MSc in High-Performance Computing

Threaded Programming

Coursework Part1

B142302

Introduction

Source Code Overview

There are two functions provided named ***loop1( )*** and another named ***loop2( )*** provided in the source code *loops.c:*

loop1( ) has two nested for-loops. The outer one has *N* iterations in which *N* is a constant. The inner loop has its range decided by the outer loop. Every iteration reads and writes values in different addresses without conflicts.

loop2( ) has three nested for-loops. The outermost one has *N* iterations in which *N* is a constant same as ***loop1( )***. The for-loop in the middle has its iteration limit decided by the iteration number of the outmost one. As the outmost loop's iteration increases by one, the middle loop fetches its new range limit from the next address in a certain list. The innermost loop has its range decided by the middle loop. Every iteration reads values in different addresses without conflicts and adds the calculation result to a floating point number. This floating point number preserved the sum of those results from every iteration.

Experiment Environment

The C version of the program source code is used for testing. Programs are compiled by the Intel-20.4 compiler with *-O3* level optimisation and *-qopemp* flag to enable OpenMP as required. The executable files are submitted to Cirrus and done by Cirrus.

Methodology

OpenMP Directives for For-loop Parallelisation

The two nested for-loops in ***loop1( )*** have a dependency. The range of the nested for-loop increased by 1 when the number of iterations of the outer loop increased by 1. We parallelise the outermost loop only.

Then we check the use of variables in the loops. There is no read or write operation at the same address between any of the two iterations. The write-to variable is defined in the global, and the iteration indexes are privately defined within the for-loop. Therefore, we don't need to declare any shared or private variables. We use ***schedule( )*** clause for for-loop parallelism scheduling. The directives are written above the outermost for-loop, e.g. using clause option **STATIC** with **chunk-size** = **1**:

```
#pragma omp parallel for schedule(static,1)
```

The three nested for-loops in ***loop2( )*** are parallelised in the same way as ***loop1( )***. However, in ***loop2( )***, we are writing to the same global variable ***zz*** and ***zz*** sums up all the values those threads outputted. Here we use a ***reduction( )*** clause with clause option ***'+'***. This allows the value of ***zz,*** privately owned by every thread, can be summed up together at the end of the parallelism. The directives are written above the outermost for-loop, e.g. using clause option **STATIC** with **chunk-size** = **1**:

```
#pragma omp parallel for schedule(static,1) reduction(+:zz)
```

Using Slurm Submit to Cirrus

The jobs are all submitted to Cirrus and tested remotely. The job submissions are handled by Slurm Workload Manager. A script loops.slurm is created.

We add ***—exclusive*** tag preventing from sharing nodes with other running jobs for variable control. The ***—nodes*** tag is set to 1 to limit the minimum as well as the maximum number of nodes for the jobs for variable control.

The number of threads utilised by OpenMP is set by ***OMP_NUM_THREADS.***

Evaluation

Loop 1 with All Schedule Options

(Serial execution time for 1000 reps of loop 1 = 10.973513 seconds)

thread=8	No size	n=1	n=2	n=4	n=8	n=16	n=32	n=64
STATIC	3.151	1.727	1.631	1.635	1.660	1.661	1.769	1.832
DYNAMIC		1.729	1.633	1.594	1.606	1.638	1.728	1.852
GUIDED		1.681	1.571	1.582	1.605	1.635	1.684	1.843
Auto	1.711							

Table1. Loops1 execution time(sec) versus the chunk size for the STATIC, DYNAMIC, GUIDED and AUTO schedules.

