Threaded Programming, Coursework3

B203104

December 6, 2021

Contents

1 Introduction

OpenMP Library provides many ways of implementing Threaded-Parallelism, among them work scheduling algorithms. However, more scheduling algorithms could be implemented on the part of the user instead of conventional strategies in the OpenMP such as STATIC, DYNAMIC, and GUIDED. Flexible as they might be, there are some grounds for concern, notably tacking race conditions could be complicated to some extent.

In this coursework, a new loop scheduling algorithm, the Affinity Scheduling algorithm, is introduced, implemented, and applied to a given piece of code. In this coursework, several environment configurations and the methodology of this coursework are introduced at the beginning. A description of the algorithm and its pseudocodes will then elaborate on how the Affinity Scheduling works, followed by the proof for the correctness and the comparison for the performance. Finally, discussions and conclusions focusing on unveiling the underlying mechanism behind the ostensible discrepancy will be provided at the end of the coursework.

2 Experimental Environments and Design

2.1 Cirrus

This coursework runs all the programs on one of the EPSRC Tier-2 National HPC Services called Cirrus. Cirrus, the Tier-2 High-Performance Computing hardware based on the SGI ICE XA system, maintained by the EPCC from the University of Edinburgh, plays an essential role in running the threaded-level parallel program. According to the cirrus website, there are 280 standard compute nodes, each of which has two 2.1 GHz (clock frequency), 18-core Intel Xeon E5-2695 (Broadwell) processors with 256 GB of memory, which is arranged in NUMA form. Meanwhile, two hyper threads are activated by default on each of the cores on these processors. Furthermore, 38 GPU nodes and their GPU accelerators are also contained in the Cirrus; however, they will not be used in this coursework, and its detailed description will be omitted here.

Although programs can be run on both compute nodes and login nodes, it is highly recommended to run the parallel tasks on compute nodes for more computing resources and fewer congestions.

2.2 Configurations and Settings

To run the program successfully, a slurm script is often used to submit the job to the scheduler, which manages hardware resources, assigns memory to several users simultaneously, and avoids congestions. Based on the request from the slurm file, the Slurm scheduler software then allocates CPU cores on compute nodes to run the program. Therefore, multiple options in the slurm file must be determined beforehand.

1. #SBATCH –nodes=1
2. #SBATCH –tasks-per-node=1
3. #SBATCH –cpus-per-task=36
4. #SBATCH –partition=standard
5. #SBATCH –qos=short

QoS stands for Quality of Service; when set to short, it would only allow two slurm jobs submitted to the compute nodes. The standard way is the only available choice for partitions to which general users could access. Most of the time, the whole job runs on one single compute node, so no extra communication overhead across nodes and sockets will happen. However, since the number of threads per parallel process is set to 36 by tasks-per-node options, communications across different nodes must be considered then.

Although the task-per-node has already been set, it is still imperative to set the environment variable OMP_NUM_THREADS before running any program. An example provided below is setting 16 active threads.

```
export OMP_NUM_THREADS=16
```

Finally, an intel compiler must be loaded to run the program with the srun command.

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

```
srun --cpu-bind=cores ./loops2
```

3 Affinity Scheduling

Simulating this scheduling strategy requires that every thread be assigned and maintain a local set, which records all the iterations needed to be executed for a single thread. Meanwhile, a thread would get a chunk from either its own or other threads' local set, and the information stored from this chunk would be passed as arguments into the actual loop. The thread will then complete all the iterations stored in the chunk.

3.1 Shared Data structures

In order to better simulate the real situation and reduce the complexity, two struct types are utilized in this program: struct local_set and struct chunk.

```
struct local_set {
  int id;
  int low;
  int high;
  int length;
  int complete;
  omp_lock_t lock;
};
```

```
struct chunk {
  int low;
  int high;
  int chunksize;
};
```

For the local set, six attributes are included in this struct type. The id indicates which thread maintains this local set instance; the low, high, and length attributes represent the range for the remaining local set; a complete attribute will be marked as 1 when the local set for a particular thread is run out; a lock attribute is included for preventing race conditions from happening.

All the attributes included for the chunk struct type show which iterations (from low to high) a thread will execute.

Successful chunk searching and stealing requires that not only does every thread get access to its own local set but also to the local set maintained by other threads. Therefore, a shared variable storing every local set in order should be declared outside the parallel region and initialized by only one thread inside the region. All the local sets could be initialized dynamically through an array allocated spaces by the malloc() function. Each element in this array represents a pointer to a local set, and the index of the element shows which thread maintains that pointer. The initialization for the array storing local sets is completed by the local_set_storage_init(), while every pointer to a local set is initialized with the local_set_init().

