Cacey Ostic and Nick Petrilli
Parallel Processing
Project 2

Project Overview

For Project 2, we were tasked with re-creating our Project 1 to now implement the solution with OpenMP clauses and constructs rather than using threads. To do this, we removed the code where our threads are spawned calling our countFrequency function. Instead, we only need to call the countFrequency function once as that is where our OpenMP implementation is.

CountFrequency Function

The countFrequency function contains the OpenMP logic that parallelizes our code. The entire contents of the function are wrapped around the following statement:
#pragma omp parallel num_threads(numThreads)
This line of code specifies that the following block of code is to be executed in parallel by multiple threads at a time. We then define the number of threads to execute using the num_threads clause, which takes the numThreads variable that is sent into the function as a parameter. Inside the parallel section, we initialize a HashMap so that each thread gets their own copy, called 'threadmap'. This threadmap will contain the words and their frequencies of the section of the file that that specific thread was assigned to.

Next we have the following statements:
#pragma omp for nowait
for (int i = start; i < end; i++)
These lines of code are used to parallelize a for loop by distributing iterations of the loop across the threads. The variable i is initialized with the value of start, which is zero, and looped until it reaches the value of end, which is the total number of lines in the file. The logic inside of the for loop remained the same from Project 1. The lines are traversed and each word is added to the threadmap and its count incremented when needed. The nowait clause specifies that threads should not wait at the end of the for loop and can start executing the critical section while other threads are still executing their iterations of the for loop.

When the for loop finishes, that means each thread has added all words to their threadmap, and each thread map needs to be merged into one hashmap. This is where we define this line of code:
#pragma omp critical
The critical construct specifies that the block of code is to be executed as a critical section, where only one thread can execute it at a time. This is because hashmap is a shared variable amongst the threads, and each thread needs to merge their thread map to the hashmap. There is a risk of a

race condition if multiple threads manipulate the hashmap simultaneously, which is why this block of code needs to be in a critical section.

Changes within Project 1

After the presentation on 3/7/24, we returned to Project 1 to fix some runtime issues with the algorithm. In the old Project 1, the hashmap data structure internally used shared_mutex to create shared and exclusive locks. Finding and accessing data only required shared locks, or read locks. Multiple threads could obtain read locks at the same time. Inserting new data or resizing the hash map required exclusive locks, or write locks. Only one thread could obtain a write lock at a time. This allowed all threads to access the data structure at the same time without causing race conditions. However, this ultimately caused more issues with the runtime of the program. Instead of the expected decrease in runtime as thread count increased, the runtime would seemingly randomly increase dramatically. These erratic increases can be seen by the red lines in the test result graphs. Likely, as a by-product of preventing race conditions, the hash map was overlocked and threads spent more time waiting than processing.

Therefore, for the new Project 1, the shared_mutex was removed. Instead, a single mutex is used in the count frequency function. This mutex surrounds the critical section of adding the data from the local hashmap into the global hashmap. As a result, only one thread can access the global hashmap at a time. The results of this change are depicted by the green lines in the graphs below.

Test Results

Tests results contain a table for Project 2 results, another table with our Project 1 old and new implementations (after our presentation on 3/7/24), and Project 2. Charts compare runtimes of all three implementations.

CPU Model

12th Gen Intel(R) Core(TM) i7-1255U
Base Speed: 1.70 GHz
Cores: 10
Logical Processors: 12

Constitution.txt

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	constitution.txt	26 KB	4503	1.59 MB	0.0027188
2				1.59 MB	0.0023737

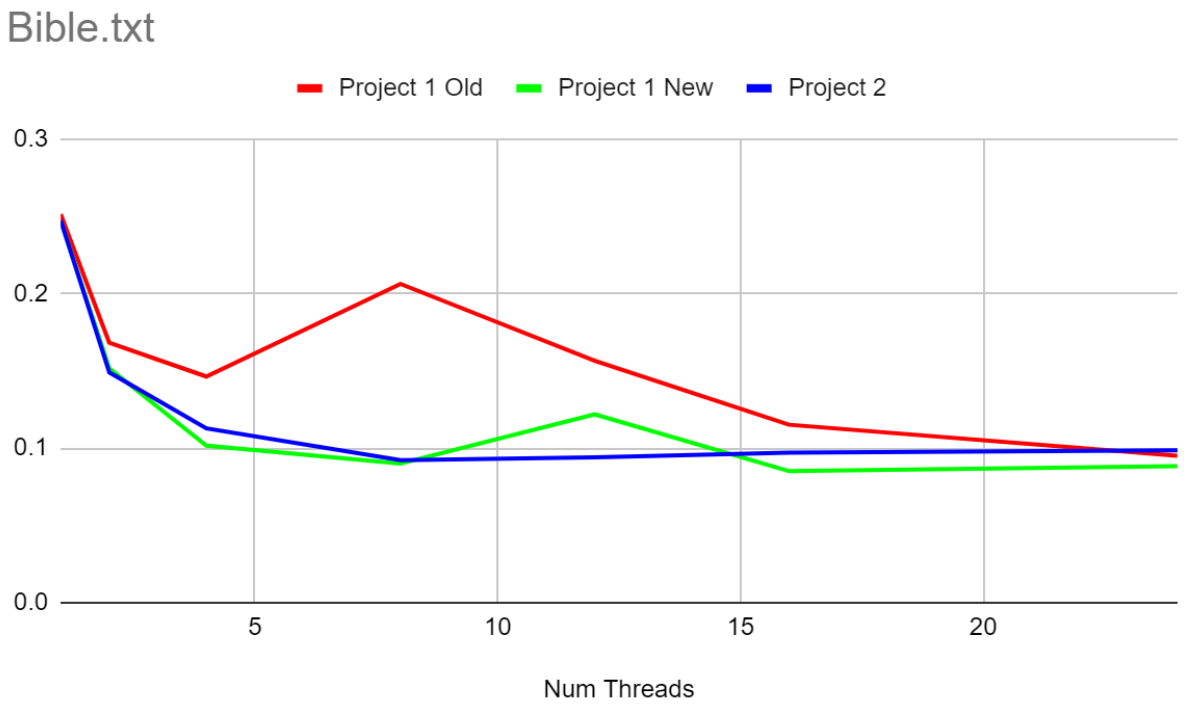
4				1.60 MB	0.0020972
8				1.60 MB	0.0029509
12				1.60 MB	0.0065435
16				1.60 MB	0.0028135
24				1.61 MB	0.003135

