CS 4552
Assignment - False Sharing

180449H
K.P.D.T. Pathirana
Computer Science and Engineering

Table of Content

1. Experiment Methodology

- The value of Pi was calculated using Gregory-Leibniz Series in three versions as follows.
    - V1: sequential program
    - V2: parallel program with false sharing
    - V3: parallel program without false sharing
- The parallel programs were implemented using the ***Pthreads*** library.
- Number of Terms: **1,000,000,000**
- Range of Number of Threads: **1, 2, 4, 8, 16, 32, 64, 128**
- For each version and each number of threads, the elapsed time was measured through 10 iterations, and the average time was taken.
- The relative time difference between V2 and V3 was calculated for each number of threads as follows.
    - $Time\ Elapsed\ for\ V2\ -\ Time\ Elapsed\ for\ V3$
- Used Machines:
    - ***University Server***: Lenovo-SR650V2 (96 cores, Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz, 125 GB RAM, Ubuntu Linux)
    - ***Local Machine***: Predator Helios 300 (8 cores, 11th gen Intel® Core™ i7-11800H @ 2.3GHz, 16 GB RAM, Windows 11 Home)

2. False Sharing - Implementation

False sharing refers to a performance issue that can occur in multi-threaded programs when multiple threads access different variables that happen to reside on the same cache line. A cache line is a unit of data storage in a CPU cache, and it typically consists of multiple bytes or words.

The calculation of Pi using the Gregory-Leibniz Series requires a summation of a set of terms. These terms can be divided into a set of groups and for each group, the summation can be done in a separate thread. Thus, it parallelizes the Gregory-Leibniz Series.

In order to introduce false sharing, a global array was defined where each element of the array will represent the summation of the group of terms given to a specific thread. Finally, after all the threads are completed, the summation of the array is taken as the value of Pi.

The code segment with false sharing is as follows (V2):

```
sum[id] = 0.0;
  int i;

  for (i = id; i < terms; i += thread_count) {
      int sign = (i % 2 == 0) ? 1 : -1;
      double term = sign * (4.0 / (2 * i + 1));
      sum[id] += term;
  }
```

Here, each iteration of a thread will access and update the sum array introducing false sharing. Even though each thread accesses an element unique to that thread, the cache is invalidated for every iteration from a different thread since all the elements in the array are in the same cache line.

The code segment without false sharing is as follows (V3):

```
int i;
  double pi = 0.0;

  for (i = id; i < terms; i += thread_count) {
      int sign = (i % 2 == 0) ? 1 : -1;
      double term = sign * (4.0 / (2 * i + 1));
      pi += term;
  }

  sum[id] = pi;
```

Here, introducing a local variable (pi) to store the local summation and updating the array at the end of the loop removes false sharing from the program.