The functionality of the chunk initialization step is basically the same process so the details is omitted here. Although the array storing chunk is initialized as a shared variable, it does not allow other threads to access; therefore, no synchronization must be applied in this data structure.

The figure below shows a shared data structure: local_set.

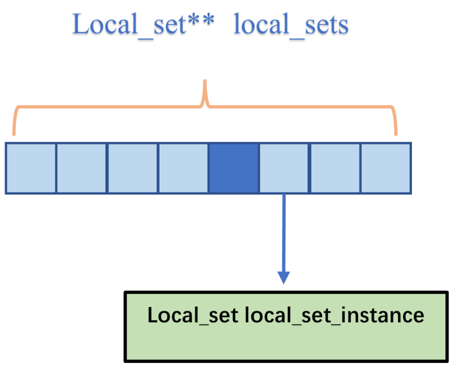

Figure 1: Shared Data Structures

3.2 Synchronization

The efficiency of the work scheduling algorithm can not run counter to the accuracy, so the fundamental step for implementing the Threaded-Level Prarllelism is to ensure the correctness of results after execution and calculation.

Race conditions could happen when multiple threads simultaneously try to get access to and update the value for a shared variable. Synchronization techniques in OpenMP could tackle that issue, such as the critical region, atomic directive, and lock routines. The atomic directive is the most efficient one when protecting a single update to a variable, though it has strict restrictions. The critical region is flexible somehow, but the efficiency will be reduced since the critical region could be illustrated as a global lock. In this experiment, lock routines are adopted to protect local sets. Because of the existence of the steal_work function(), multiple threads could iterate over every local set to find possible chunks. Therefore, a lock must be set when a thread enters a local set to check any available chunks. The lock could not be unset until a thread makes sure either the chunk has already been acquired, or the local set is not available for more chunks.

3.3 Algorithm Description

After the initialization and declaration step, the algorithm will be implemented with a function called assign_work() inside the runloop() method. The assign_work function is mainly used to allocate different chunks to each thread. The chunk, whose size is the magnitude of the remaining iterations of its local set divided by the number of threads, is initially derived from its own local set. The thread, whose local set has been exhausted, will then steal works from another thread with a most loaded incomplete local set.

The task for finding a chunk of the thread itself is implemented by the find_chunk() function, and the steal_work() function will be used to get a chunk from another local set, with the help of the find_most_loaded() function aiming at searching a local set with the most remaining iterations. The dependency flow is provided below.

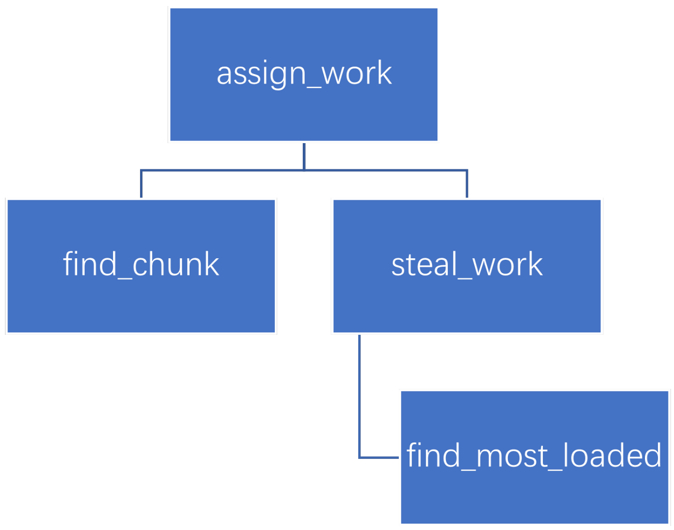

Figure 2: Flow

The digraph shown below illustrates how the algorithm works. The red line in the figure indicates the size of the local set for every single thread. Thread 0 completes all the iterations from its local set and steals chunks from Thread 1 and 2, while Thread 1 steals a chunk from Thread 2, and Thread 2 does not steal any work. Every colored box represents a chunk to be executed and the while one represents the chunk stolen from the current thread.

