Homework 2 - Report

Group Members
Anton Hammar - antham@kth.se
Ludwig Flodin - ludwigfl@kth.se

Compute the Sum, Min, and Max of Matrix Elements
For this assignment the goal was to create a parallel program that computes the sum, min and max for a matrix using the API openMP. The code for calculating the values is written the same as a sequential program, the matrix is iterated through with a nested for loop and the min and max are compared to all the values on the row and will be assigned a new value if necessary.

To create a parallel program using openMP the command “#pragma omp parallel for” is used before the for loop to indicate that this is supposed to be run as parallel and not sequential. Calculating the sum is made easy with openMP command reduction which will either add or minus the result of the row together with all the other rows. In this case the reduction will add the sum of each row to create the sum of the whole matrix. To avoid race conditions each element of the matrix that is looked at will be private for each thread, this will ensure that the program gets the correct values and positions.

The times shown below are the program’s measured median time when running the code with a number of threads between 1 and 4. The speedup increases roughly linearly with each increase to the number of threads. This is easy to notice with a matrix size, when the matrix is smaller the difference in median time with a different number of threads is less noticeable. However speedup is still visible when making big jumps in the number of threads.

The decrease in noticeable speedup when the matrix is smaller is because of diminishing returns, a high amount of threads are not necessary if the data that is worked on does not need a big number of tasks. In the worst case scenario the overhead can cause the parallel program to be even slower than the sequential if the amount of data or tasks are small enough where the sequential program can run and finish the code will the parallel code will be working overtime on scheduling instead.

Matrix size of 10 000

Threads	Time (sec)	Speedup (s/p)
1	0.131	1.000
2	0.068	1.941
3	0.044	2.977
4	0.035	3.797

Matrix size of 5 000

Threads	Time (sec)	Speedup (s/p)
1	0.033	1.000
2	0.017	1.941
3	0.012	2.750
4	0.009	3.667

Matrix size of 1 000

Threads	Time (sec)	Speedup (s/p)
1	0.0010	1.0000
2	0.0010	1.0001
3	0.0010	1.0002
4	0.0005	2.0005

Quicksort

This code sorts an array of numbers using the QuickSort algorithm. It starts by taking two command line arguments which sets the number of threads using OpenMP and fills an array of a certain size with random numbers.

After the setup it reaches the parallel part which it starts with a #pragma omp parallel line. After this it calls the quickSort function inside a #pragma omp single nowait to ensure that only a single thread calls the method and that other threads does not have to wait for it.

The quickSort function takes the array, leftmost element and rightmost element as arguments. It starts by selecting a pivot and then iterating from left to right and swaps elements to ensure that only values smaller than the pivot is on the left side of it and values greater is on the right side.

After this it will call the quickSort function twice, for the left and right side of the array, where it will repeat the swapping but only on its own side. If it has more than 1 thread it will call one of the quickSort in a #pragma omp task command, which will give the threads a task to work on when a thread is available. The firstprivate command is used on the array, left and right values to ensure they are private and won't affect each other in other threads while the contents of the variables are still kept.

This recursion will continue until each thread would work on a single element in the array. At that point the array will be completely sorted and the program prints the time it took and ends.

As can be seen from the measured times and speedup in the tables below, it is clear that more threads makes the program run faster. This also increases the speedup appropriately. This makes sense, as the program does a lot of separate tasks at the same time, which fits perfectly for a parallel program and it is therefore very useful to use threads in these cases. However, when the array is smaller, the time gain is not very noticeable. It is not free to use threads and at some point, it is not worth it to use them instead of just running the program in a sequential manner. This can be seen on the third table with an array size of 100000 elements.

Array size of 1000000

Threads	Time (sec)	Speedup (s/p)
1	7.15	1.000
2	3.682	1.942
3	2.52	1.942
4	1.95	3.667

Array size of 500000

Threads	Time (sec)	Speedup (s/p)
1	1.8	1.000
2	0.972	1.852
3	0.68	2.647
4	0.553	3.255

Array size of 100000

Threads	Time (sec)	Speedup (s/p)
1	0.075	1.000
2	0.059	1.271
3	0.056	1.339
4	0.058	1.293