Num Threads	Project 1 Old (s)	Project 1 New (s)	Project 2 (s)
1	0.0067007	0.0031595	0.0027188
2	0.0061212	0.0026334	0.0023737
4	0.0051689	0.0024345	0.0020972
8	0.0055229	0.0031638	0.0029509
12	0.0082456	0.004251	0.0065435
16	0.0058346	0.0032114	0.0028135
24	0.0110729	0.0040866	0.003135

Bible.txt

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	bible.txt	4,332 KB	853,632	14.77 MB	0.247427
2				15.11 MB	0.148951
4				16.28 MB	0.112778
8				17.11 MB	0.0922105
12				18.59 MB	0.0939599
16				19.29 MB	0.0971113
24				19.54 MB	0.0986032

Num Threads	Project 1 Old (s)	Project 1 New (s)	Project 2 (s)
1	0.251366	0.247162	0.247427
2	0.168128	0.151619	0.148951
4	0.146302	0.101535	0.112778
8	0.206224	0.0900857	0.0922105
12	0.156501	0.121897	0.0939599
16	0.11512	0.0850465	0.0971113
24	0.0951076	0.0883957	0.0986032



Big.txt

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	big.txt	12,925 KB	2,231,692	33 MB	1.18071
2				40.76 MB	1.12592
4				46.25 MB	0.491859
8				48.35 MB	0.653443
12				50.14 MB	0.425101
16				54.28 MB	0.4942
24				60.71 MB	0.393602

Num Threads	Project 1 Old (s)	Project 1 New (s)	Project 2 (s)
1	1.21651	1.20344	1.18071

2	1.45206	1.09252	1.12592
4	0.581993	0.45707	0.491859
8	0.316373	0.413397	0.653443
12	0.598773	0.414019	0.425101
16	1.27184	0.402826	0.4942
24	1.61473	0.311648	0.393602

Verybig.txt

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	verybig.txt	29,210 KB	5,124,675	72.75 MB	4.75055
2				75.41 MB	2.07875
4				91.03 MB	2.18754
8				96.01 MB	1.55312
12				100.65 MB	1.06135

