COMPE571 PA 1 Report

Skye Russ

1 Programming Implementation

1.1 Baseline Code Implementation

The baseline implementation of the workload has a parameterized passing of the target number followed by a for loop that increments from 1 to the target number adding each integer to a sum variable. This sum is then printed to the console.

Timing is accomplished using the <time.h> header file. The start and ending time of the workload are stored in a "timespec" struct variable. Then the difference between the two times is calculated and printed to the console.

This program is 30 lines long (including comments) with an executable size of 16192 bytes. The workload is accomplished in 6 lines.

1.2 Multithreading Code Implementation

The multithreading implementation of the workload has a parameterized passing of the target number and number of threads. The program then creates the correct number of threads to run a helper "sum" function. The "sum" function takes a 3-element array of longs as an input (cast from a void pointer). The first element is the lower bound, the second element is the upper bound, and the third element stores the result. The parent program stores an array of these arrays that have an index for each thread. Because the memory space is shared, each thread directly accesses their portion of this array. While this solution is not the most memory efficient, it does prevent any race conditions to access the results variable and allows each thread to function within its own small section of memory. After creating all the threads, the parent process waits for each thread to finish with the "pthread_join" function and adds the results of that thread's processing to a personal sum variable. This sum is then printed to the console at the end of operation.

Timing is accomplished using the <time.h> header file. The start and ending time of the workload are stored in a "timespec" struct variable. Then the difference between the two times is calculated and printed to the console.

This program is 75 lines long (including comments) with an executable size of 16376 bytes. The workload is accomplished in 40 lines.

1.3 Multitasking Code Implementation (With Fork)

The multitasking code implementation using the "fork" function is almost identical to the multithreading implementation. The parent process takes a parameter of the target number and number of tasks. It then iterates through the number of tasks, splitting up the workload between each child. In this for loop, the process opens a pipe and sets the lower and upper bound for the next process before calling fork. The results of the "fork" command are stored in a pid_t type array. After each "fork" call, the process checks if it is the parent or a child. If the process is a child, it breaks from the for loop and moves on to calculate the sum based on the current value of the lower and upper bounds. Because the child breaks from the for loop, the current value of the boundary integer "i" is also able to be used as a child identification number (the first child has i==0, the second has i==1, etc). Once the sum is calculated, the process writes the results on the pipe that corresponds with its integer identifier. The parent finishes creating all the children and then iterates through a for loop bounded by the number of tasks once again. This time, the parent calls "wait_pid" for every stored child PID before reading on the pipe corresponding to that child. The value on the pipe is then added to an internal sum and finally printed to the console once all children complete their computation.

Timing is accomplished using the <time.h> header file. The start and ending time of the workload are stored in a "timespec" struct variable. Then the difference between the two times is calculated and printed to the console.

This solution is specifically designed to mirror that of the threads solution. Multiple children are made in a single for loop with each having a designated portion of the workload to complete. An individual pipe is created per child which prevents any race conditions for writing/reading the data on the pipes but costs some memory space for the program.

This program is 95 lines long (including comments) with an executable size of 16528 bytes. The workload is accomplished in 58 lines.

1.4 Multitasking Code Implementation (With Popen)

The multitasking implementation with the function "popen" is very similar to the other multitasking implementation with one significant difference. The program takes four parameters on the command line: the lower bound to sum, upper bound to sum, number of tasks, and parent flag. The lower bound indicates the value to start the summation at. The upper bound indicates the exclusive value to end the summation at. The number of tasks indicates the number of tasks to delegate the summation to. The parent flag indicates if the process is the parent process.

The first thing this implementation does after reading in the parameters is check if the parent flag is 0. If that is the case, the process deems itself a child process, calculates the sum based on the upper and lower bound parameters and outputs that sum to the console before exiting.

If the parent flag is 1, the process deems itself the parent. It then divides up the workload based on the number of tasks it is supposed to create and creates a new task using the "popen" function that calls the same executable as itself with the parameters being the lower and upper bound for that child to sum, and the parent flag set to 0. The results of the "popen" function are stored in an array of file pointers. After creating all the child tasks, the parent task iterates through its number of children, calling "fscanf" on each file pointer allocated to each child. Because "fscanf" is a blocking function, this causes the parent to wait until something is printed to the file pointer which only happens when the child finishes its workload. The results of this scan are then added to an internal sum value and printed to the console.

