Liam Bishop(lb3306), Mike Maher(mrm2281), Ryan Kennedy(rdk2132)
Computer Architecture HW 4

Testing

For testing these programs we had three sets of testing strings. The first two strings were around size 32, the second were size 1000, and the largest strings were length 100000. Our average run times were computed by running each of the optimized programs 100 times and returning the average run time. For optimizations, we focused on starting with a simple program, and making optimizations from there instead of using more complex algorithms. Additionally, many of the algorithms we looked at required storing M*N elements in memory which is impossible to do for our large strings.

Program: slowestSubstring.c
Optimizations: None

We started with a naive approach which iterated over the entire length of two strings doing comparisons to find the longest common substring. This program consisted of two size N for loops and had $O(N^2)$ runtime. This completely unoptimized program took about 80 seconds for our big strings.

Program: fasterSubstring.c
Optimizations: threading, compiler optimization, and hardware optimization.

The first optimization we tried was multithreading which would enable us to do multiple smaller string comparisons in parallel, thereby greatly increasing the search speed of the program. By having multiple threads start at different points of a string and compare them in increments the result is achieved much faster.

We also added in a hardware optimization to assist in branch prediction using unlikely and likely in our program. We experimented with both and realized that they had varying results for strings depending on how many comparisons were correct. This would stand to reason that for strings with many common substrings and few non matching characters likely would yield better results, and vice versa for unlikely. We decided to use unlikely for our program given that we were optimizing for very large strings. This optimization had no noticeable effect on the run time of the program. We guessed it was because checking for equality between the characters was simply not a bottleneck in the program.

These optimizations improved the runtime of our program from about 80 seconds to 16 seconds for 100 threads with large input strings. When we added in a compiler optimization flag (O3, the fastest compiler optimization available) the runtime of this threaded program further improved to an average run time of 8.9s for strings of length 10000.

Program: fasterSubstringVectorized.c
Optimizations: vectorization and compiler optimization (O3 flag once again)

We then made a separate program based on our multithreaded program with vectorization added to see how it would compare to the multithreaded one. However, when we added vectorization to our multithreaded program we realized that vectorization is not thread safe (our test machine achieved 100 percent cpu usage and 1.5 GB of memory). It's possible

this was a limitation from our computing power and would be beneficial when combined with a more powerful computer.

As a result we took the threads down to 1 and simply added vectorization using SIMD functions to the comparison to see how it performed in comparison to multi-threading. This allows the program to compare several chars in parallel rather than individual chars and should give some speed boost to the program by enabling some simultaneous comparisons. For our program we used vectors of size 16 to store chars meaning that we could do 16 comparisons at once rather than 1 at a time. With the compiler options mentioned in the multithreaded program, the average run time of the vectorized program came out to

Time Log:

Date:	Time worked:	People:
12/3	3 hours	Liam
12/4	2 hours	All
12/6	30 minutes	Liam
12/12	2 hours	Ryan, Mike
12/13	4 hours	All