MSc in High Performance Computing Coursework for Threaded Programming Part 2

Exam Number: B129230

1. Introduction

Threaded programming is mainly based on OpenMP to implement parallel programs. OpenMP is a set of compiler directives and library routines which can specify parallelism in Fortran and C. The purpose of this report is to describe author's own customized affinity scheduling for two loops and compare the results with the best OpenMP schedule in Part1 of the coursework.

2. Experiments

2.1 Experimental Setup

The whole program which contains two loops is running on the 'Cirrus' by using PBS(portable batch system) to submit.

2.1.1 Computing Environment.

'Cirrus' is a SGI ICE XA system with 280 compute nodes 10, 080 cores, utilizing Infiniband interconnect. Each node provides 36 physical cores which use Intel(R) Xeon(R) E5-2695 v4, clocked at 2.1GHZ, and contains 256GB RAM per node.

This program is compiled with icc version 17.0.2. The optimization level is set to -O3.

2.1.2 Experimental Steps

The first step of this experiment is to design author's own affinity scheduling and then select one full node to run customized affinity schedule for each loop by using different number of threads and lastly count average time to draw graphs of the execution time for each loop compared with the best OpenMP schedule in Part1 of the coursework.

2.1.3 PBS(portable batch system).

This experiment uses PBS script to submit jobs, control resources, required module and number of threads. The following code is an important part of PBS script:

```
# Select 1 full node and 36 cores per node
#PBS -l select=1:ncpus=36
# Exclusive Node Access and scatter to the full set of assigned compute cores
#PBS -l place=scatter:excl
```

2.2 Implementation of the affinity scheduling algorithm

2.2.1 Design

According to the affinity scheduling description, the first step for each thread is to finish its own local set of iterations. After this, the second step is that finished threads start to help a most loaded thread. Thus, affinity scheduling design is mainly divided in two parts.

The design idea of this algorithm is to use shared variables to track remaining iterations of each thread. Threads which have finished choose one most loaded thread to help, which means it is necessary to judge which is the most remaining iterations for each thread in the process of scheduling. After helping a most loaded thread, the key point is to update remaining iteration of most loaded thread in order to skip the iteration which has been executed by finished threads, and then computing the most loaded thread again. However, before helping a most loaded thread, it also needs to make sure that there is a thread at least waiting for to help, which means not all threads have been finished.

In order to make algorithm robust, during design, the author also considers that using 64 threads will cause 61st, 62nd, 63rd threads do not have their own local set of iterations, which means it will help other most loaded threads directly. Thus, these extra threads which do not have own iterations will skip the first step that finished themselves.

2.2.2 Data Structures

During programming, the author uses malloc to create array dynamically in order to adapt to different number of threads.

There are four shared arrays:

```
/* left boundary of each thread*/
leftbound = (int *)malloc(P*sizeof(int));
/* right boundary of each thread*/
rightbound = (int *)malloc(P * sizeof(int));
/* remaining iteration of each thread */
remainingiteration = (int *)malloc(P * sizeof(int));
/*remaining iteration of each thread, but used to track most loaded one  */
mostloadedthread = (int *)malloc(P * sizeof(int));
```

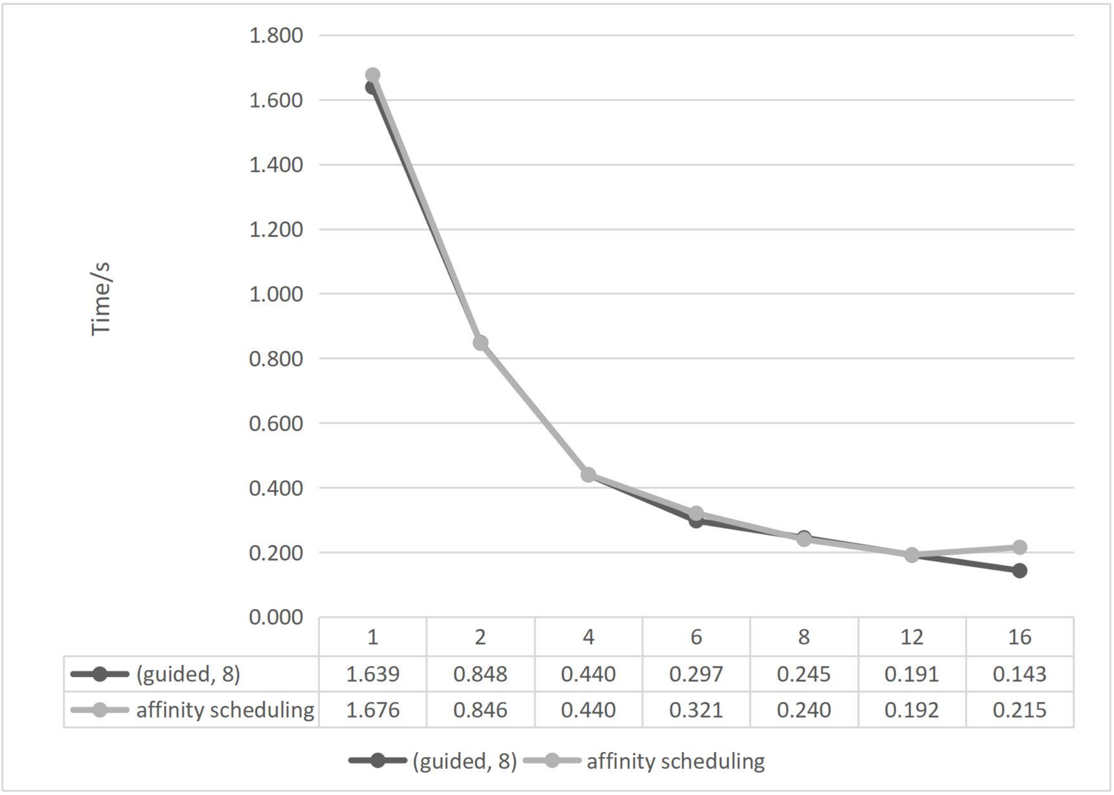
For example, `remainingiteration[3]` means number of remaining iterations in 3rd thread. Because of these shared variables, it is not difficult to track every thread by different thread number.