Timing is accomplished using the <time.h> header file. The start and ending time of the workload are stored in a "timespec" struct variable. Then the difference between the two times is calculated and printed to the console.

This program is 88 lines long (including comments) with an executable size of 16440 bytes. The workload is accomplished in 49 lines.

1.5 Data Collection Code Implementation

Each script is modified slightly to be used by a custom "collect_data.sh" bash script that is used to collect multiple results. The "collect_data.sh" bash script runs each script 100 times and stores the output in a csv format. It supports setting different numbers of tasks as well as target summation values.

For ease of use, the baseline, multithreading, multitasking (fork), and multitasking (popen) scripts' outputs are changed slightly. Instead of printing pretty, human readable, outputs, the scripts only print the calculated sum and time taken to the console. This result is captured by the "collect_data.sh" bash script and printed into a csv file.

These implementations and bash script are included in the "data_collection" folder of the code repository as well as a single CSV file output that contains 100 iterations of the workload for 1,2,4, and 8 threads/tasks as well as summation values of 10^8, 10^9, and 10^{10}.

2 Analysis

2.1 Code Size

The following table shows the size of each program (in both lines of code and executable):

Implementation	Lines of Code (Critical Lines)	Size of Executable (bytes)
Baseline	30 (6)	16192
Multithreading	75 (40)	16376
Multitasking (fork)	95 (58)	16528
Multitasking (popen)	88 (49)	16440

As expected, the critical lines of code (lines that complete the workload) and size of the executable files increase across the cases with the baseline case being the smallest, followed by the multithreading, then the multitasking (popen) and finishing with the multitasking (fork) case.

Notably, the multithreading case is roughly 18.4% smaller lines of code than the multitasking (popen) case and roughly 31% smaller than the multitasking (fork) case. Because of the overhead that comes from making new processes as opposed to starting threads in the same process.

2.2 Workload Efficiency

The time it took for each case to complete the workload was recorded over 100 iterations across multiple different target values and thread/task counts. The iteration count was chosen arbitrarily based on the fact that one iteration takes about 1 minute to complete on the computer's hardware. The combination is as follows:

$$\mathbf{N = \{10^8, 10^9, 10^{10}\}}$$
$$\mathbf{NUM\_TASKS/NUM\_THREADS = \{1, 2, 4, 8\}}$$

*Because the baseline implementation cannot use additional tasks or threads, it was omitted when the thread/task count was above 1.

The results of running these tests are shown in the Appendix as Tables 1-3. As values of N increase by an order of magnitude, the average time taken to complete each process increases by roughly an order of magnitude.

For the multithreaded processes, the time taken decreases significantly with even 2 threads, and continues to decrease as more threads are added. However, there are diminishing returns. Having 2 threads to complete the workload decreases the time taken by roughly half, however having 8 threads only decreases the time taken to a quarter of the baseline. This could be because of the overhead taken to create the threads and synchronize the outputs, or because of hardware scheduling. The multitasked processes behave similarly with almost the same ratio, both implementations showing diminishing returns as additional processes are added.

This is observed almost certainly because the operating system is allocating each thread and each process to a core. When the script was run with the system monitor open, this was recorded in real time. Every time the script was run, the number of active CPU cores equaled the number of threads or tasks requested. Because the scripts were run on a 12 core CPU, there was always a CPU core available to allocate to the data collection script. There was not a clear winner between the multithreading and multitasking implementations. The final times for the largest workload were within 1% of each other. By a very small margin, the multithreaded application was faster than the multitask (fork) application which was in turn marginally faster than the multitask (popen) application. However, these differences were in the order of 100ths of seconds. This is likely because the operating system allocated resources in the same way whether the process was multithreaded or multitasked. Interestingly, in the largest workload, the multitasked (popen) process had a faster average mean than the multithreaded application. However, this does not explain the diminishing returns. That can potentially be explained by the limited scope of the workload. When 8 threads/tasks were allocated, the average time to complete the largest workload (10^{10}) was around 1.5 seconds. Perhaps the overhead of creating each thread/task and synchronizing takes up a large portion of this time as opposed to the actual calculation time.

While the means were not notably different, the standard deviations tell a different story. On the aggregate, the multithreaded implementation had a higher standard deviation than either multitasked application. This indicates that there was more variance in execution time than in the other cases. This is likely because the operating system may not have always allocated a singular CPU core per thread. The fact that this did not largely affect the overall average time to complete the workload shows that this happened few times over the 100 iterations.

