**Report on Parallelization of**

**the linear algebra C library**

A report on the comparison

between serialized and parallelized code

Connor Felton

Jonathan J. Allarassem

Grove City College

COMP233

4/27/2022

1. **Race conditions**

We avoided race conditions by using a for directive as well as reduction and private clauses in the compiler’s directives.

1. **Explanation of the parallelization**
2. **Parallelization of addition**

For the addition method, we parallelized the part where we had to access different indexes of the array and add them together to get the new array. We used a for directive that avoids race conditions by dividing the array in multiple chunks that will then be assigned to different threads. We did not need a reduction clause or critical section for the matrix ret because no two threads access the same indices of ret, so there is no conflict there. The collapse clause allows the work of the nested loops to be more evenly distributed between the threads, maximizing efficiency. For addition, the parallelized version becomes faster past matrices of size 250x250.

1. **Parallelization of l1**

For the l1 method, we parallelized the part where we had to access different indexes of the matrix vals [][]. The inner for loop uses a variable toAdd for calculations and stores results in the results variable. toAdd is protected with a private clause so that every thread gets its own copy, and results is protected with a reduction clause to add the results of the threads efficiently and safely. For addition, the parallelized version becomes faster past matrices of size 100x100.

1. **Parallelization of l2**

For the l2 method, we parallelized the part where we access the indexes of the variable and computing the squares. We used the for directive to distribute the work between the threads and use the collapse clause so to transform the two nested loops into one big loop which optimize the work of the thread, we also had ret matrix as a reduction variable. For L2, the parallelized version becomes faster past matrices of size 100x100.

1. **Parallelization of multiplication**

For the Multiplication method, we parallelized the part where we had to access different indexes of the array and multiply them together to get the new array. We used a for directive that avoids race conditions by dividing the array in multiple chunks that will then be assigned to different threads. We did not need a reduction clause or critical section for the matrix ret because no two threads access the same indices of ret, so there is no conflict there. The collapse clause allows the work of the nested loops to be more evenly distributed between the threads, maximizing efficiency. For addition, the parallelized version becomes faster past matrices of size 5x5.

1. **Benefits**

Some functions like multiplication needs to do a lot of computations. For these functions, the benefits will be more visible because they require a lot of computations compared to very simple functions like L1 that do not require much computations. The greater the workload, the bigger the benefit from splitting it.

1. **Is it worthwhile?**

Whether or not parallelization was worth it really depends on the size of matrix you’re dealing with. Usually, college students or average people using excel for example, do not really use very big matrices. It is usually matrices of less than 1000 elements. Hence, unless if you are doing some sort of research or big project where you need to read a lot of datapoints, the performance of the parallelized version compared to the serial version is not big enough to be worth it.