|epcc|

Threaded Programming, Coursework2

s2195209, Cheng Xu

October 29, 2021

Contents

1 Introduction **1**

2 Experimental Environments and Design **2**
2.1 Cirrus and Compute Nodes 2
2.2 MobaXterm, Slurm File, and Compiler 2

3 Methodology **3**

4 Experiment Process **4**

5 Results **5**
5.1 Performance for different scheduling strategies and chunksizes 5
5.2 Performance for STATIC and AUTO strategy 6
5.3 Optimal strategies for different loops 6
5.3.1 Optimal Strategy for loop1 6
5.3.2 Optimal Strategy for loop2 7

6 Analysis **9**
6.1 Analysis for loop1 9
6.2 Analysis for loop2 10
6.3 Analysis for overall optimal strategy 11

7 Conclusions and Takeaways **12**

1 Introduction

In order to improve the performance of a running program, Threaded-Level Parallelism is often adopted and requires us more attention to investigate on. In this article, several environment configurations and the methodology of this coursework are introduced at the beginning. After different schedule strategies of multi-threaded programming in OpenMP concerning a program containing two "for loops" are introduced, general descriptions along with basic visualizations are forwarded to show the trend of running time, speedup, and the optimal schedule strategy. Furthermore, an analysis regarding the result is given, followed by an attempt to unveil the underlying mechanism behind this ostensible discrepancy. A brief conclusion is provided to summarize the full text.

2 Experimental Environments and Design

2.1 Cirrus and Compute Nodes

Cirrus, which is High-Performance Computing hardware maintained by EPCC from the University of Edinburgh, plays an essential role in running the threaded-level parallel program. Although programs can be running on both compute nodes and login nodes in cirrus, it is highly recommended to run the parallel tasks on compute nodes for more computing resources and fewer congestions. According to the cirrus website, there are 282 compute nodes, most of which are standard type, while 2 of which contain GPU accelerators.

2.2 MobaXterm, Slurm File, and Compiler

To utilize compute nodes, a slurm file is ought to be used to ask for compute nodes on cirrus as well as wait in line for the resources to be assigned. Several arguments need to be set properly to run slurm file for exclusive access to a specific compute node. The intel-compilers-19 module is loaded as an intel C++ compiler to compile the OpenMP program. MobaXterm is utilized as a remote control in Windows System to SSH cirrus. Alternatives of remote control to SSH cirrus could be PuTTY or Xshell, which are often being used in many ways.

3 Methodology

In order to better investigate which strategy gives the best performance, the Variable Controlling method is adopted in this literature, which means when one variant is being investigated and changed, other variants must be maintained stable. This literature focuses on the performance of two different loops with separate scheduling strategies and split up the experiment into three parts:

1. Running program using STATIC, AUTO scheduling strategies respectively without specifying the chunksize in STATIC.
2. Setting different values of chunksize with respect to scheduling strategies STATIC, DYNAMIC, GUIDED separately.
3. Finding the optimal scheduling strategies regarding two different loops and running the program on distinct threads numbers.

After running the program, visualization and relevant explanation will be given following the experimental process.

4 Experiment Process

A general command used to run the program are listed below:

```
module load intel-compilers-19
```

intel compilers must be specified in the beginning in order to run the OpenMP.

In order to speed up the running process, a bash file is written to specify different environment variables such as chunksize and strategies, and then it is provided to the slurm file. The slurm file is running on computed nodes using

```
sbatch XXX.slurm
```

command, and its relative outputs are stored in a slurm-XXXX.out file, where we can extract the values we need in the visualization step. Several possible ways could be used to get the output simply. For example, a for loop could be written in the bash file to print all the values we need separated by comma delimiter, which is convenient for them to be copied to a Python program; or the Regular Expression package could be used in Python to extract corresponding value in the text, but that requires further investigations.

Accuracy is important when dealing with running time problems. Error approximation could be minimized by averaging the value generated. For the sake of getting an accurate value, each program with a fixed scheduling strategy is running 10 times. However, some values could fall within the region very far away from the real average, those errors might be partly removed by setting an acceptable region, only the value fall within which would be kept calculating the final average. In this coursework, $(-3 * \sigma, +3 * \sigma)$ is set to be the acceptable region, where σ represents the variance with respect to data points in a single file. Another thing worth mentioning is that the Chebyshev Equation could be considered to give mathematical proof of calculating the appropriate average, but the premise is based on the large dataset, which is not required in this coursework.

Then matplotlib package in Python is being used to plot the data generated from the program and comparison could be displayed using visualization.

5 Results

Each program with a different scheduling strategy is running ten times. The standard variance is calculated per program. Based on the three-sigma rule, which is 68-95-99.7 rule, values falling within the three-sigma region are accepted to calculate the average. And the corresponding graphs indicating the performance vary with different settings are given.