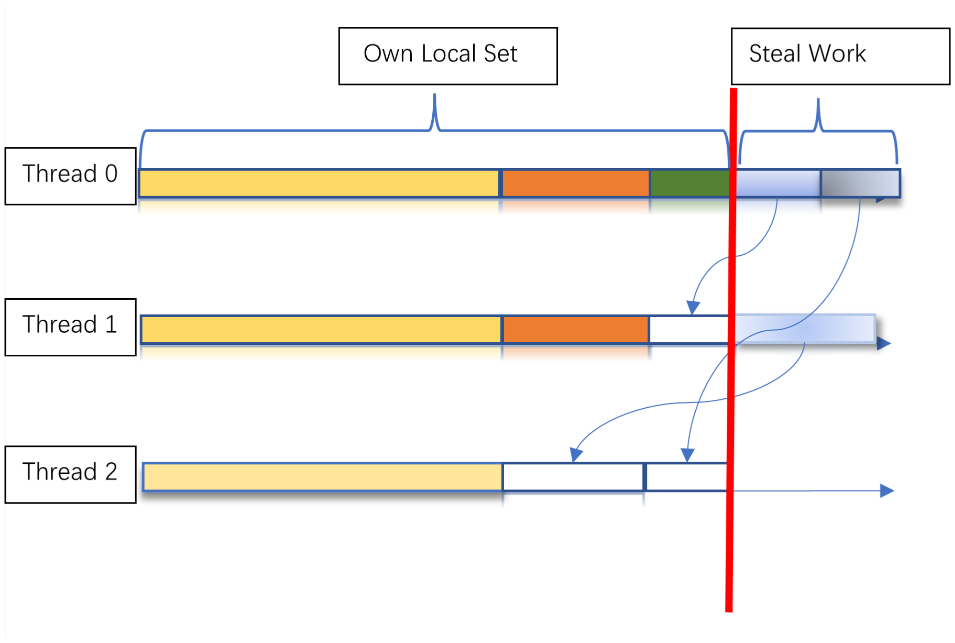

Figure 3: Digraph

4 Pseudocode

Algorithm 1: Assign_work

```
Input: $local\_sets \quad (type : local\_set**)$,
       $chunks \quad (type : chunk**)$,
       $int \quad thread\_num$, a unique id for a thread,
       $int \quad num\_threads$, the total number of threads.
Output: $flag$, if successful, return 1; else return 0
1 Initialization: $local\_set\_instance = local\_sets[num\_thread]$;
2                 $chunk\_instance = chunks[num\_thread]$;
3 flag = find_chunk(), find an available chunk from its own local set;
4 if $flag != 0$ then
5     flag = steal_work(), try to steal work from other threads' local_set;
6     return $flag$;
7 else
8     return $flag$;
9 final ;
```

Algorithm 2: Steal_work

```
Input: $local\_sets \quad (type : local\_set**)$,
       $chunks \quad (type : chunk**)$,
       $int \quad myid$, a unique id for a thread,
       $int \quad num\_threads$, the total number of threads.
Output: $flag$, if successful, return the thread id with most loaded local set;
        else return 0
1 int most_loaded_set (id for most loaded local set) = find_most_loaded(), find an
  available chunk from other threads's local set, return the id of that;
2 if $most\_loaded\_set != -1$ then
3     int a = find_chunk(), try to steal work from this specified local_set;
4     return $a$;
5 else
6     return 0;
7 final ;
```

Algorithm 3: Find_chunk

```
Input: $local\_set$ $(type : local\_set*)$,
       $chunk$ $(type : chunk*)$,
       $int$ $thread\_num$, a unique id for a thread,
       $int$ $num\_threads$, the total number of threads,
       $int$ $steal$, indicates whether or not the chunk is stolen from other threads.
Output: $flag$, if successful, return 1; else return 0
1  if $steal\,! = 1$ then
2      set lock for this local set, prevent other threads from accessing this set again.
3  if $local\_set.complete == 1$ then
4      unset its lock, does not find any chunk from this local set;
5      return 0;
6  if $local\_set.legnth > num\_threads$ then
7      Evaluate chunksize to a fraction 1/num_threads of remaining iterations from its
         local set(local_set.length);
8  else
9      Evaluate chunksize to the number of remaining iterations from its local
         set(local_set.length);
10 For the chunk, the low and high attributes will be set to new values:
     chunk.low = local_set_low;
     chunk.high = local_set_low + chunksize -1;
11 For the local set, the attributes will be updated as well:
     local_set.low += chunksize;
     local_set.high remains the same;
     local_set.length = local_set.high - local_set.low + 1;
12 if $local\_set.legnth == 0$ then
13     local_set.complete = 1;
14 unset the lock for this local set.
15 return 1;
16 final ;
```

