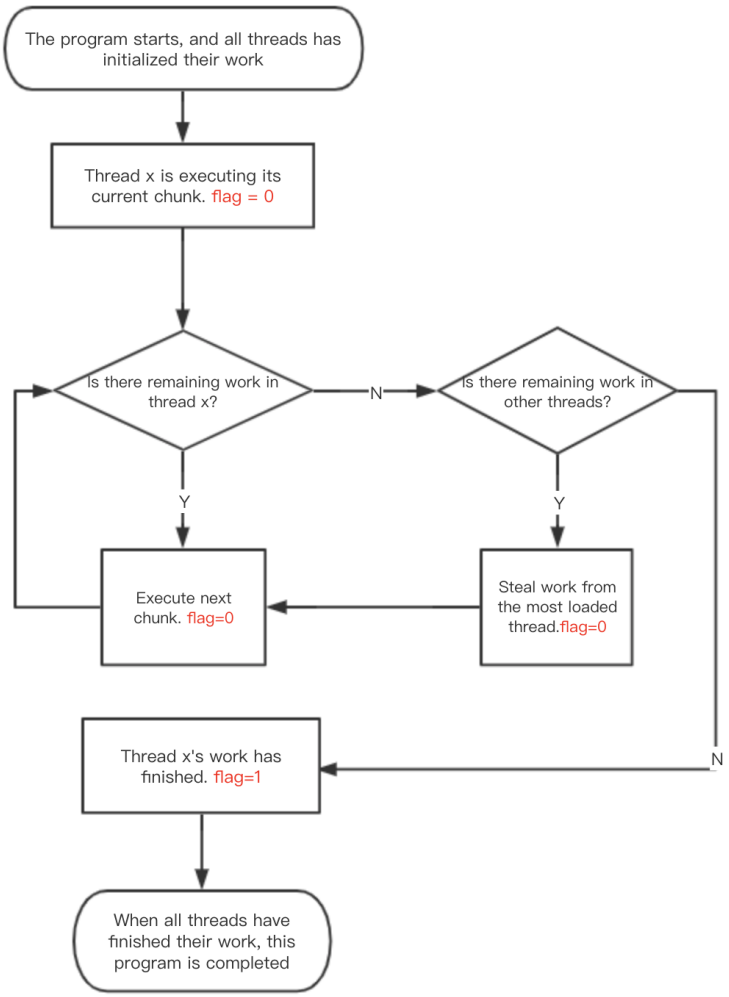

Figure 2: This flow chart portraits the workflow of a thread.

3 Results and analysis

After completing the program design, it is necessary to test the results of this program to judge how much performance improvement is gained from affinity scheduling. In this experiment, the execution time of loop1 and loop2 with 1, 2, 4, 6, 8, 12 and 16 threads is recorded respectively. Then, we need to compare these results with that of the best scheduling mentioned in the last coursework. Therefore, we could find out which loop can gain more advantages from the affinity scheduling and which can not.

3.1 Loop1 results analysis

From Figure 3 we can see that guided,2 schedule and affinity schedule have a similar effect on the execution time as the thread sum grows. When less than 4 threads are

allocated for loop1, the execution time for both schedules is almost the same. Then, with the thread counts increasing, the run time of the affinity schedule drops slightly and then rises gradually. The speedup of two schedules can be seen in Figure 4. It is obvious that the guided,2 schedule's speedup grows steadily with the number of threads increasing. On the contrast, affinity schedule drops to a lower value after reaching to the peak(about 7.5).

To explain these results, the loop structure and workload assignments in both schedules are worth noting. By analyzing the loop1 structure, we could know that it is a top-heavy loop. The internal iterations gradually decrease as the loop variable i increases. Its structure influences the performance of the schedules. The chunks in two schedules are similar since they all start large and get smaller. Besides, both schedules allow idle threads to steal work from busy threads. These are the reasons why they show similar trends when less than 8 threads are allocated. The difference between the two schedules is that the guided,2 schedule' chunks have been divided before entering the parallel region, however, the affinity schedule's chunk allocation heavily depends on the efficiency of different threads, and the chunk size needs to be calculated while one thread is stealing it, which results in higher overheads. When 16 threads share the whole work, the effect of overheads becomes more significant. When the loop has a small number of iterations to be executed, the workload in each thread is very small, but the affinity schedule allows them to steal work frequently. Therefore, more execution time is spent on reallocating chunks. As a result, the total run time becomes longer.

