MYE023: Homework #1

Due on Monday, April 7, 2017

Vassilios V. Dimakopoulos

George Z. Zachos

March 30, 2017

Contents

1 Exercise #1

1.1 About

This exercise is about the calculation of the mathematical constant π using `POSIX` threads and dynamic scheduling. During dynamic scheduling the parallelizable loops are divided into chunks of iterations (tasks) and are dispatched to the threads available to the runtime system for execution. The dispatch takes place in respect to the current processor workload where each thread executes and as a result load balancing is achieved. In case chunk size is one (1) iteration, we refer to this technique as self-scheduling. The purpose of this exercise is to time the calculation of π and observe how altering the number of threads will affect execution time for a given chunk size.

1.2 Experiment details

The calculation consists of $5 * 10^8$ loop iterations, while thread number takes value in $\{1, 4, 16\}$ and chunk size in $\{1, 10, 10^2, 10^3, 10^4, 10^5\}$.

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

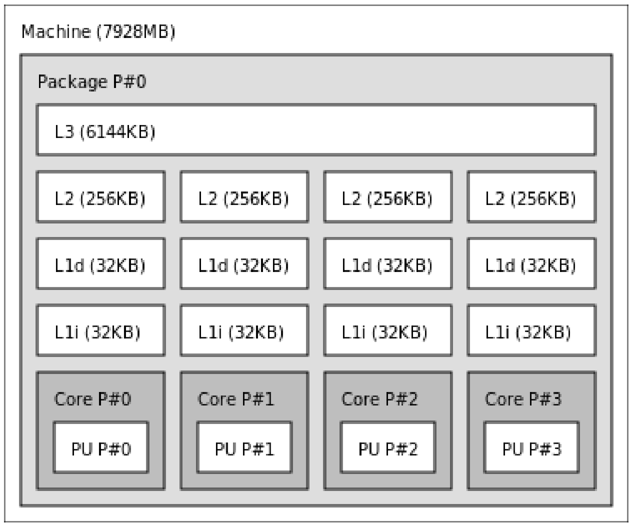

Figure 1: Topology information of a Dell OptiPlex 7020

1.3 Timing Results

In the following tables and plots 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.

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
1	1	18.451202	18.443870	18.444319	18.441278	18.44516725
1	4	98.559317	98.393137	99.515415	98.189223	98.664273
1	16	95.482310	95.205719	95.275233	95.197046	95.290077

Table 1: Timing results of π calculation using chunk size = 1 iteration

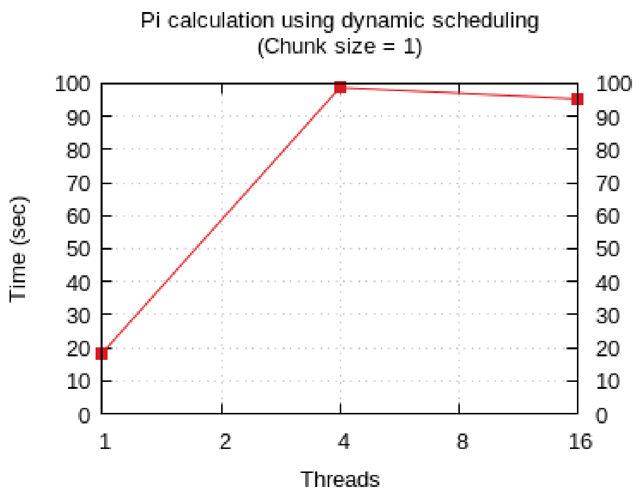

Figure 2: Timing results of π calculation using chunk size = 1 iteration

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
10	1	6.505206	6.510850	6.507051	6.511070	6.50854425
10	4	10.843631	10.728116	10.715714	10.832101	10.7798905
10	16	10.829372	10.820372	10.842818	10.748566	10.810282