2.2.3 Synchronised

Before starting loops, it is necessary to use `#pragma omp barrier` to ensure all threads have started successfully. In addition, each thread executing affinity scheduling should be `#pragma omp critical`, which means each thread execute affinity scheduling one by one.

2.3 Data and Result Analysis

2.3.1 Affinity Scheduling Result for loop1



	1	2	4	6	8	12	16
(guided, 8)	1.639	0.848	0.440	0.297	0.245	0.191	0.143
affinity scheduling	1.676	0.846	0.440	0.321	0.240	0.192	0.215

Figure 1: Execution Time of loop1 on (guided, 8) and affinity scheduling

According to Figure 1, the best built-in OpenMP (guided, 8) schedule of coursework part1 and affinity scheduling are extremely similar.

```
void Loop1chunk(int lo, int hi) {
  int i,j;
  for (i=lo; i<hi; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }
}
```

There are two important reasons why affinity scheduling has a similar performance with (guided, 8). According to loop1chunk, it is easy to find that the second loop j always starts from `N-1` to `i = lo`, which means the whole loop1 scatters calculating amount to each thread across iterations from the high to the low, which means it is a very little load unbalance on the whole. In addition, the local set of iterations for them are almost same, they have the similar calculating amount, which means affinity threads and guided threads have the almost same load. On the other hand, they also have the similar scheduling principle but just different the front chunk size and last chunk size. For example, guided scheduling chunks start off roughly `(total iterations/ number of threads)`, and get smaller to 8. Affinity scheduling chunks start from `729/(number of threads)/(number of threads)`, and get smaller to 1. So difference in time would not be very big for this two scheduling. Thus, load and scheduling principle are two key reasons why two scheduling have the chance to get similar performance of loop1.

On the other hand, from the figure 1, affinity scheduling with 16 threads is worse than (guided, 8) because with more threads, each thread executes very few a chunk of iterations per calling affinity scheduling. Take 16 threads for example, each thread local set of iteration is `729/16≈46, 46/16≈3,` which means each thread only can execute 3 iterations per scheduling. And then a chunk of iteration will reduce to 2 and then 1 until no remaining iterations. But for (guided, 8), with 16 threads, it probably has better performance, its chunk of iteration only can reduce to 8, which means can finish local set of iterations more quickly.