5.1 Performance for different scheduling strategies and chunksizes

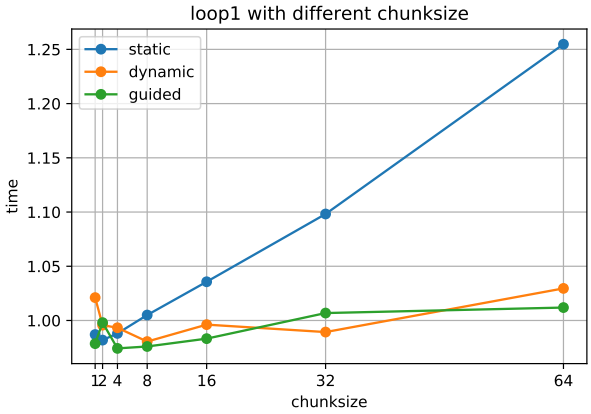

Figure 1: loop1 with different chunksizes

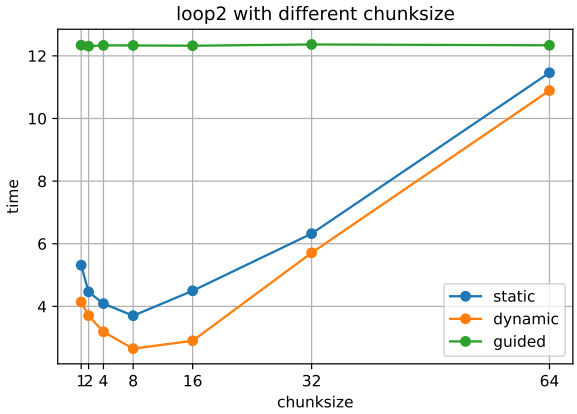

Figure 2: loop2 with different chunksizes

Figure 1 and Figure 2 display the performance for loop1 and loop2 respectively. For each loop, three scheduling strategies, STATIC, DYNAMIC, and GUIDED, are used to

be compared. Each scheduling strategy is running with different chunksize specified, as the x-axis indicates in every figure. It turns out that both STATIC AND GUIDED strategies will reach the local minimum, which is the optimal performance when the chunksize specified as 8. Any chunksize smaller or bigger than 8 will reduce the optimal performance for STATIC and GUIDED. In the meanwhile, the performance of the DYNAMIC scheduling strategy does not depend on the different settings of chunksize and always stays at a relatively bad level as it takes more time to execute the code than other strategies.

5.2 Performance for STATIC and AUTO strategy

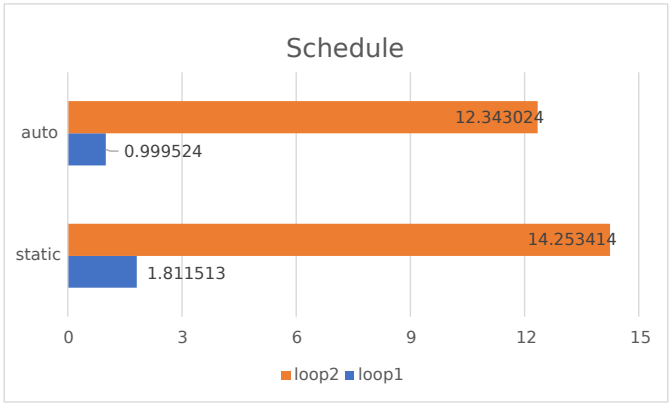

Figure 3: Performance for STATIC and AUTO

Figure 3 indicates that the AUTO strategy always performs better than the STATIC one with respect to different chunksize. Therefore, another experiment using this scheduling strategy with different thread numbers has been running to show how thread number effect the performance of threaded-parallelism.

5.3 Optimal strategies for different loops

5.3.1 Optimal Strategy for loop1

For loop1, the optimal scheduling strategy is (guided, 4) with 8 active threads running simultaneously.