Table 2: Timing results of π calculation using chunk size = 10 iteration

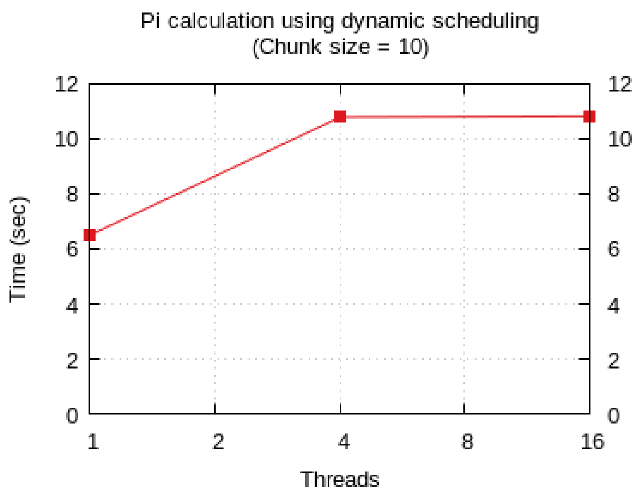

Figure 3: Timing results of π calculation using chunk size = 10 iterations

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
100	1	6.275921	6.279012	7.893470	6.281098	6.68237525
100	4	2.428611	2.464799	2.463414	2.425332	2.445539
100	16	2.425184	2.459710	2.432488	2.458897	2.44406975

Table 3: Timing results of π calculation using chunk size = 100 iteration

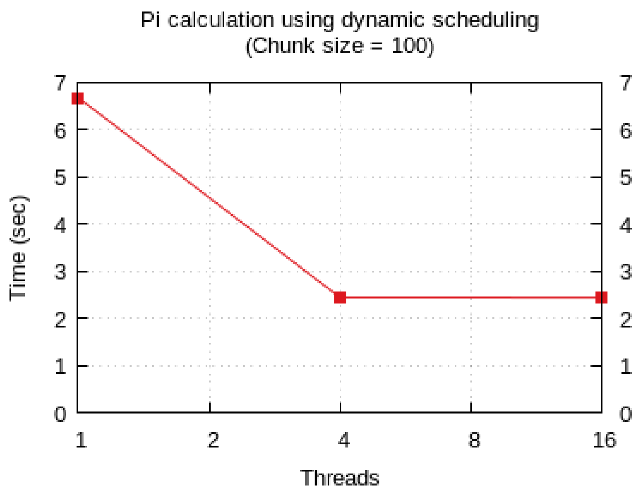

Figure 4: Timing results of π calculation using chunk size = 100 iterations

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
1000	1	6.248489	6.254362	6.254913	6.251492	6.252314
1000	4	1.733363	1.735331	1.732440	1.734742	1.733969
1000	16	1.731060	1.726669	1.730891	1.732937	1.73038925

Table 4: Timing results of π calculation using chunk size = 1000 iteration

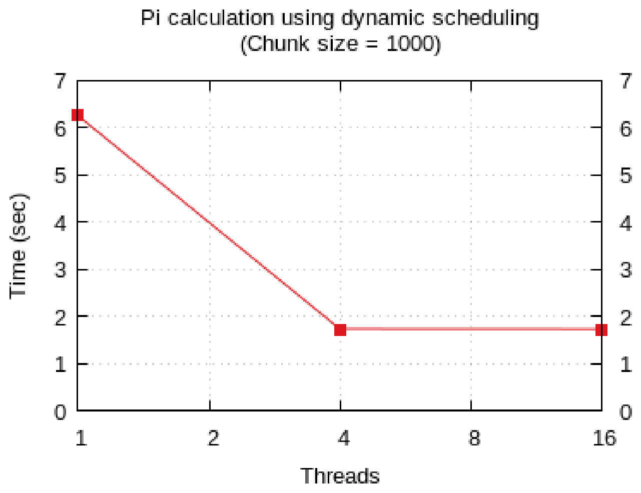

Figure 5: Timing results of π calculation using chunk size = 1000 iterations

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
10000	1	6.244337	6.252478	6.250002	6.252064	6.24972025
10000	4	1.664214	1.659239	1.660260	1.659163	1.660719
10000	16	1.660762	1.664066	1.661164	1.658869	1.66121525

