MYE023: Homework #2

Due on Monday, May 2, 2017

Vassilios V. Dimakopoulos

George Z. Zachos

May 2, 2017

Contents

1 Exercise #1

1.1 About

This exercise is about the multiplication of integer NxN arrays using the `OpenMP` specification. The serial calculation consists of three (3) nested for-loops and the purpose of this exercise is to parallelize all three, one at a time. The three resulting programs will be executed using both `static` and `dynamic` scheduling policies.

1.2 Experiment details

The calculation consists of N^3 loop iterations (N=1024), while the number of threads used in `parallel` regions is four (4) and chunk size is automatically set to default values.

1.2.1 System Specifications

The experiments were conducted on a Dell OptiPlex 7020:

- CPU: Intel® Core^{TM} i5-4590 CPU @ 3.30GHz (64 bit)
- RAM: 2 DIMMs x4GiB @ 1600MHz DDR3
- Cache line size: 64B (in all levels)
- Cache associativity:
  - L1, L2: 8-way set associative
  - L3: 12-way set associative

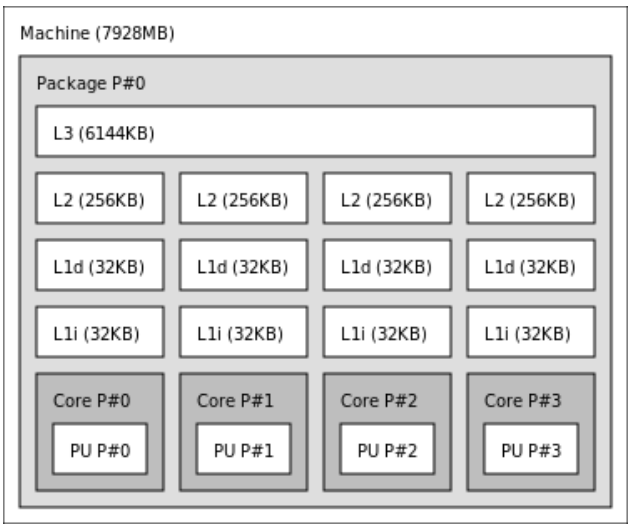

Figure 1: Topology information of a Dell OptiPlex 7020

1.3 Timing Results

In the following table and plot the recorded execution times are displayed. Note that X axis is plotted on a linear scale while Y axis on a (base 10) logarithmic scale.

Timing results of matrix multiplication (Time unit: seconds) Array size: 1024x1024, Number of threads: 4			
	Parallelized loop nesting level		
Scheduling Policy	0	1	2
Static	0.9228565	1.0030725	1.988372
Dynamic	0.87141825	1.049885	28.30894775

Table 1: Timing results of 2D matrix multiplication

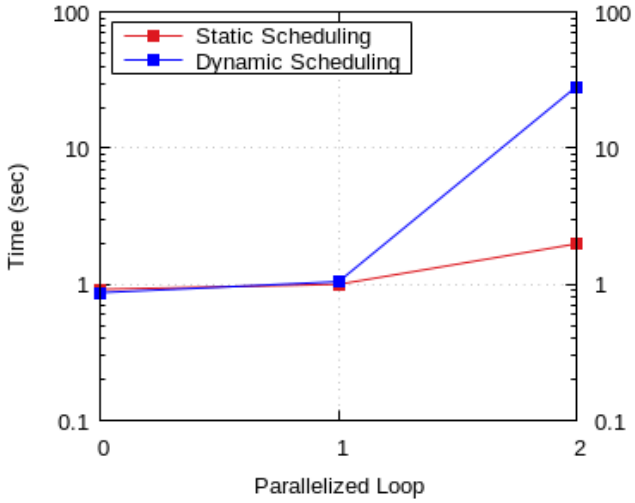

Figure 2: Timing results of 2D matrix multiplication

1.4 Conclusion

Based on the results presented above and given that the average execution time of the serial program is 3,82147025 seconds, we conclude that:

- The best program performance[1] is achieved by parallelizing the outermost for-loop as only one parallel region is encountered. Parallelizing the middle and the innermost loop will cause N and N^2 region encounters respectively and the granularity of the tasks being dispatched to the team threads to decrease. Due to these continuous `parallel` region encounters, execution time is increased as overheads are introduced by thread management (creation, synchronization[2], destruction etc.).
- Both `dynamic` and `static` schedules result in approximately the same execution time, except for the case of parallelizing the innermost for-loop. During `static` schedule, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. In constrast, during `dynamic` schedule, default chunk size equals to one iteration and in total N^3 dispatches take place[3]. For this reason, `dynamic` schedule exponentially increases program time.

[1] About 86% speedup.
[2] There is an implied barrier at the end of every `parallel` region.
[3] N dispatches every time the `parallel` construct is encountered.

2 Exercise #2

2.1 About

This exercise is about the parallelization of `serial_primes()` function which given a positive number N, counts the number of prime numbers and calculates the maximum prime found in $(0, N]$. The purpose of this exercise is to use the `parallel for` construct defined by the `OpenMP` specification and observe how different scheduling policies affect execution time.

2.2 Experiment details

During this experiment, the schedules used are `static`, `dynamic` and `guided` while chunk size takes value in $\{1, 10, 10^2, 10^3, 10^4, 10^5, 10^6\}$. The number of threads used in `parallel` regions is four (4).

2.2.1 System Specifications

The experiments were conducted again on a Dell OptiPlex 7020.

2.3 Timing Results

In the following table and plot the recorded execution times are displayed. Note that X axis is plotted on a (base 10) logarithmic scale while Y axis on a linear scale.