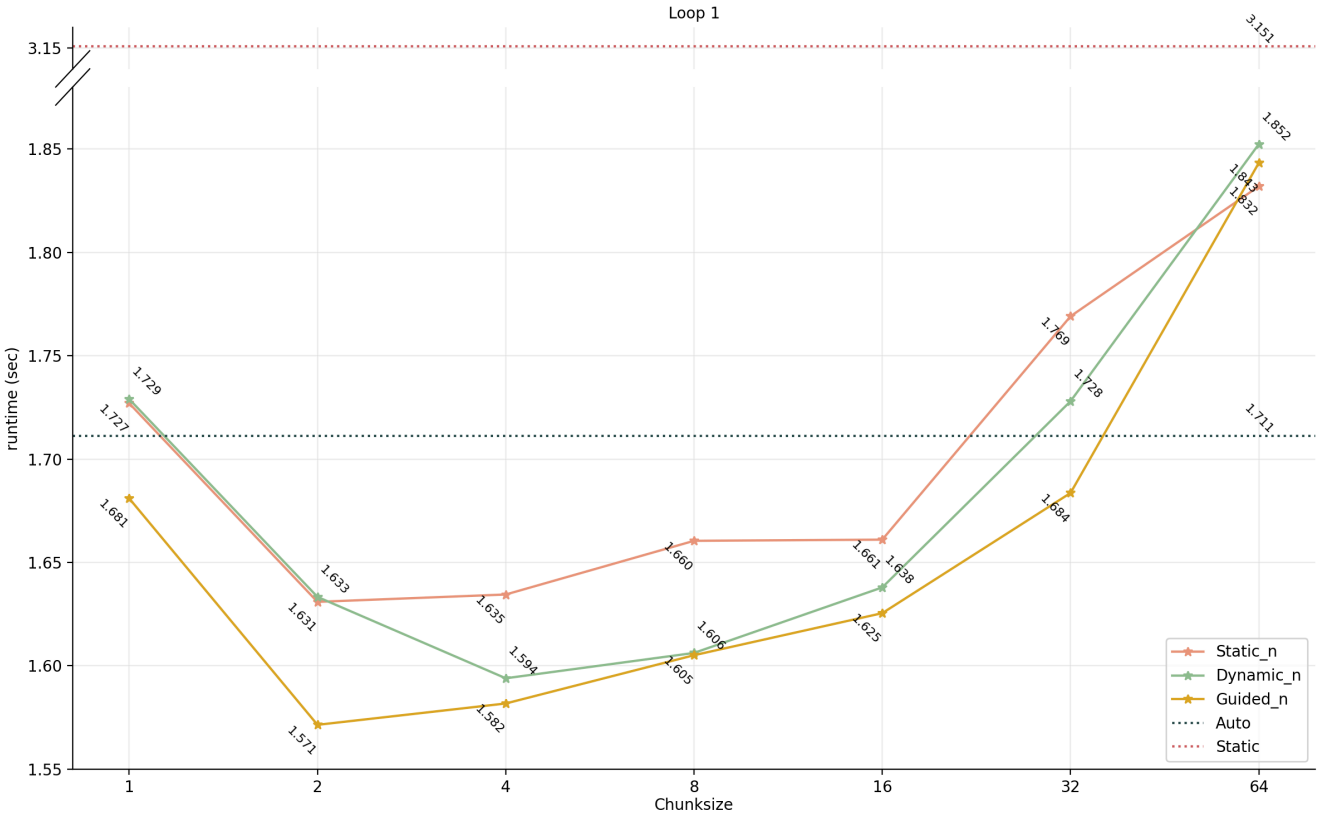

Table1. Loops1 execution time(sec) versus the chunk size for the STATIC, DYNAMIC, GUIDED and AUTO schedules.

STATIC option with default chunksize gives the worst performance. **GUIDED** option with chunksize=2 gives the best performance.

To parallelise ***loop1( )***, **DYNAMIC_N** works slightly better than **STATIC_N** on average. We can deduce that loops1( ) has a not large variance in the problem size of the chunks in every iteration. However, there are still differences in workload for each thread, and every thread has a slightly different speed. If the next chunk of size n tasks can be allocated to the thread first finished its execution, there should be a less blocking waiting time. **DYNAMIC_N** scheduling allows the next chunk to be distributed to the threads anytime they are available. Therefore, **STATIC_N** scheduling can not balance the workloads of each thread compared to **DYNAMIC_N** scheduling. For **STATIC** scheduling, the optimal chunksize is between 4 to 8. For **DYNAMIC** scheduling, the optimum chunksize locates between 8 to 16, evaluated by linear regression.

However, **DYNAMIC_N** works worse than **GUIDED_N** on average. For **GUIDED_N**, the master thread schedules the chunk size starts off large and decreases. We can deduce that there are many small chunks, and **DYNAMIC_N** puts much effort into scheduling those chunks frequently. **GUIDED_N** has fewer total dynamic chunks than **DYNAMIC_N**. The allocation time negates the advantages of **DYNAMIC_N**. The optimum performance for **GUIDED** scheduling is about chunksize=2.