Algorithm 4: Find_most_loaded

```
Input: $local\_sets \quad (type : local\_set **)$,
       $int \quad thread\_num$, a unique id for a thread,
       $int \quad num\_threads$, the total number of threads.
Output: $flag$, if successful, return 1; else return 0
1  Declaration:
     local_set* most_loaded_set = NULL;
     local_set* temp_loaded_set = NULL;
     int max_length = 0;
     int temp_length= 0;
2  Initialization:
     for each $i$ in $range(num\_threads)$ do
3        if $i$ == thread_num then
4            continue;
5        temp_loaded_set = local_sets[i];
6        set the lock;
7  temp_loaded_set = NULL;
8  for each $i$ in $range(num\_threads)$ do
9      temp_loaded_set = local_sets[i];
10     if temp_loaded_set.complete then
11         continue;
12     temp_length = temp_loaded_set;
13     if $temp\_length > max\_length$ then
14         max_length = temp_length;
15         most_loaded_set = temp_loaded_set;
16 for each $i$ in $range(num\_threads)$ do
17     if $i$ == thread_num then
18         continue;
19     if most_loaded_set != NULL then
20         if $i$ == loaded_set.id then
21             continue;
22     unset the lock for every local set;
23 if most_loaded_set != NULL then
24     final $-1$;
25 else
26     return $local\_set.id$;
27 final ;
```

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 1oop1

For loop1, the optimal scheduling strategy is (guided, 4) among all the embedded work scheduling algorithms.

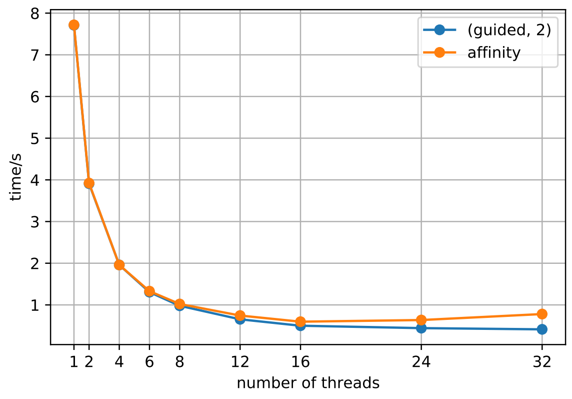

Figure 4: Running time for Loop1

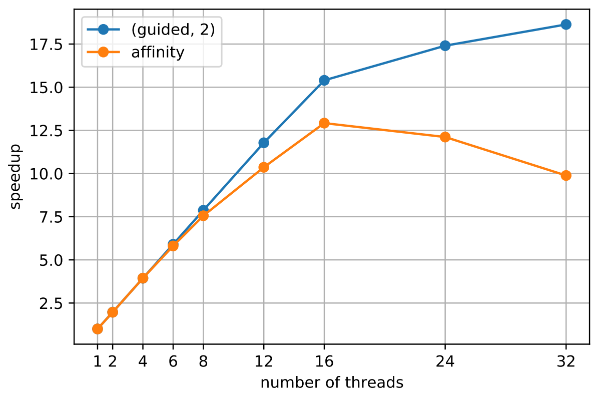

Figure 5: Speedup for Loop1

Figures above display the performance and speedup for loop1, respectively. For each loop, scheduling strategies Affinity and (GUIDED, 2) are used to be compared. Each scheduling strategy is running with different number of thread specified, as the x-axis indicates in every figure.

It turns out that both of these strategies will share similar trends. However, when the thread number is larger than 12, a slight difference appears that the Affinity Scheduling strategy behaves slightly worse than the guided one. For the speedup, the Affinity strategy first increases dramatically and then drops when the thread number reaches 16.

5.2 Performance for loop2

For loop2, the optimal scheduling strategy is (dynamic, 8) among all the embedded work scheduling algorithms.