Table 5: Timing results of π calculation using chunk size = 10000 iteration

Timing results of π calculation (Time unit: seconds)						
Chunk Size	# of threads	1st run	2nd run	3rd run	4th run	Average time
100000	1	6.237799	6.234975	6.244593	6.235083	6.2381125
100000	4	1.661888	1.658459	1.667569	1.651900	1.659954
100000	16	1.653965	1.652250	1.651300	1.651015	1.6521325

Table 6: Timing results of π calculation using chunk size = 100000 iteration

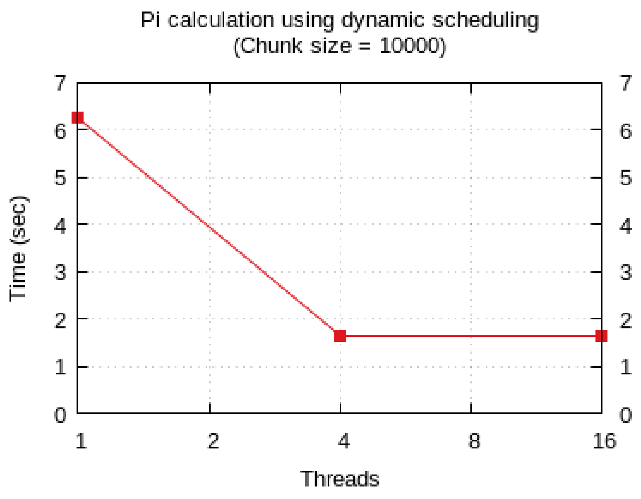

Figure 6: Timing results of π calculation using chunk size = 10000 iterations

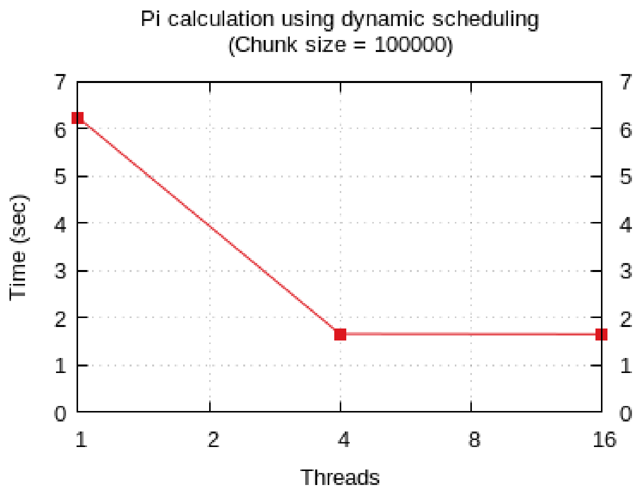

Figure 7: Timing results of π calculation using chunk size = 100000 iterations

1.4 Conclusion

Based on the results presented above and given that the average execution time of the serial program is 6,245896 seconds, we conclude that:

- Program performance is increased until oversubscription appears. Even though we expect overheads to be introduced due to time slicing (e.g. context switching, cache pollution), the execution time becomes approximately constant after the number of threads exceeds the number of the processors available (4). This happens because switching between threads is less resource-intensive than switching between processes.
- Self-scheduling leads to execution times multiple times greater than the one of the serial program and in case of multithreaded calculation, dozens of times greater. The chunk size of one (1) iteration is a fine-grained task something that results in threads constantly racing to acquire the same mutex lock. As the number of threads is increased, race overhead is increased too. Moreover, the granularity of self-scheduling leads to more function calls taking place, something that adds up to the existing overheads.
- A single thread executing tasks with chunk size ≥ 10 requires almost the same time as the serial program.
- Multiple threads and tasks with chunk size $\ll 10^2$ result in higher execution times compared to the serial program. The reasons for these overheads are the same as in self-scheduling (fine-grained parallelism).
- Parallel program efficiency is unfolded for chunk size $\geq 10^2$ but hits a bottleneck for more coarse-grained tasks (chunk size $\geq 10^4$ iterations in this case).