Timing results of primes calculation (Time unit: seconds) Number of threads: 4							
	Chunk size						
Scheduling Policy	1	10	10^2	10^3	10^4	10^5	10^6
Static	3.76737175	3.74108125	3.73942225	3.738404	3.747804	3.884757	5.433113
Dynamic	3.75348175	3.73795675	3.7347005	3.736425	3.748476	3.88446	5.431626
Guided	3.736479	3.730526	3.7326305	3.735151	3.749482	3.914775	5.056508

Table 2: Timing results of primes calculation

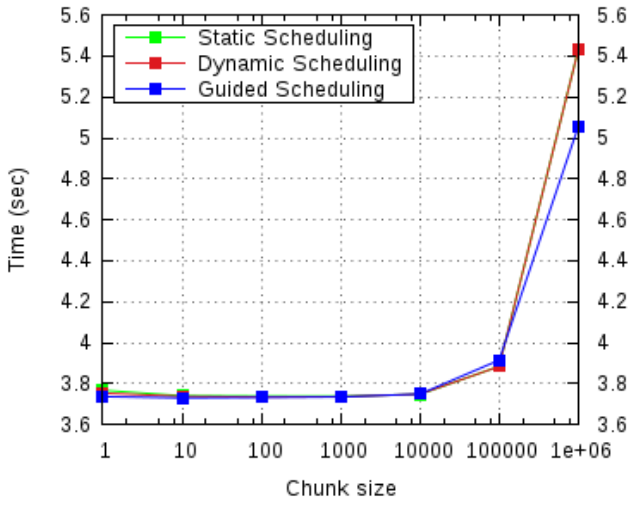

Figure 3: Timing results of primes calculation

2.4 Conclusion

Based on the results presented above and given that the average execution time of the serial program is 13.02629725 seconds, we conclude that:

- All schedules for a given chunk size result in approximately the same execution time. This is due to the negligible workload[4] under which the four system processors were while the experiments were conducted.
- Parallel program efficiency is unfolded for chunk size between 10 and 10^3.
- Fine-grained tasks (chunk size: 1) result to a slight increase of program time as overheads are introduced due to the continuous chunk dispatch.
- More coarse-grained tasks (chunk size $> 10^3$) cause a significant increase of program time. This happens because of the nature of the calculation conducted. The time spent in the `while` loop of the calculation is increased as loop counter variable `i` is increased[5]. For this reason, dividing iteration space in large chunks, will lead to a lopsided work share between threads and the total execution time will increase.

3 Exercise #3

3.1 About

This exercise is about the multiplication of integer NxN arrays using `OpenMP` tasks and checkerboard partitioning. During checkerboard partitioning, the initial matrix is divided into submatrices of size SxS. The purpose of this exercise is to assign the calculation of each submatrix to an `OpenMP` task and observe how altering the number of tasks will affect execution time.

3.2 Experiment details

During this experiment, the number of `OpenMP` tasks takes value in {16, 256, 1024}, initial array size is 1024x1024 and the number of threads used in `parallel` regions is four (4).

3.2.1 System Specifications

The experiments were conducted again on a Dell OptiPlex 7020.

3.3 Timing Results

In the following table and plot the recorded execution times are displayed. Note that X axis is plotted on a (base 2) logarithmic scale while Y axis on a linear scale.

[4] Processor on idle state with some lightweight system services running every once in a while
[5] In general terms. Not for every single value of `i`

Timing results of matrix multiplication (Time unit: seconds) Array size: 1024x1024					
# of `OpenMP` Tasks	1st run	2nd run	3rd run	4th run	Average time
16	0.798303	0.790874	1.156564	0.901301	0.9117605
256	1.003373	0.993386	0.863506	0.974971	0.958809
1024	1.191109	1.036035	1.138259	0.956815	1.0805545

Table 3: Timing results of 2D matrix multiplication using `OpenMP` tasks and checkerboard partitioning

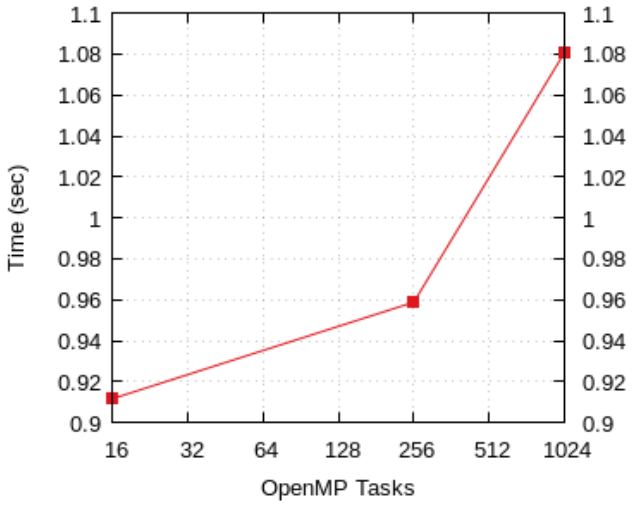

Figure 4: Timing results of 2D matrix multiplication using `OpenMP` tasks and checkerboard partitioning

3.4 Conclusion

Based on the results presented above and given that the average execution time of the serial program is 3,82147025 seconds, we conclude that:

- Program performance is decreased as the number of `OpenMP` tasks is increased. This happens due to the overheads related to task management (creation, task switching etc.).
- Even though overheads related to task management are introduced, execution time remains significantly lower compared to the one of the serial program. This is because the significant synchronization overheads[6] do not increase relatively to the task number. Actually no syncrhonization takes place at all except for the implicit barrier at the end of the `parallel` region.

[6] As observed in Exercise #1 where the increase of `parallel` region encounters results in a linear increase of the number of implicit barriers.