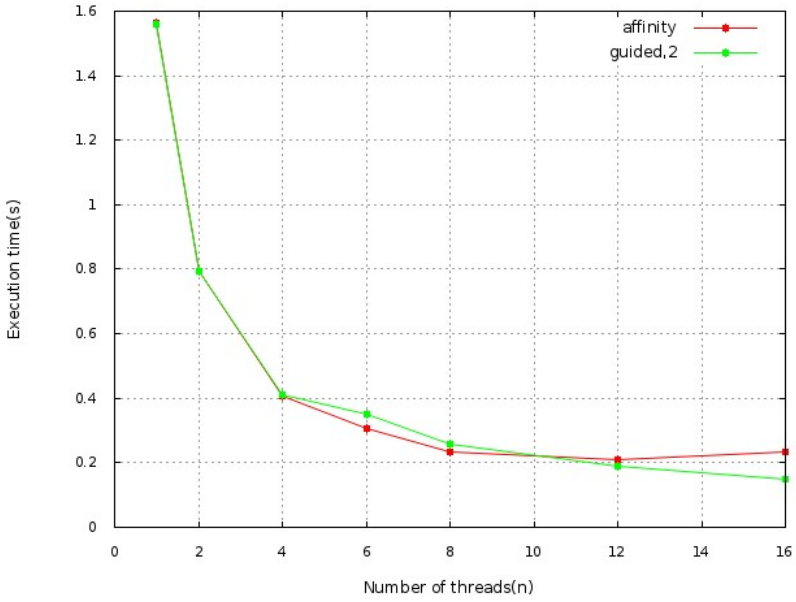

Figure 3: The execution time against the number of threads for loop1 with affinity schedule and guided,2 schedule.

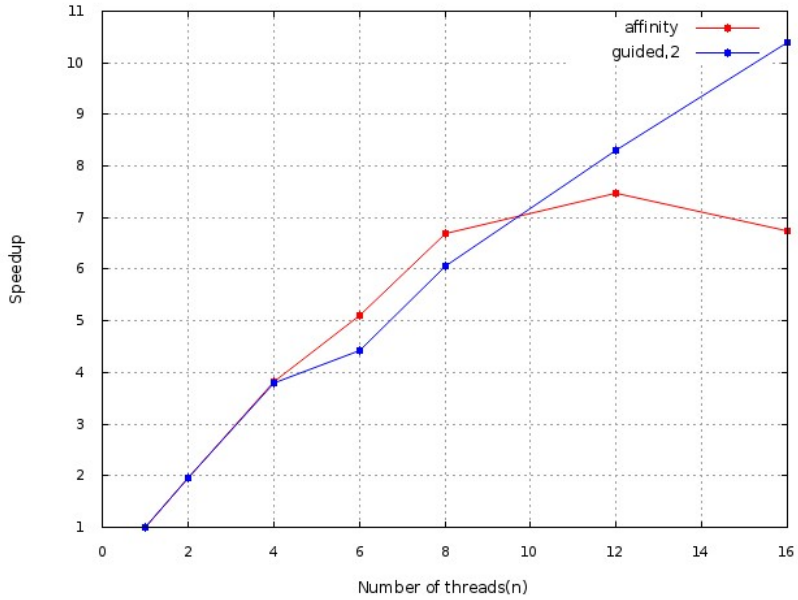

Figure 4: The speedup against the number of threads for loop1 with affinity schedule and guided,2 schedule.

3.2 Loop2 results analysis

Affinity schedule shows better performance for loop2. Figure 5 and Figure 6 illustrate the trends of the execution time and the speedup against the number of threads under affinity schedule and dynamic,16 schedule respectively. In Figure 5, the run time of the affinity schedule decreases steadily when more threads are provided. Whereas the run time of dynamic,16 schedule quickly drops to about 2s and then remains constant. From Figure 6, it is obvious to see that the speedup of the dynamic,16 schedule does not increase any more when more than 6 threads are allocated.