16				104.75 MB	1.03222
24				113.75 MB	0.937127

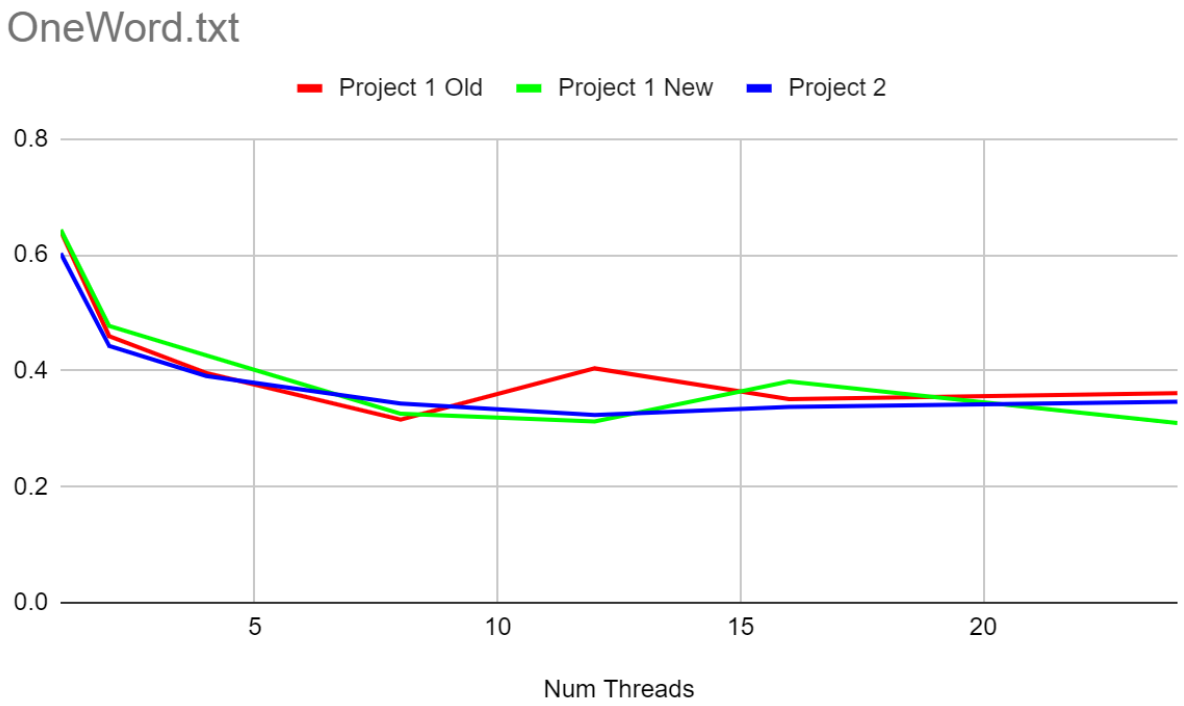
Num Threads	Project 1 Old (s)	Project 1 New (s)	Project 2 (s)
1	3.69576	4.0094	4.75055
2	2.3562	2.18026	2.07875
4	16.5716	1.2812	2.18754
8	13.1414	1.45921	1.55312
12	13.8329	0.9488	1.06135
16	13.2985	0.805008	1.03222
24	8.50777	0.858468	0.937127

OneWord.txt

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)

1	oneWord.txt	23,555 KB	6,000,000	36.54 MB	0.60282
2				36.59 MB	0.442706
4				36.61 MB	0.391155
8				36.68 MB	0.34355
12				36.74 MB	0.32366
16				36.85 MB	0.337848
24				37.00 MB	0.346555

Num Threads	Project 1 Old (s)	Project 1 New (s)	Project 2 (s)
1	0.641207	0.643272	0.60282
2	0.459928	0.477298	0.442706
4	0.396143	0.426494	0.391155
8	0.315945	0.325926	0.34355
12	0.40418	0.312556	0.32366
16	0.351265	0.381686	0.337848
24	0.361441	0.310081	0.346555



Conclusions:

Overall, the OpenMP version generally performed faster as more threads were used to run the program. Almost all of the test cases had a consistent decrease in time as the thread count increased. Additionally, every test case showed a steady increase in memory consumption as more threads were used.

In the cases of “bible.txt”, “big.txt”, “oneWord.txt” and “verybig.txt”, using 2 and 4 threads resulted in the largest improvements of runtime. Beyond 4 threads, the improvement started to level off, resulting in only minor improvements for each additional thread count. In some cases, there was a slight increase in runtime after about 12 threads. This is likely because the overhead of creating threads eventually overtook the benefit of splitting the text into more parts. This overhead cost can more easily be seen in the “constitution.txt” results: the one exception as compared to other test cases. The results show that using more threads generally increased the runtime of the program. This is likely due to the small size of the file. The benefit of processing the file with multiple threads was outweighed by the overhead cost of creating and maintaining the threads.

The new version of Project 1 also performed as expected. In all of the test cases, the new Project 1 time graph followed the OpenMP graph very closely with only slight differences. There was significant improvement with 2 and 4 threads and the time began to level off around 8 threads. It also performed similarly to OpenMP in the outlier case of “constitution.txt”. This is

further evidence that the file is too small to be properly sped up with multithreading before the overhead of managing threads becomes significant.

Despite this outlier, both the new Project 1 and OpenMP versions performed much more consistently than the old Project 1 version. While some cases, like “oneword.txt” and “bible.txt”, show similar results between the two methods, the old Project 1 version performed much more erratically in other test cases. This is most easily seen in the “big.txt” and “verybig.txt” cases. Both the new Project 1 and OpenMP versions had a consistent decrease in runtime, resulting in similar curves for both cases. On the other hand, the old Project 1 results look nothing alike, with large jumps and dips not present in other test cases. This makes the old Project 1 results more inconsistent and unpredictable.

Additionally, the OpenMP version was much simpler to implement than the threading used in Project 1. When converting Project 1 to Project 2, a large section of code was replaced by only a few OpenMP directives. OpenMP simplified the process of parallelizing the algorithm. Less code means less chance for human error in programming. This error is a possible explanation for the unpredictable results in the old version of Project 1.

In summary, the results show that the runtimes of using OpenMP and the new Project 1 version were much more consistent, both with themselves and compared to the results from the old version of Project 1. This, combined with the comparative simplicity of implementing OpenMP, makes OpenMP the better choice for parallelization.