The **AUTO** option shows no significant advantage on ***loop1( )***. **AUTO** scheduling gives complete control to the compiler. The **STATIC** option has a default chunksize equal to problem_size/ total_threads. The iteration space is divided into chunks that are approximately equal in size, and one chunk is distributed to each thread. The work is finally found not enough balanced distributed compared to other scheduling strategies.

Loop 2 with All Schedule Options

(Serial execution time for 1000 reps of loop 2 = 12.701866 seconds)

thread=8 LOOP2	No size	n=1	n=2	n=4	n=8	n=16	n=32	n=64
STATIC	11.219	4.480018	4.911169	4.320773	4.080170	3.985681	4.805430	8.063697
DYNAMIC		3.420091	3.231933	3.259350	3.184029	3.220742	3.229973	6.327306
GUIDED		10.148814	9.508035	9.354773	9.254869	9.584187	9.646040	9.653305
Auto	10.509776							

Table1. Loops1 execution time(sec) versus the chunk size for the STATIC, DYNAMIC, GUIDED and AUTO schedules.

STATIC option with default chunksize still gives the worst performance. **DYNAMIC** option with chunksize=8 gives the best performance.

To parallelise ***loop2( )***, **DYNAMIC_N** works better than **STATIC_N** on average. Changing the chunksize in the range of 1 to 32 does not affect their performance too much. Even out the distribution of work among the threads by the number of tasks makes the execution time of every thread uneven. If the next chunk of size n tasks can be allocated to the thread first finished its execution, there should be a less blocking waiting time. **DYNAMIC_N** scheduling allows the next chunk to be distributed to the threads anytime they are available. Therefore, **DYNAMIC_N** scheduling balances the overall workflow of each thread better than **STATIC_N**.