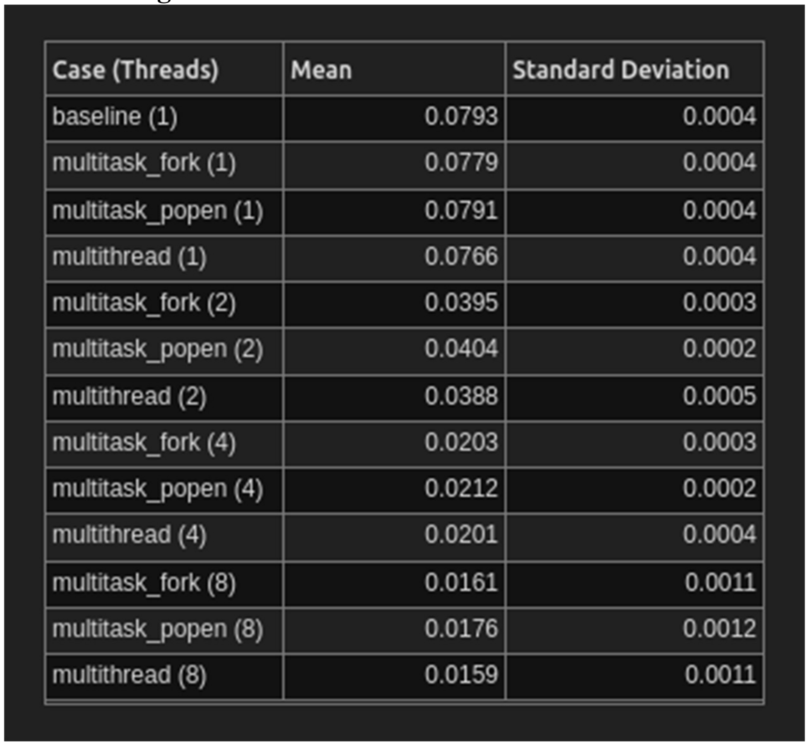
The differences between the multitasking case implemented using popen and the multitasking case implemented with fork is very slight, but present. The fork implementation has a faster completion time on average. This is likely due to the overhead of creating a new process using popen versus copying an existing process and opening a pipe using fork. Potentially passing information through a file pointer is slower than using pipes. However, these details are unable to be concretely analyzed with the current implementation and analysis tools.

2.3 Further Thinking

Because of the implementation, odd numbers of threads and tasks would not extremely adversely affect the completion time of the workload. The parent process assigns an interval to each child it creates and is the direct parent of all children who complete the workload. The code is written in such a way that any inconsistencies in workload length simply get allocated to the final child; the final child would add any remainder from uneven division of labor. If this child were delayed because many processes are running on the CPU, that could cause a significant deviation in completion time. Because of this, I would expect to see a greater standard deviation in cases that have uneven division of labor.