2.3.2 Affinity Scheduling Result for loop2

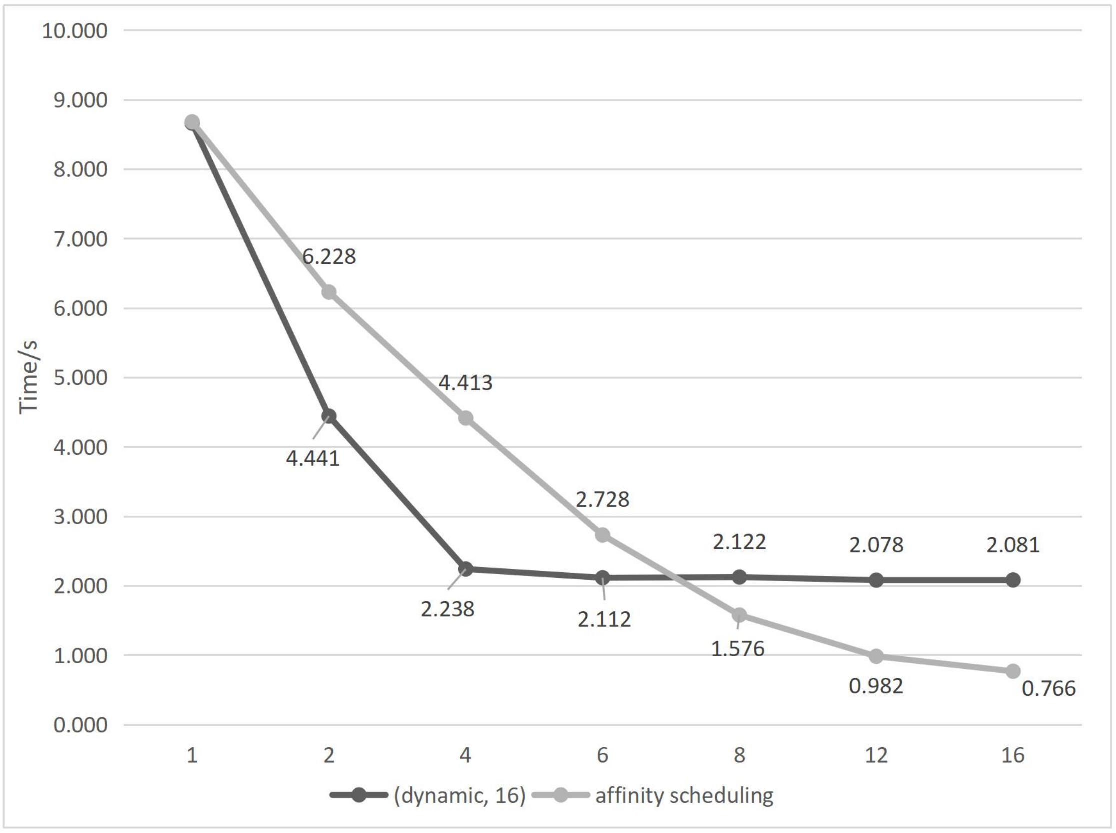


Figure 2: Execution Time of loop2 on (dynamic, 16) and affinity scheduling

According to figure 2, when the number of threads increase 8 or over 8, affinity scheduling is better than (dynamic, 16) schedule. Before 6 threads, (dynamic, 16) are better.

```
  for (i=0; i<N; i++){
    expr =  i%( 3*(i/30) + 1);
    if ( expr == 0) { jmax[i] = N;}
    else { jmax[i] = 1; }
  }

void loop2chunk(int lo, int hi) {
  int i,j,k;
  double rN2;
  rN2 = 1.0 / (double) (N*N);
  for (i=lo; i<hi; i++){
    for (j=0; j < jmax[i]; j++){
      for (k=0; k<j; k++){
     c[i] += (k+1) * log (b[i][j]) * rN2;
      }
    }
  }
```

According to the analysis of loop2, there are a very huge calculating amount at the beginning. This is the reason why guided scheduling with different chunk sizes have almost the same performance, for guided scheduling, the first thread almost can finish the whole calculating amount. (dynamic, 16) is the best because the first 30 iterations from `i = 0` to `i = 29` which include huge calculating amount assigned to two threads, but other threads on a first-come-first-served basis, which means threads which are assigned with the latter chunk size will have less calculating amount so that they can get new chunk size quickly. In other words, finished threads will help other unfinished threads. This is the reason why dynamic scheduling has similar performance with affinity scheduling.

There are some reasons why affinity scheduling is becoming better and better with the increase of number of threads. Dynamic scheduling chunk size is 16 and will not change any more, no matter threads increase or not, it will not improve performance. Because it only just needs two threads to finish the calculating amount of the first 30 iterations, which is the most loaded thread chunk. No matter other threads finish or not, they have to wait for this chunk until its finish. This is why speedup of dynamic scheduling stop on 4 threads. So chunk size is smaller means more threads can share the beginning calculating amount.

However, for affinity scheduling, take 6 threads for example, the first local set of iterations is *729/6 ≈ 122* and the first chunk is *122/6 ≈ 21*. The first thread which is assigned 21 iterations, which is bigger than 16 of dynamic scheduling. Take 8 threads for example, the first local set of iterations is *729/8≈92* and the first chunk is *122/6≈12*, which means the first 30 iterations will be assigned to three threads to compute. Take 16 threads for example, the first local set of iterations is *729/16≈46* and the first chunk is *46/6≈3*.