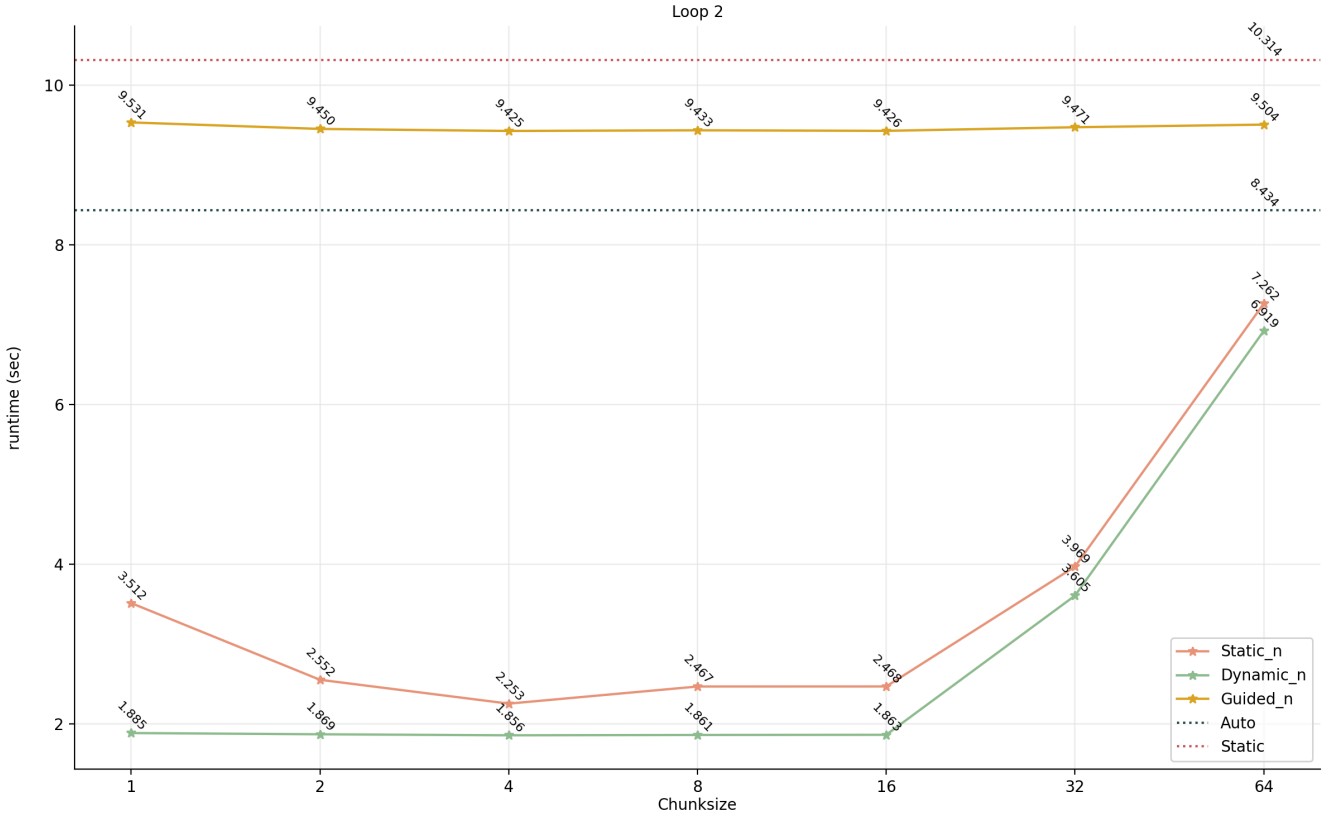

Table1. Loops2 execution time(sec) versus the chunk size for the STATIC, DYNAMIC, GUIDED and AUTO schedules.

However, **STATIC_N** works better than **GUIDED_N** this time. For **GUIDED_N**, the master thread schedules the chunk size starts off large and decreases. We can deduce that loops2( ) has a large variance in the tasks' size, and there are not many small chunks. Even though **GUIDED_N** has fewer total dynamic chunks and does not need to schedule those chunks more frequently than **DYNAMIC_N**, the load balance in the first few iterations of **GUIDED_N** scheduling is not optimistic. The computation overheads of each chunksize of **GUIDED_N** scheduling are also considerable. Those big chunks determine the overall execution time. Changing the chunksize in the test range of 1 to 64 does not affect **GUIDED_N**'s performance too much.

The **AUTO** option still shows no advantage on ***loop1( )*** as well as the **STATIC** option. The **STATIC** outermost loop's iteration space is divided into chunks that are approximately equal in size. However, the problem size of each iteration of the outermost loop in ***loop2( )*** has very large differences due to their double-nested inner loops. The work is finally found not enough balanced distributed compared to other scheduling strategies.

Speedup for Loop 1

Following the observation in the previous experiment, let the schedule method be **GUIDED**, and chunksize is equal to 2 for ***loop1( )***.

(Serial execution time for 1000 reps of loop 1 = 10.973513 seconds)

NUM_THREADS	1	2	4	6	8	12	16	24	32
(GUIDED, 2)	11.828	5.523	2.957	2.152	1.677	1.157	0.889	0.732	0.689
Speedup	0.928	1.987	3.710	5.099	6.543	9.486	12.338	15.000	15.916

Table3. the speedup (T1/Tp) for *loop1( )* using (guided,2) versus number of threads

