Overview

I began this assignment by writing a C-file for the assignment. I then implemented the methods GetProcessMemory()**,** &parseLine() as well as a struct processMem\_t to hold the virtual and physical memory. From here **I** added the method, DoWork() to implement the given code. Finally, I set up Main() to handle command line arguments so I could pass in the amount of threads, the number of iterations, and keep track of which trial I was working with. All input was used in DoWork() and changed depending on the input of the command-line arguments. I then copied the original file (parallel\_origin.c)twice and renamed the files parallel\_omp.c&parallel\_pthreads.c respectively.

Using OMP was straightforward. I included the omp libraries and used commands like#pragma omp parallel(which tells the program to run the following behavior in parallel), and #pragma ompprivate() to declare which variables were local to a thread. To use a certain/set amount of threads, I had to use omp\_set\_num\_threads(THREAD COUNT). The command, #pragma omp critical was used to “lock” access to a global variable for assignment/alteration by a part of the current thread. This ensured there wouldn’t be any race conditions. Speed-ups for this will be discussed in the **Tests & Code Scenario’s: Time & Temporality** section.

Utilization of the pthreads libraries I found to be a bit more frustrating. As with OMP, I had to give pthreads the number of threads to work with. This was stored in an array with type pthread. I also had to use a thread detached attribute, which is basically a way of ensuring that once a created thread has finished its job, its attributes and thread ID may be recycled. I looped over the behavior of creating threads using pthread\_create() passing in the number of threads, the detached attribute, a behavior for the thread (DoWork()), and a void pointer to an int. From here, I added code for locking access to any data or point in memory that would be global, or outside of a thread. The only bit of code in my DoWork() method was accessing “global\_sum” to add the local sum of the thread. To do this, I used a mutually exclusive signifier (“mutex\_sum”) to block access to global sum from other threads. This ensured there would be no race conditions. Finally, I had to destroy or release the thread attribute I had created for each thread I was using. I used pthread\_attr\_destory(created thread attribute) to accomplish this task. I was then able to process and print the memory and time from the program. Speed-ups for this will be discussed in the **Test & Code Scenario’s: Time & Temporality** section.

From here, I wrote the shell scripts to automate the compiling, and test cases of my programs. In the shell script, parallel\_origin.sh**,** I wrote a bash script to execute the object file of parallel\_origin.c. I did this for each of the other files as well. I also wrote mass\_sbatch.sh to compile & run each of the respective shell files, entering certain cases as command-line arguments into each file. For each file ran, I passed in a thread count from 2, 4, 8, & 16. I also passed in the number of iterations, which ranged from 100,000,000 to 400,000,000 as well as a counter to keep track of which case I was on that ranged (discretely) from 1 to 10 (inclusive). After I compiled my shell files, I ran mass\_sbatch.sh and submitted 360 jobs to Beocat, constrained to only work with the Elf nodes. I used the command “grep DATA \*.out > hw2\_data.csv” to compile my data to a csv file and computed my averages, created my graphs, and found what was efficient, and what wasn’t.

Tests & Code Scenario’s: *Introduction*

To see how my altered code behaved, I wanted to take an average of at least 10 jobs for each number of iterations. The range of iterations {100,000,000, 200,000,000, 300,000,000, 400,000,000} was repeated for each thread 10 times for each iteration. My original code only ran with 4 threads, but the number of iterations was changed and ran 10 times each. The original code was running a total of 40 times, with each number of iterations being ran 10 times. I did this, so I could attain a decent average and account for any outliers. Parallel\_omp.c & Parallel\_pthreads.c both were ran 160 times. 40 times for each thread, with 10 of the 40 tests being ran with 100,000,000 iterations and 2 threads, 10 of the 40 tests running with 200,000,000 with 2 threads, and so on until the last iteration was ran with 400,000,000 iterations, 16 threads, and 10 tests.

Tests & Code Scenario’s: Time & Temporality

In order to test the efficiency of my altered, parallel programs (OMP & Pthreads), I needed to display the data in a way that could be easily read. The following tables depict the averages of OMP & Pthreads time in milliseconds, as well as the original programs time with a set number of threads.

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iterations: 100,000,000 | | | |
| Thread Count | Original Averages | OMP Averages | Pthreads Averages |
| 2 | 1419.26 | 726.38 (x1.9) | 756.67 (x1.8) |
| 4 | 1419.26 | 398.59 (x3.6) | 394.07 (x3.6) |
| 8 | 1419.26 | 405.04 (x3.5) | 392.33 (x3.6) |
| 16 | 1419.26 | 397.62 (x3.6) | 387.94 (x3.6) |

**Table 1:** The number of threads is the same for all values in the original values column;

Each average is set against the time (ms) of the original with 4 threads;  
 The average speed up is to the right of each average time for OMP & Pthread  
 in parenthesis. Both were fairly similar in terms of speed up.

This data may also be represented in a graphical fashion. See **Figure 1** below for a visual representation of the average time (ms) by the number of threads in use. Keep in mind that for the original program (the blue bar), there were only 4 threads in use and it ran serial.

**Figure 1:** Keep in mind, the number of threads used for the original averages is 4 for all representations   
 and tests ran. It ran serial in comparison to the OMP & Pthreads programs.

There was an average speed up of about 3.6, or 360% overall for these tests. As **Figure 1** shows, we can actually see that OMP & Pthreads brought about similar results when ran on different threads and at times only varying by a few milliseconds. Overall, the OMP & Pthreads programs were equivalent in their speed-ups, with an overall speed-up average of 3.15, or 315%. The number of iterations on this run was 100,000,000, so next we will look at 4x the iteration size to see if my speed-ups are consistent. **Table 2** depicts the average speed-ups of OMP & Pthreads against the original program with 4 threads.