2 Exercise #2

2.1 About

This exercise is about the multiplication of integer NxN arrays using `POSIX` threads and static scheduling. During static scheduling the parallelizable loops are evenly (when possible) divided into chunks of iterations (tasks) and are dispatched to the threads available to the runtime system for execution. Due to this even distribution of iterations and in contrast to dynamic scheduling, a thread executing on a processor under heavy workload will increase the total execution time. The purpose of this exercise is to parallelize only the outermost for-loop of the serial calculation, time the matrix multiplication and observe how altering the number of threads will affect execution time.

2.2 Implementation details

If T is the number of threads, and N is the array dimension, then the outermost for-loop of the serial program consists of N iterations, that should be divided into T chunks. When T divides N evenly, the exact chunk size is N/T. In the opposite case, $S = N \bmod T$ threads will be assigned $\lfloor N/T \rfloor + 1$ iterations and $T - S$ threads will be assigned $\lfloor N/T \rfloor$ iterations. We are going to refer to S as the number of special threads because these threads execute one more iteration than the rest. This policy manages to avoid a lopsided distribution of iterations to threads as it increases workload by only a single iteration [1].

[1] This additional iteration may add significant delays in coarse-grained tasks

2.3 Experiment details

During this experiment, thread number takes value in {1, 2, 4, 8, 12, 16} and array size is 1024x1024.

2.3.1 System Specifications

The experiments were conducted once again on a Dell OptiPlex 7020.

2.4 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.

Timing results of matrix multiplication (Time unit: seconds) Array size: 1024x1024					
# of threads	1st run	2nd run	3rd run	4th run	Average time
1	5.057095	4.319662	3.157559	5.297035	4.45783775
2	2.648351	2.545161	1.940197	2.043663	2.294343
4	0.918773	0.800195	1.504583	1.504583	1.07884625
8	0.971947	1.428921	1.224236	0.990774	1.1539695
12	0.965793	0.986273	0.981548	1.130868	1.0161205
16	1.208460	1.241845	1.015431	0.974093	1.10995725

Table 7: Timing results of 2D matrix multiplication

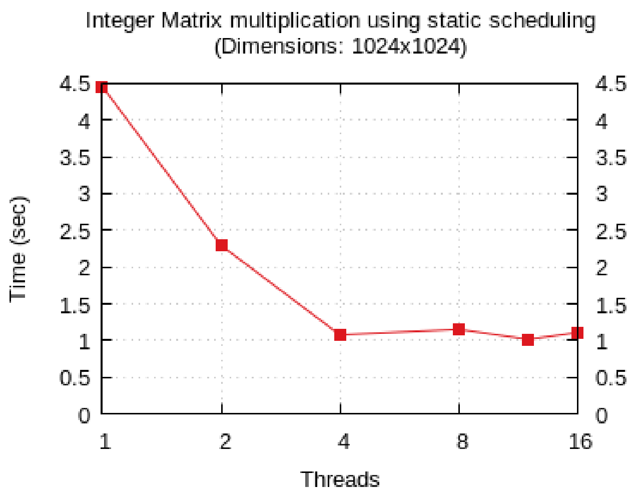

Figure 8: Timing results of 2D matrix multiplication

2.5 Conclusion

Based on the results presented above and given that the average execution time of the serial program is 4,297561 seconds, we conclude that:

- Program performance is approximately doubled as the number of threads is increased, until oversubscription bottleneck is hit and execution time becomes almost constant [2].
- Execution time of the parallel program when a single thread is used to perform the calculation is a bit higher than the time of the serial program. Moreover, we observed that execution time in general varies from time to time. Several reasons such as context switching, thread affinity, cache misses & cache pollution justify the existence of these overheads and consequently this behavior.

[2] This is also the conclusion we came to on Exercise #1