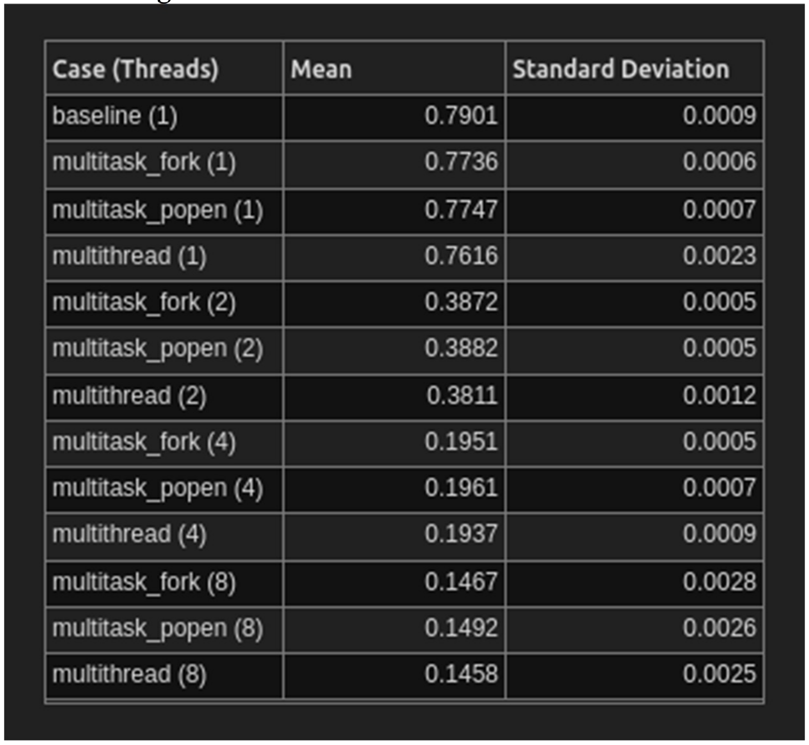
3 Appendix

Table 1: Target 10^8



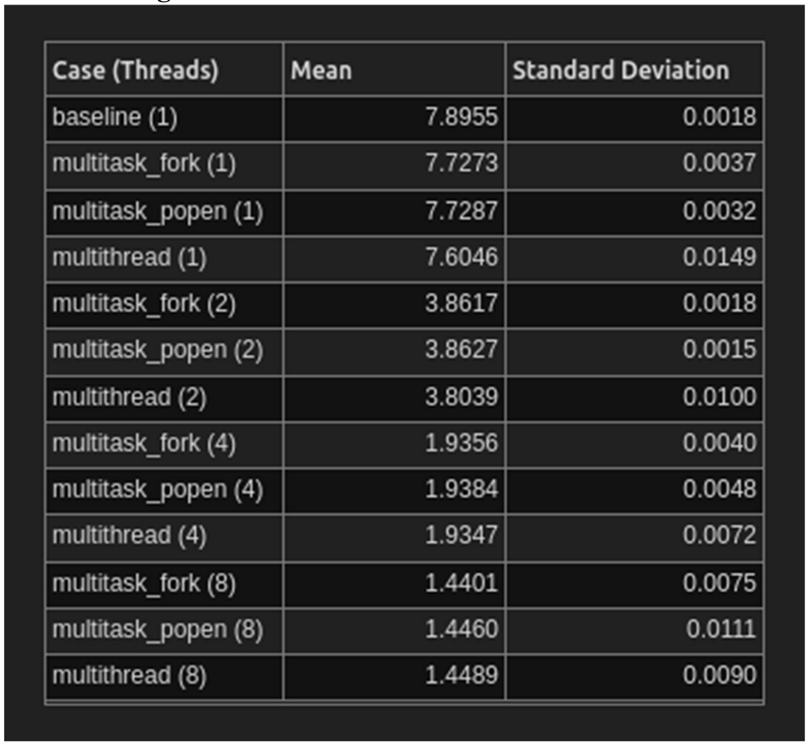
Case (Threads)	Mean	Standard Deviation
baseline (1)	0.0793	0.0004
multitask_fork (1)	0.0779	0.0004
multitask_popen (1)	0.0791	0.0004
multithread (1)	0.0766	0.0004
multitask_fork (2)	0.0395	0.0003
multitask_popen (2)	0.0404	0.0002
multithread (2)	0.0388	0.0005
multitask_fork (4)	0.0203	0.0003
multitask_popen (4)	0.0212	0.0002
multithread (4)	0.0201	0.0004
multitask_fork (8)	0.0161	0.0011
multitask_popen (8)	0.0176	0.0012
multithread (8)	0.0159	0.0011

Table 2: Target 10^9



Case (Threads)	Mean	Standard Deviation
baseline (1)	0.7901	0.0009
multitask_fork (1)	0.7736	0.0006
multitask_popen (1)	0.7747	0.0007
multithread (1)	0.7616	0.0023
multitask_fork (2)	0.3872	0.0005
multitask_popen (2)	0.3882	0.0005
multithread (2)	0.3811	0.0012
multitask_fork (4)	0.1951	0.0005
multitask_popen (4)	0.1961	0.0007
multithread (4)	0.1937	0.0009
multitask_fork (8)	0.1467	0.0028
multitask_popen (8)	0.1492	0.0026
multithread (8)	0.1458	0.0025

Table 3: Target 10^{10}



Case (Threads)	Mean	Standard Deviation
baseline (1)	7.8955	0.0018
multitask_fork (1)	7.7273	0.0037
multitask_popen (1)	7.7287	0.0032
multithread (1)	7.6046	0.0149
multitask_fork (2)	3.8617	0.0018
multitask_popen (2)	3.8627	0.0015
multithread (2)	3.8039	0.0100
multitask_fork (4)	1.9356	0.0040
multitask_popen (4)	1.9384	0.0048
multithread (4)	1.9347	0.0072
multitask_fork (8)	1.4401	0.0075
multitask_popen (8)	1.4460	0.0111
multithread (8)	1.4489	0.0090