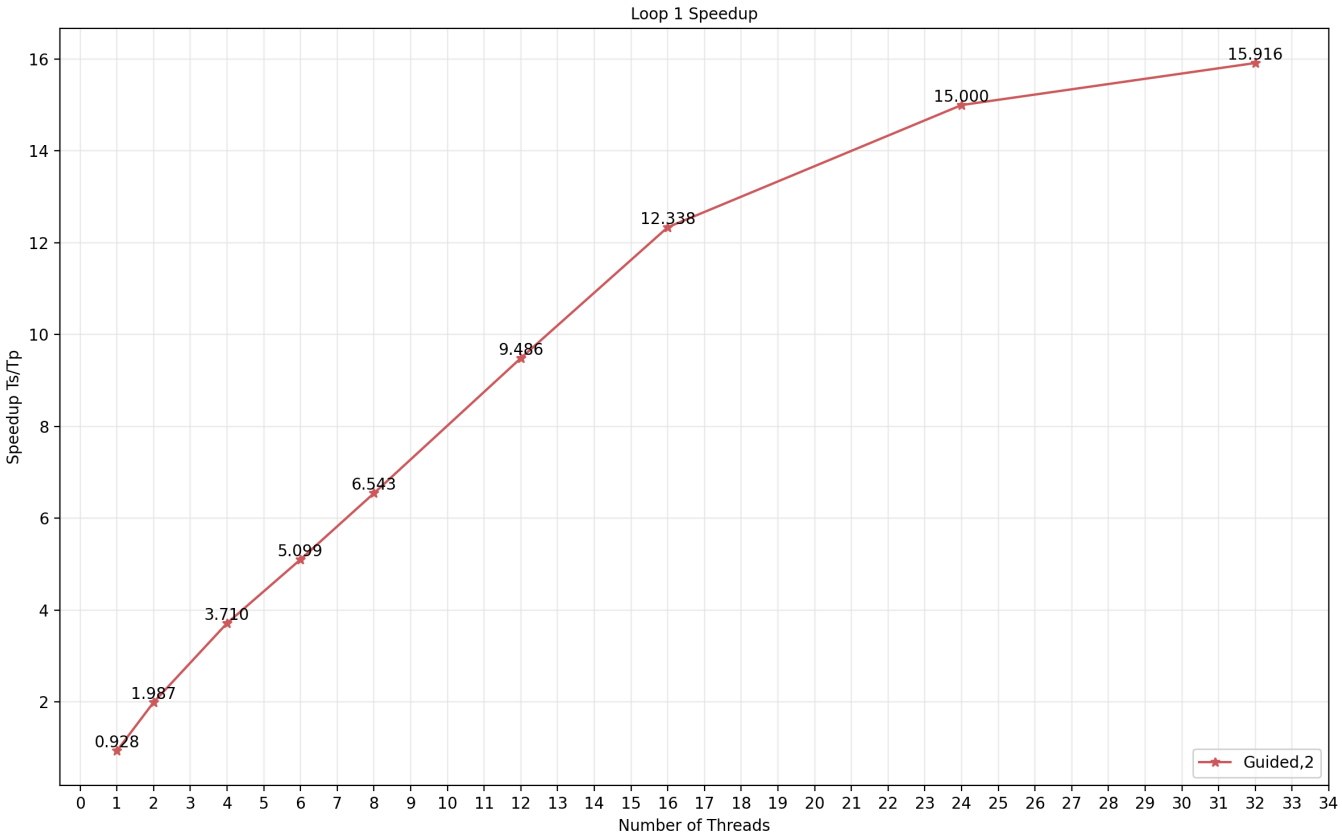

Fig3. the speedup (T1/Tp) for *loop1( )* using (guided,2) versus number of threads

The ***loop1( )*** speedup increases as the number of threads increases. At the number of threads equals 1, the parallel performance is similar but less than the serial. That may be caused by the redundant parallel scheduling operation of the master thread. There is only the master thread which is overall no different from serial execution, but the extra cost is paid for parallel environment setup.
When the thread number equals 2, the execution speedup is approximately doubled. The performance is ideally linearly optimising n times better when the number of threads increases n times. The fact is that the scheduling overheads are increasing for those parallel scheduling computations. So the increase rate of speedup decreases as the number of threads increases.

When the number of threads increased from 24 to 32, the speedup only increased less than 1. A lot of resources are taken up by the threads managing instead of solving problems. If we continue to increase the number of threads, we can assume that the performance cannot increase anymore.

Speedup for Loop 2

Following the observation in the previous experiment, let the schedule method be **DYNAMIC**, and chunksize is equal to 4 for ***loop2( )***.

(Serial execution time for 1000 reps of loop 2 = 12.701866 seconds)

NUM_THREADS	1	2	4	6	8	12	16	24	32
(DYNAMIC,4)	13.532	6.535	3.455	2.450	1.890	1.352	0.997	0.882	0.887
Speedup	0.939	1.944	3.676	5.185	6.722	9.392	12.741	14.395	14.322

Table4. the speedup (T1/Tp) for *loop2( )* using (Dynamic,4) versus number of threads