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Iterations: 400,000,000 | | | |
| Thread Count | Original Averages | OMP Averages | Pthreads Averages |
| 2 | 5679.78 | 3036.1 (x1.9) | 2960.33 (x1.9) |
| 4 | 5679.78 | 1560.49 (x3.6) | 1550.73 (x3.7) |
| 8 | 5679.78 | 1585.17 (x3.6) | 1558.25 (x3.6) |
| 16 | 5679.78 | 1568.82 (3.6) | 1498.86 (x3.8) |

**Table 2:** The number of threads for the original averages are 4 for every test;  
 The thread counts change for OMP and Pthreads in a discrete manner set of   
 {2, 4, 8, 16}.

As we can see, the average speed-ups remained consistent, even with an increased number of iterations. The best speed-up came from my Pthreads program, yielding a speed-up by a factor of 3.8, or 380%. The average speed-up for Pthreads was also greater than the average speed-up of OMP, where OMP’s average speed-up was a factor 3.18, and Pthreads average speed-up was a factor of 3.25. **Figure 2** gives a graphical representation of the speed-ups in relation to time, the number of threads, and the number of iterations.

**Figure 2**: Depicted are the average times of OMP & Pthreads in milliseconds over the number of   
 threads in use. The original averages has a fixed thread count of 4 for all tests.

Tests & Code Scenario’s: Virtual & Physical Memory

In order to see how my programs were running in relation to Virtual & Physical Memory, I needed to implement the GetProcessMemory() method from Dr. Andresen. I was able to record the process memory of both virtual & physical memory by printing it to an output file after each job had finished. After accumulating and organizing my data, I found that the averages for virtual and physical memory remained fairly constant and only change with the number of threads I was implementing. **Table 3** depicts this.

|  |  |  |  |
| --- | --- | --- | --- |
| THREAD COUNT X VIRTUAL MEMORY | | | |
| Thread Count | Original Average | OMP Average | Pthreads Average |
| 2 | 4344 | 16904 | 90572 |
| 4 | 4344 | 33296 | 106964 |
| 8 | 4344 | 66080 | 106964 |
| 16 | 4344 | 131648 | 106964 |

**Table 3**: Depicted is the average of virtual memory (in KB) for each respective program.  
 The average virtual memory used for pthreads remained constant after the   
 introduction of 4 threads. The average for OMP, however, roughly doubles   
 with the doubling of threads.

In respect to average amount of memory used, OMP beat Pthreads having an average of 61,982 KB used, while Pthreads had an average of 102866 KB used. This is depicted in **Figure 3.**

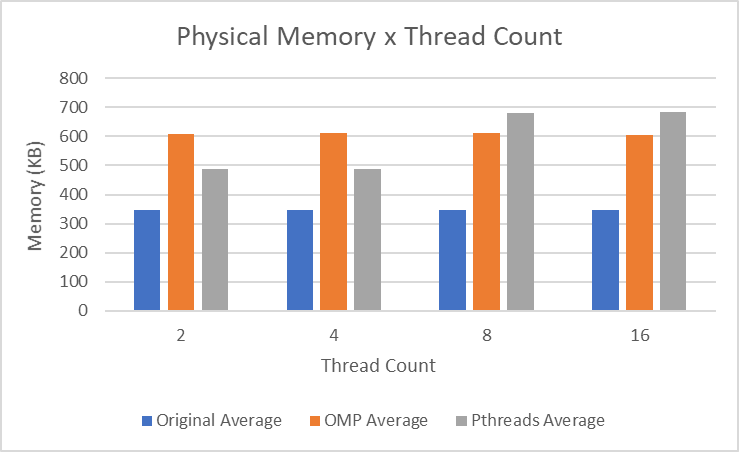
**Figure 3:** Depicted is the average usage of Virtual Memory by each program.  
 We can see that the average usage for OMP is better until 16 threads   
 are used. Overall, OMP had a better average, almost being twice as   
 efficient as Pthreads in respect to Virtual Memory.

As for physical memory, Pthreads had an average usage of 584.95 KB with OMP having an average usage of 607.9 KB. Pthreads was only slightly better on average memory-wise in respect to physical memory. **Table 4 & Figure 4** depict the averages of both OMP & Pthreads against the number of threads by the number of KB used.

|  |  |  |  |
| --- | --- | --- | --- |
| THREAD COUNT X PHYSICAL MEMORY | | | |
| Thread Count | Original Average | OMP Average | Pthreads Average |
| 2 | 348 | 607 | 487.7 |
| 4 | 348 | 610.3 | 488 |
| 8 | 348 | 610.4 | 680.2 |
| 16 | 348 | 604 | 683.9 |

**Table 4:** Depicted is the average number of KB (physical memory) used by each  
 program in respect to the number of threads. For the original average,   
 the number of threads is equal to 4.

I’m honestly unsure as to why Pthreads used memory more efficiently when dealing with 2 and 4 threads. Pthreads also used memory *inefficiently*when using a thread count of 8 and 16. Overall, OMP was more consistent with its memory usage. This is also depicted in **Figure 4** below.

****  
**Figure 4:** Depicted is the average physical memory usage in KB for each program.  
 Overall, OMP had a more consistent usage, while Pthreads had, on average  
 a more efficient usage.

Conclusion

In conclusion, it appears that it could come down to user preference as to whether open mp (OMP) or Pthreads is used. On average, pthreads was more efficient temporally speaking, but lacked consistency memory wise. If both are used correctly, no race conditions will occur in the code.

\* The environment I ran my code on was the Elf nodes on the Beocat cluster. I only utilized multiple threads on different cores, and did not utilize multiple nodes.