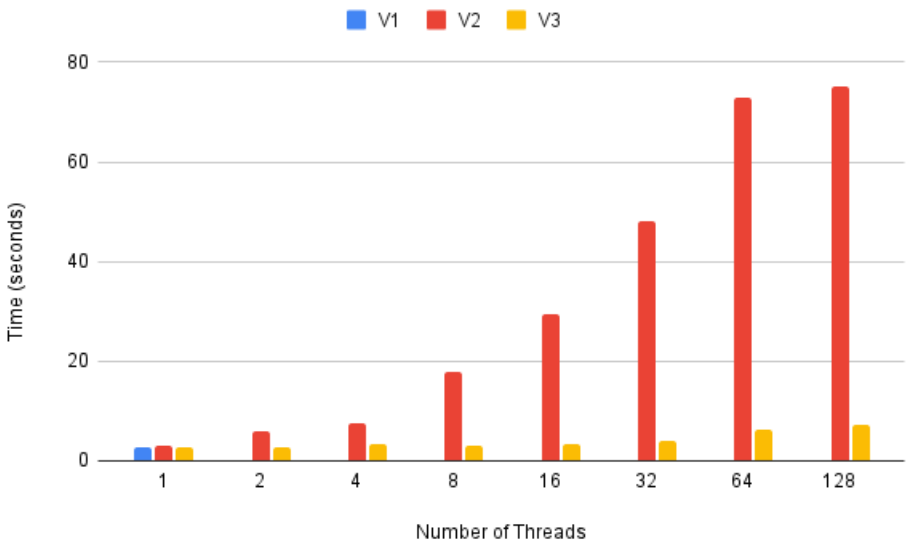
3. Experiment Results and Analysis

3.1 University Server

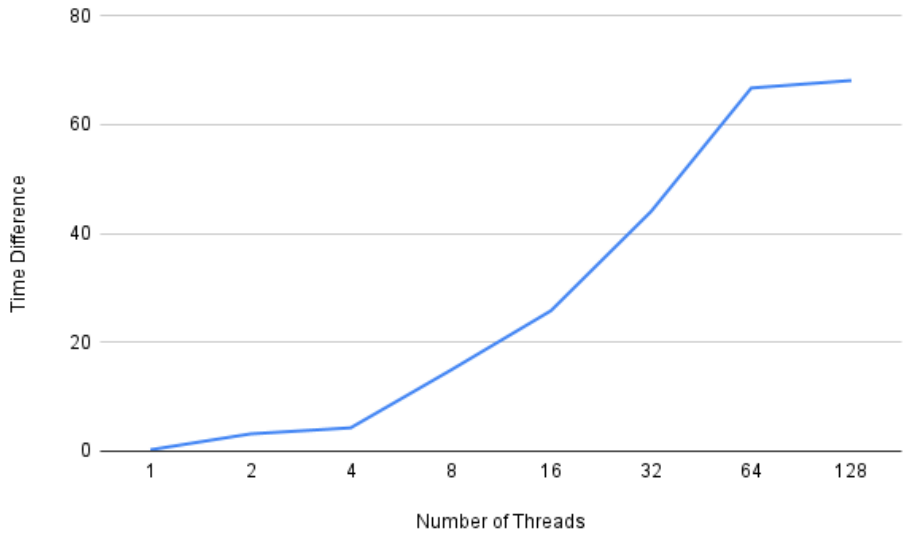
3.1.1 Test Results

	Number of Threads							
Version	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
V1	2.729733	N/A	N/A	N/A	N/A	N/A	N/A	N/A
V2	2.9875	5.9752	7.5300	17.9378	29.2763	48.0561	72.9042	75.2941
V3	2.7346	2.802	3.2457	3.0595	3.4845	4.0352	6.2209	7.2319

3.1.2 Plotted Test Results



3.1.3 Time Difference between V2 and V3 against Number of Threads



3.1.4 Analysis

According to the figure in Section 3.1.2, it is clear that with false sharing, as the number of threads increases, the execution time also increases rapidly. This indicates that the performance is affected by false sharing, leading to larger execution times.

Without false sharing, the execution time remains relatively stable or slightly decreases as the number of threads increases. This suggests that without false sharing, the performance scales better with increasing thread count.

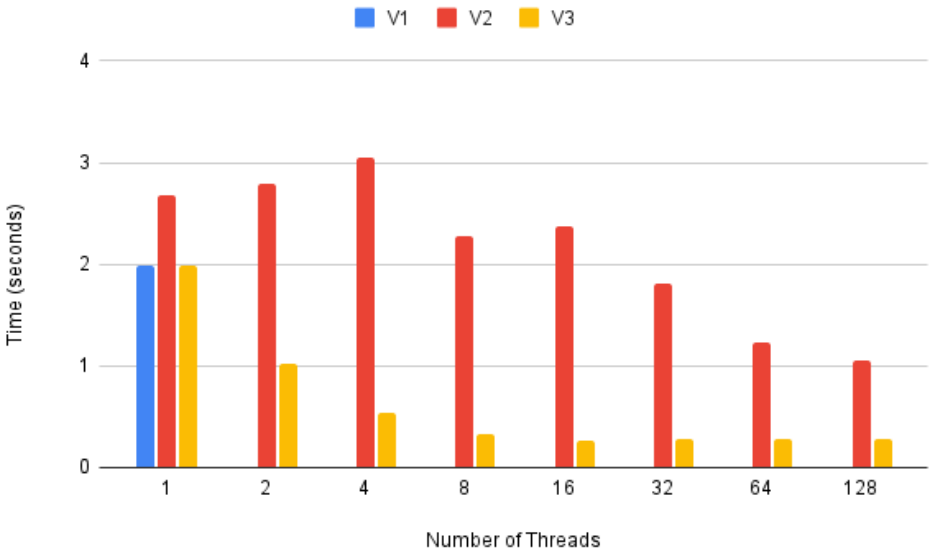
Furthermore, the figure in Section 3.1.3 further clarifies the above observation where the time gap between V2 (with false sharing) and V3 (without false sharing) is always positive and increases as the thread count goes up. Therefore, for any number of threads, the time taken for V2 (with false sharing) is always greater than that of V3 (without false sharing), resulting in far greater performance when false sharing is avoided.