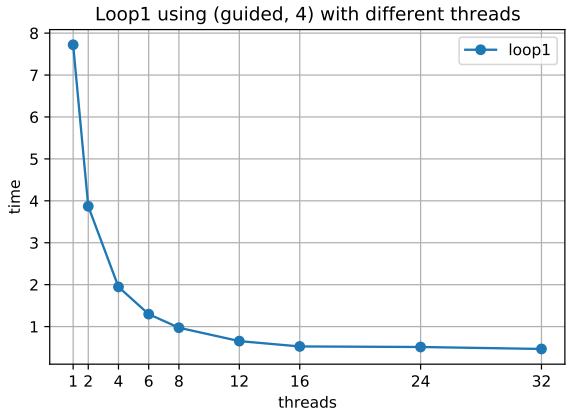

Figure 4: Loop1 using (guided, 4) with different threads

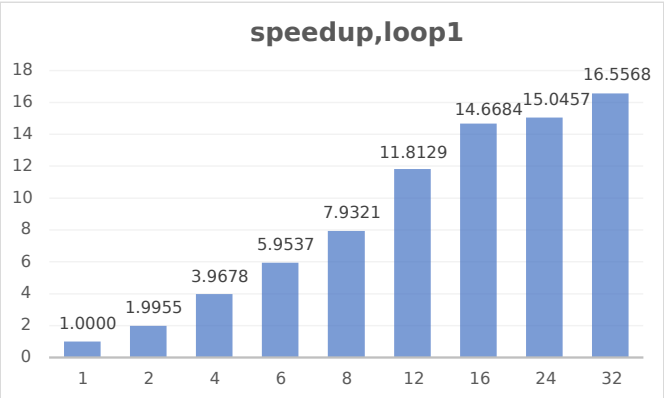

Figure 5: speedup for loop1

Figure 4 illustrates that the improvements of performance when the scheduling surge along with the increment of threads number until which reaches 16. After that, the performance still enhances at a relatively slow pace.

Figure 5 shows the speedup with respect to the different number of threads. The optimal performance for loop1 occurs when 32 threads executing the for loop concurrently, with a speedup of 16.5 approximately.

5.3.2 Optimal Strategy for loop2

For loop1, the optimal scheduling strategy is (dynamic, 8) with 8 active threads running simultaneously. Therefore, another experiment using this scheduling strategy with different thread numbers has been running to show how thread number influence the performance of this parallel loop.

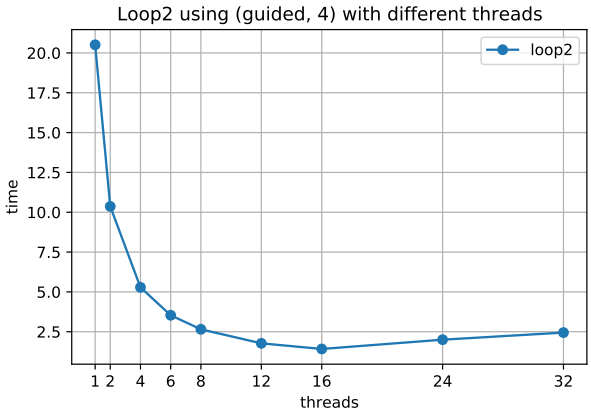

Figure 6: Loop2 using (dynamic, 8) with different threads

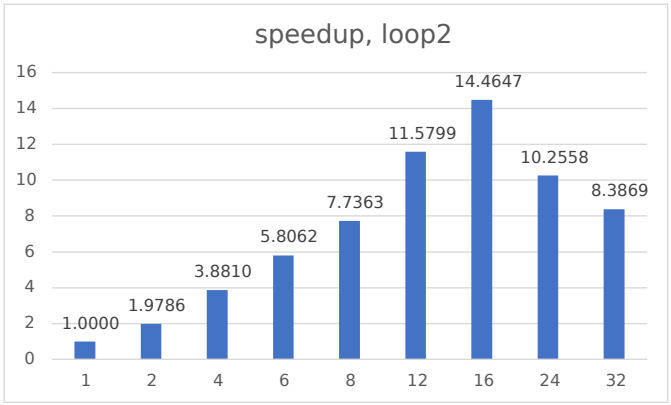

Figure 7: speedup for loop2

Figure 6 shows that the elapsed time, as loop1, decreased dramatically when executing loop2 using relatively small thread numbers. However, the performance starts to get worse when the number of threads is configured as a number bigger than 16.

Figure 7 indicates that the speedup with respect to the different number of threads for loop2. The optimal performance for loop2 occurs when 16 threads execute the for loop concurrently, with a speedup of 14.5 approximately.

6 Analysis

6.1 Analysis for loop1

```
#define N 1729

double a[N][N], b[N][N]

void loop1(void) {
  int i,j;
#pragma omp parallel for default(none) shared(a,b)\
private(i,j) schedule(static)
  for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }
}
```

The code of the implementation of loop1 is provided above. In this case, the OpenMP directives are put before the outer loop, which means the program is divided across the outer loop. Each part will be assigned to different threads according to the scheduling strategy given in the directives. When it comes to the inner loop, a clear fact could be observed is that the number of iterations of inner loops decreases as i (index of the outer loop) increases, which leads to another point that the workload for the first few iterations of the outer loop is much heavier than the last few ones.