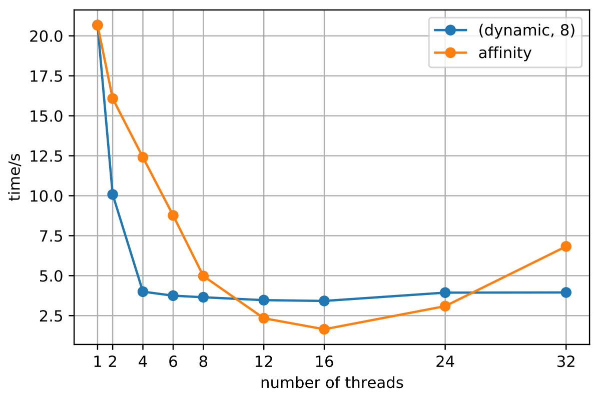

Figure 6: Performance for Loop2

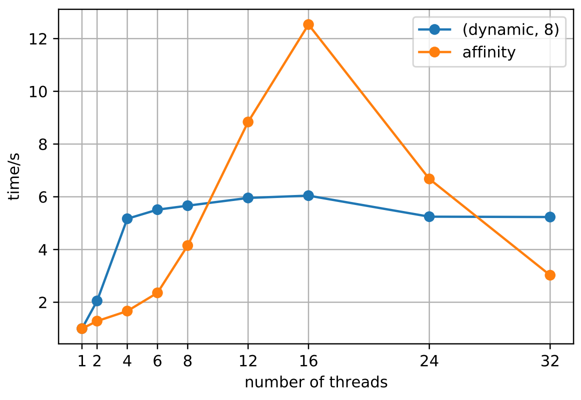

Figure 7: Speedup for Loop2

Figures above display the performance and speedup for loop2, respectively. For each loop, scheduling strategies Affinity and (DYNAIMIC, 8) are used to be compared. Each scheduling strategy is running with different number of thread specified, as the x-axis indicates in every figure.

It turns out that both strategies will reach the local minimum, which is the optimal performance when the chunksize specified as 16. The elapsed time 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. Both strategies share a similar trend as well as in loop1.

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.

Unlike STATIC, both Affinity and GUIDED scheduling strategies do not split up the loop equally; thus, neither thread would execute a hefty load with all other threads waiting before the implicit barrier. The GUIDED strategy is on a first-come-first-served basis; hence, every idle thread would be assigned to a new piece of loads until completing the iterations. At the same time, the affinity executes consecutive iterations at the beginning and then steals or gives out the iterations, which better balances the loads across different threads. Because they both handle the imbalance issue pretty well, their performance changes share an identical trend with minor fluctuations. To be more precise, the Affinity Strategy performs worse to some extent. That might be due to the overheads for iterating over every local set from a different thread to find the most loaded one, while GUIDED does not have to know the information other threads maintain.

6.2 Analysis for loop2

```
void loop2(void) {
  int i,j,k;
  double rN2;
  rN2 = 1.0 / (double) (N*N);

#pragma omp parallel for default(none)\
shared(b,c,jmax,rN2) private(i,j,k)\
schedule(static)

  for (i=0; i<N; i++){
    for (j=0; j < jmax[i]; j++){
      for (k=0; k<j; k++){
        c[i] += (k+1) * log (b[i][j]) * rN2;
      }
    }
  }
}
```

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.

```

N = 1729
jmax = list(range(N))
for i in range(N):
    expr =  i%( 4*(i/60) + 1)
    if expr == 0:
        jmax[i] = N/2;
    else:
        jmax[i] = 1

f = lambda x:jmax[x] != 1
```

```

for i in range(N):
    if f(i):
        print('The index where jmax[index]
            is not 1:', i)
        print(i)
```

```

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

From the messages printed, a clear fact is that only four elements in jamx contain a value other than 1, but 864.5. When looking through the index of element holding a large value, an apparent observation is that most of those indices fall within the region of the first hundred inner iterations. Therefore, both strategies, either on a first-come-first-served basis or can steal and give out chunks, perform well in this case as the load balances are allocated well. However, the Affinity strategy performs slightly better than the DYNAMIC one, as the DYNAMIC assigns a fixed number of iterations or workloads to each thread – the thread then must complete those to get a new one. Meanwhile, the Affinity one with more flexibility assigns iterations based on the remaining number stored in the local set.

7 Correctness

As mentioned above, the efficiency and performance improvements must not run counter on the accuracy and correctness. Therefore, two valid() functions are provided beforehand to check the correctness. Any modifications of the work scheduling methods can not affect the original value of the array. Furthermore, the programmer could print all the iterations numbers explicitly onto the screen and pass that to a Python code. For example, the Python code shown below returns a list. If the list becomes empty and no exceptions are thrown after several executions, then it can be proved that no repeated or omitted iterations exist.

```
a = [i for i in range(0, 1730)]
def f(start, end, a):
    for i in range(start, end+1):
        a.remove(i)
    return a
a = f(857, 865, a)
...
```

8 Conclusions and Takeaways

1. More flexible methods could be implemented, in addition to work scheduling algorithms embedded in OpenMP.
2. There are no optimal scheduling strategies for all situations. The scheduling algorithms could only be determined after analyzing the loop, and the analysis must be based on the principle of maintaining load balance across each thread.
3. The newly implemented Affinity Scheduling algorithm performs well for imbalanced loaded loops, allowing each thread to steal and give out the iterations. Usually, the loops are imbalanced; therefore, careful consideration for the work scheduling methods is significant.
4. Synchronization is essential for Multi-threaded Programming. The efficiency and performance improvements must not run counter on the accuracy and correctness. The critcal region, lock routines, single region, and atomic directive should be adopted when encountering race conditions.