Loop2 is also an irregular loop. The number of the loop's inner iterations jumps back and forth between a large value and 0. When the loop variable i is small, there are many inner iterations with a big size, and with i grows, the frequency decreases gradually. As for dynamic,16 schedule, it divides the outer iterations equally with the chunk size 16. In this case, the idle thread can only execute the next chunk instead of stealing the work remained in the busy thread. When there are over 6 threads allocated, it happens that the busy threads are still working when other threads have completed their work and keep waiting because there is no chunk left. Consequently, the run time cannot be improved. As for the affinity schedule, it can ensure a good workload balance for loop2. When the thread with a small workload completes its work, it can steal work from the remaining work of the busy thread. In this way, the run time drops when the number of threads increases.

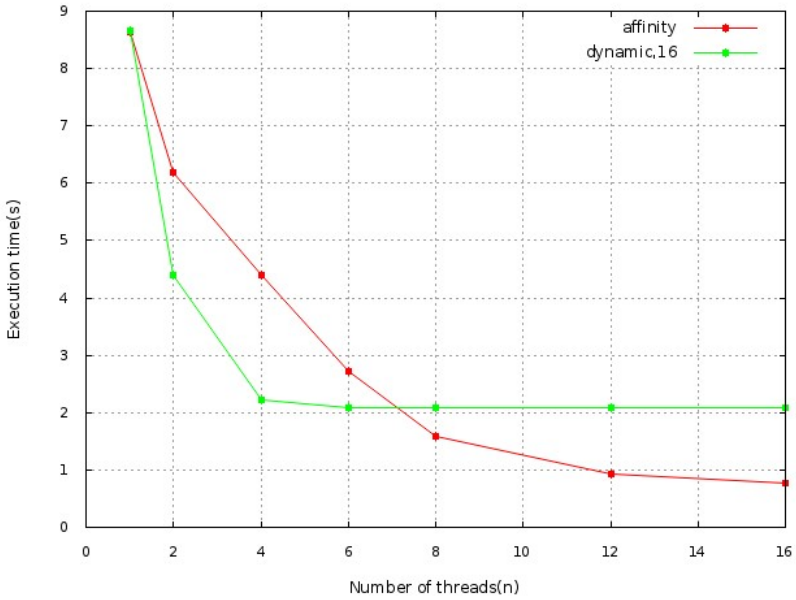

Figure 5: The execution time against the number of threads for loop2 with affinity schedule and dynamic,16 schedule.

4 Conclusions

In conclusion, the affinity scheduling is more suitable for loop2 with a discontinuous inner iterations. However, for loop1, it can obtain more advantages from guided,2 schedule. It is obvious that affinity scheduling has a limitation regarding different structures of loops. It can only promise a good load balancing when the gained chunk from other threads takes more execution time than the overheads from the stealing process.

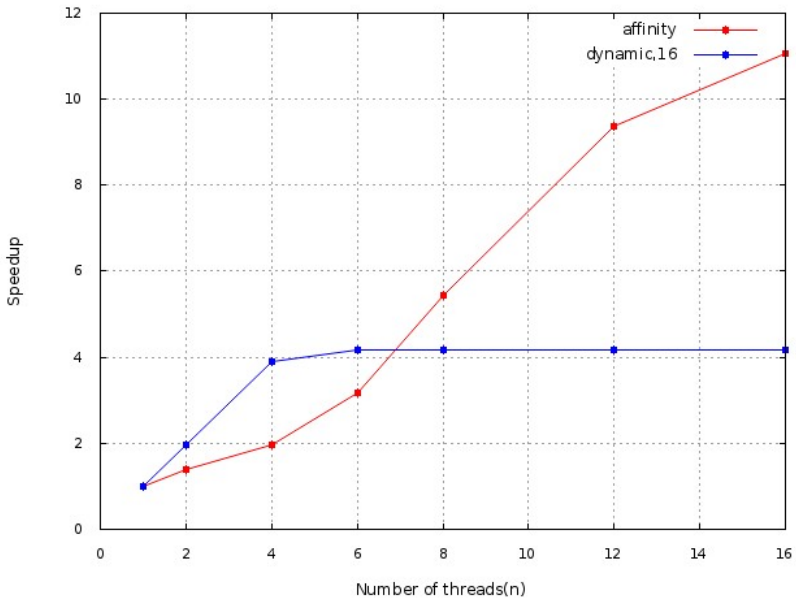

Figure 6: The speedup against the number of threads for loop2 with affinity schedule and dynamic,16 schedule.