3.2 Local Machine

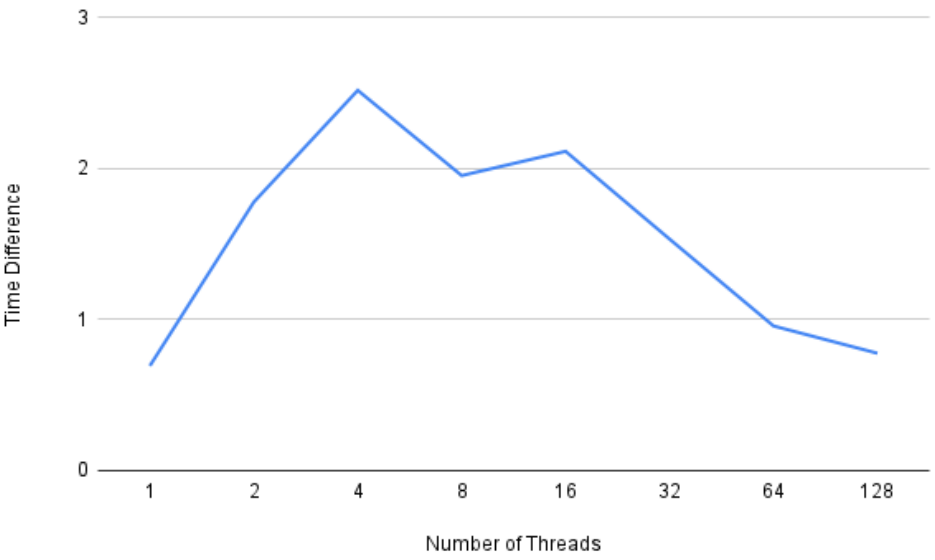
3.2.1 Test Results

	Number of Threads							
Version	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
V1	1.98045	N/A	N/A	N/A	N/A	N/A	N/A	N/A
V2	2.6766	2.7912	3.0462	2.2819	2.3748	1.806	1.2302	1.0601
V3	1.9833	1.0144	0.5304	0.3301	0.2627	0.2742	0.2732	0.2837

3.2.2 Plotted Test Results



3.2.3 Time Difference between V2 and V3 against Number of Threads



3.2.4 Analysis

The above results are taken for a C program that was compiled and run in a Windows environment. According to the figure in Section 3.2.2, with false sharing, the execution time generally decreases as the number of threads increases. However, there is a slight increase in time when moving from 8 to 16 threads. This suggests that false sharing negatively affects performance, but parallelization still provides some improvement. The cause for this performance improvement should be the C compilation techniques used in the Windows environment by MinGW even though the -O0 (no optimization) tag was used.

Without false sharing, the execution time significantly decreases as the number of threads increases. This indicates that parallelization without false sharing leads to efficient utilization of resources and better performance. And it can be clearly seen that the execution times of V3 are greatly smaller than those of V2.

Furthermore, the figure in Section 3.2.3 further clarifies the above observation where the time gap between V2 (with false sharing) and V3 (without false sharing) is always positive. But the difference tends to decrease after 4 threads and thus explaining the optimization added by MinGW compilation further.

4. Summary

In summary, ***avoiding false sharing in the implementation seems to improve the overall performance***, as it results in shorter execution times compared to the implementation with false sharing in multithreaded environments. On a side note, it was observed that the MinGW compilation in Windows seems to add some optimization code to the code while compiling.