Based on the points mentioned above, the worst performance occurs when adopting the STATIC scheduling strategy without specifying chunksize. In this case, loop1 is split up into n pieces of relatively equal size and assigned to every thread correspondingly, where n represents the number of active threads. Therefore the thread assigned to the first piece of chunk would have a hefty workload. Hence, loop with STATIC performs poorly if no chunksize has been specified or chunksize has been set to a large number since one thread would have to execute much more inner loop iterations than other threads, causing load imbalance and the fact that other threads have to wait until the thread with heavier workloads completes before the implicit barrier. However, suppose the loop adopts a STATIC schedule with small chunksize. In that case, the load imbalance could be improved as every thread would execute consecutive iterations, ensuring that each threads' workloads are balanced since as the index of the outer loop increases by one, only one iteration of the inner loop is reduced. Furthermore, DYNAMIC and GUIDED scheduling strategies perform relatively the same since they are both on a first-come-first-served basis. Thus, when a thread with more inner iterations executes the code, other threads keep being assigned with small loads without waiting for one thread to complete.

6.2 Analysis for loop2

```
1  void loop2(void) {
2    int i,j,k;
3    double rN2;
4    rN2 = 1.0 / (double) (N*N);
5
6  #pragma omp parallel for default(none)\
7  shared(b,c,jmax,rN2) private(i,j,k)\
8  schedule(static)
9
10   for (i=0; i<N; i++){
11     for (j=0; j < jmax[i]; j++){
12       for (k=0; k<j; k++){
13         c[i] += (k+1) * log (b[i][j]) * rN2;
14       }
15     }
16   }
17 }
```

The code of the implementation of loop2 is provided above. The performance of the loop depends on the initialization part since the value of jmax is not clear. So a Python code to show the output of this initialization process has been written.

```
1  N = 1729
2  jmax = list(range(N))
3  for i in range(N):
4      expr =  i%( 4*(i/60) + 1)
5      if expr == 0:
6          jmax[i] = N/2;
7      else:
8          jmax[i] = 1
9
10 f = lambda x:jmax[x] != 1
11 for i in range(N):
12     if f(i):
13         print('The index where jmax[index] is not 1:', i)
14         print(i)
```

```
1  The index where jmax[index] is not 1: 0
2  The index where jmax[index] is not 1: 30
3  The index where jmax[index] is not 1: 60
4  The index where jmax[index] is not 1: 210
```

From this piece of Python code, a clear fact is that only four elements in jamx contain a value other than 1, but N/2, which is 864.5. When looking through the index indi-

cating the element includes an enormous value, a straightforward observation is that those indices most fall within the first hundred inner iterations. Therefore, STATIC schedule without specifying the chunksize and GUIDED schedule do not perform well because a thread assigned to the first hundred outer iterations would have enormous inner iterations workloads, and other threads could do anything but wait for that thread completes. That is because the STATIC schedule divides the outer iterations into pieces with equal size, and the GUIDED schedule makes chunks start large and become exponentially smaller. For DYNAMIC and (STATIC, n) scheduling strategies, they are both first-come-first-served. Therefore the workload could be better balanced as well as the situation in loop1.

6.3 Analysis for overall optimal strategy

Generally, the performance would be improved as the number of active threads increases initially but then slows down as thread numbers reach a considerable level. A possible reason for that is that the number of threads could help implement Threaded-Parallelism better to some extent, but when more threads are forked, the overhead of creating a new thread can not be ignored.

7 Conclusions and Takeaways

1. Never rely on default scheduling strategies. Different compilers may have disparate implementations. It is always a good practice for the user or programmer to set the schedule in OpenMP directives or corresponding environment variables manually and adequately based on the principle of maintaining load balance across each thread.

2. AUTO is a compiler-specific strategy. By giving the compiler some freedom to choose its partition over iterations to threads, the compiler could find a relatively low overhead and good load balance strategy. However, this strategy may be better than STATIC but way worse than the optimal strategy that the programmer set manually. Therefore, in practice, it is a good habit to choose the scheduling strategy manually after carefully analyzing the parallel loop.

3. STATIC without specifying the chunksize should be used carefully, and it can only be best utilized in load-balanced loops. (STATIC, n) could be used in a smooth load-imbalanced program, though the chunksize should be carefully set. DYNAMIC and GUIDED could be used when encountering varying load-balanced loops as they both have a first-come-first-served basis. However, more attention should be paid to the GUIDED scheduling strategy since one of the threads would be assigned to hefty workloads initially. The programmers' responsibility is to make sure the first few iterations are not the most expensive ones.

4. It is convenient to use RUNTIME in the program and modify the environment variables in the runtime. For example, put all the commands and variables exporting statements together in a bash file.