Dynamic chunk size is 16, but for affinity scheduling, take 6, 8, 16 threads for example, chunk size are from 21, 12 to 3. This is the reason why when threads are 6 or below 6, (dynamic, 16) is better than affinity. And this is also the reason why affinity scheduling is becoming better and better.

On the other hand, for affinity scheduling, it also needs to notice that the latter of the whole loop have the less calculating amount. Threads which are assigned these the latter chunk can finish early so that they can help unfinished threads, which means share the burden calculating amount of the beginning. This is also another positive influence of affinity scheduling for loop2.

2.4 Optimization Suggestions

In the affinity scheduling design, the author finds that there are some points which can be optimized in the future.

```
/* threads call affinity_schedule one by one  */
#pragma omp critical
{
    remainingthreads = affinity_schedule(leftbound, rightbound, &left, &right, myid,
remainingiteration, mostloadedthread);
}
```

The optimization key point is that every time just one thread can be scheduled because of synchronization problems, which means other finished threads have to wait in the queue to be scheduled. In addition, in the author's design, no matter unfinished threads and finished threads always are executed one by one. The optimization method is to separate unfinished threads and finished threads to different functions and use more exact separation for each thread in order to reduce wait time.

On the other hand, the author also tries to use more threads 32, 64, 128 on cirrus.

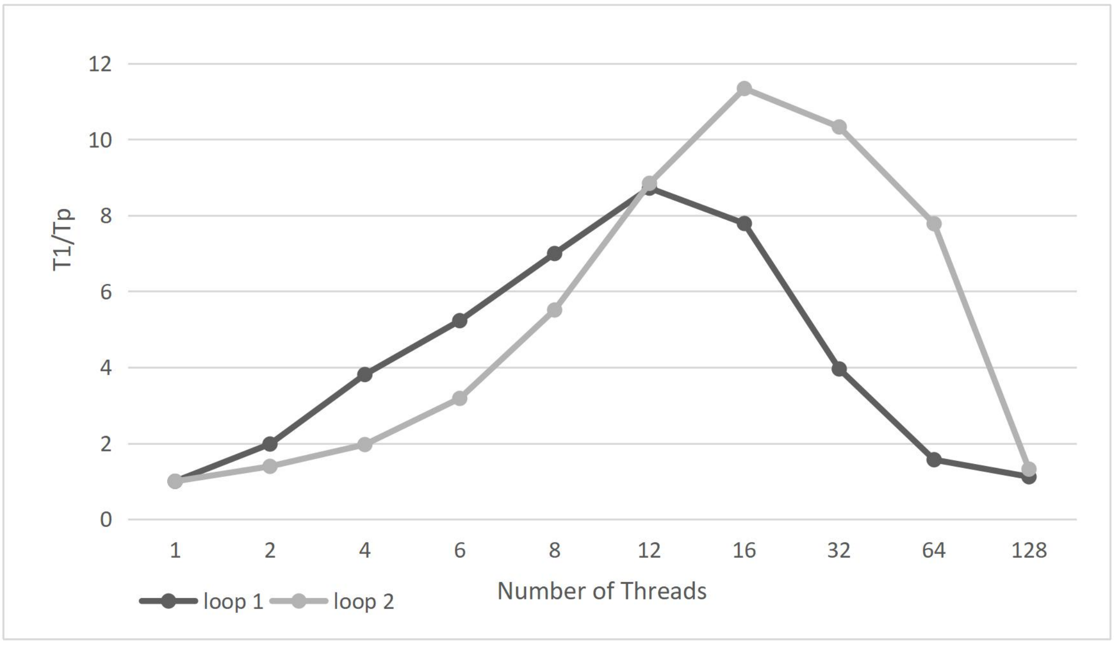


Figure 3: Speedup of loop1 and loop2 on affinity scheduling with different number of threads

According to Figure 3, 12 threads are the best for loop1. 16 threads are the best for loop2.

For both of loops, when the number of threads increases over 32, the chunk size is just 1 per scheduling, so it is too small so that there are very high overheads. In addition, another reason why it is slow is cirrus just only have 36 cores and disable hyper-threading technology.

3. Conclusion

According to these experiments and data analysis, affinity scheduling has the similarity performance for loop1 with the best built-in OpenMP schedule (guided, 8). However, affinity scheduling has a better performance for loop2 compared with the best schedule (dynamic, 16) when number of threads are from 8 to 16. For loop1, 12 threads are the best affinity scheduling options. For loop2, 16 threads are the best affinity scheduling options.

No matter built-in OpenMP schedules or customized schedules, they can specify which loops iterations are executed by which threads. If schedules are used wisely, it can improve programs performance dramatically. Usually, the best schedule option and specified number of threads can be speculated roughly before running programs. However, most importantly, appropriate thread number can improve program performance but it does not mean performance will always improve with threads increase.