David Wade

Concurrent Programming

Lab 3

Write-Up

For this lab, we were tasked with performing parallelization for one of the previous sorting algorithms utilizing openMP. I elected to parallelize merge sort as I found it easiest to parallelize in Lab 1. I feel like openMP is designed to be easier than parallelizing with pthreads, but I had difficulty directly translating my parallelization approach from Lab 1 to this lab. This is mainly because I feel like openMP thrives when parallelizing iterations of a for loop since it will use its algorithm to determine the optimal number of threads to fork and the best fork join parallelization approach. I personally like pthreads parallelization more because I feel like I have more control over the flow of the program rather than trying to accommodate a new algorithm for openMP. I could see openMP being extremely useful in scientific applications where one needs too loop through large amounts of data, however in recursive applications, pthreads is easier to manage. Although, this could be strictly due to my lack of experience using the openMP tool.

My parallelization strategy for Lab 1 was to split the data into n threads and run merge sort on each chunk of data. Similarly, I had the same approach for this lab, but trusted openMP to determine what it thought to be the optimal number of threads. However, through testing, openMP would often spawn more threads than data, which resulted in failures majority of the time and success occasionally. As a result, I had to set an active thread limit for openMP that

equaled the size of the data. My intuition leads me to believe that most executions involved the same number of threads as the size of the data.

In this lab, my code is organized into two C++ files each with a corresponding header file. In addition, there is a make file, a shell script, and a folder containing all of the test cases used in Lab 1.

The first C++ file is the Driver for the whole application. This file simply handles the user input from the command line, creates an instance of the MergeSort class, calls parrallelMergeSort, and outputs the resulting sorted data into an output file. The second C++ file is the Merge_Sort file. This files contains the MergeSort class and all of its corresponding methods. The methods include, the parallelMergeSort, split_data, mergeSort, and merge_vectors. MergeSort and merge_vectors are both standard merge sort algorithms. The parallelMergeSort is where the openMP parallelization of the sorting algorithm is located. The split_data is themethod that splits the data into the number of threads. The driver calls the paralllelMergeSort method in order to perform the sorting algorithm. In addition, there is a shell script I wrote that performs all the test cases with a single command line input.

In order to compile this lab, one simply needs to be located in the Lab's directory and type a "make" command in the command line. This will go ahead and compile all the associated files in the project.

In order to execute this lab, one must first compile, then perform aan executable called mysort with the syntax "./mysort [--name] [source.txt] [-o out.txt]". If the name argument is given, then the program will display my name and terminate. Otherwise, the program will read

the input file source, sort thew file, and then output it to a file named of the user's choosing. If an invalid input is given, the program will. Give a Usage error.

There are no extant bugs in this lab.