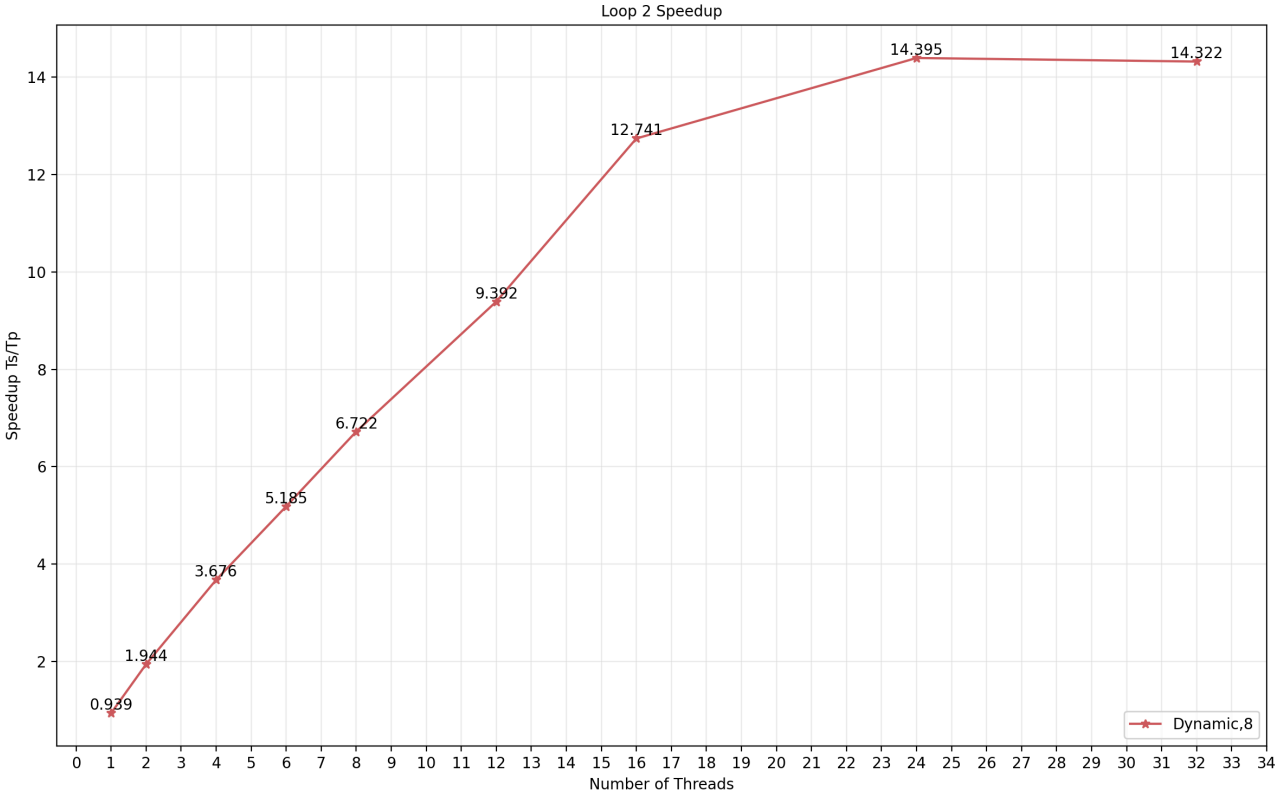

Fig4. the speedup (T1/Tp) for *loop2( )* using (dynamic,4) versus number of threads

The ***loop2( )*** speedup increases as the number of threads increases. The case of 1 thread parallelises problem is the same as under serial execution but has the redundant parallel scheduling operation of the master thread. ***Loop2( )*** has to pay for the extra cost of parallel environment setup even though it won't benefit the performance.

When 2 threads are employed to share the work, the execution speedup is approximately doubled. The performance cannot be optimised linearly to the number of threads. Scheduling overheads increase for those parallel scheduling computations when we increase the number of threads. When the number of threads increased from 24 to 32, the execution time was even longer. The cost of task distribution negates the benefit of multithreading.

Conclusion

The effects of scheduling strategies are very dependent on the problem itself. The factors could be the smallest and largest chunk, the variance of task size, how much the task can be divided, and also the number of threads and the chunksize. In the report, we first control other variances and iterate the chunksize; second, we only iterate the thread number with fixed chunksize and scheduling strategies. However, there are still factors out of control that influence the accuracy of the resulting data. The runtime of every experiment supposes to be constant if we assume every thread of the CPU is identical. Ideally, they have the same amount of CPU resources, use the same amount of time to access and write the data, and so on. However, every single experiment gives a different runtime value, and we have to estimate the runtime by calculating the mean.

The disadvantage of using multi-threading has also been exposed in the experiment. The checksum of ***loop2( )***'s calculation by using multiple threads differs from the serial because of round error.

Speedup is mainly determined by the scheduling method and the thread number. Power overheads and other factors should be considered in real-world cases. The trade-off is required while purely increasing the threads number could not benefit the performance efficiently.