Multiprocessor Systems (DV2544) – OpenMP Programming – Project Implementation Report

Name	Robin Gustafsson	Trishala Chalasani
Personal number	931220-1271	930602-0281
E-mail	roga15@student.bth.se	trch15@student.bth.se

QUICKSORT

AIM: To implement a parallel version of Quicksort using OpenMP directives and library functions and Measure the execution time and speedup the parallel version on 8 CPUs.

ASSUMPTIONS: An array of randomly initialized values is taken as the input. It is always the same size (64*1024*1024). The values of the array are integers.

IMPLEMENTATION: Work is divided among the CPUs by the use of task pool. The divide-and-conquer step is implemented using task pool where all the threads pick tasks dynamically. Due to the nature of quicksort the tasks will generally get smaller towards the end, which would make it similar to a type of "guided" scheduling approach. The task pool is implemented using OpenMP's task directive.

An OpenMP parallel block is used to create the threads. Within the parallel block, a single block is used to create a task for the full sorting algorithm (without the single block, every thread would create that task meaning the sort would be performed once for each thread). Within the sort, a new task is created for each half (i.e. for each sorting task in the next step). At the end of the parallel block is an implicit synchronization barrier which makes sure that all tasks are finished before destroying the threads and moving on.

TIME MEASUREMENTS:

The array size: 64*1024*1024 (= 67108864)
The number of threads: default (i.e. 8 on kraken)

Measurements of the execution time of sequential and parallel implementation at 5 runs each are as follows:

	Sequential implementation	Parallel implementation	Speedup
Run 1	18.75	4.88	
Run 2	18.47	4.57	
Run 3	18.47	4.64	
Run 4	18.47	4.51	
Run 5	18.47	4.81	
Average	18.53	4.68	3.96

CONCLUSION: We have fixed the array size and the number of threads for the parallel version. We have noted the values at five runs for the sequential as well as the parallel version. We then calculated the Speed up by dividing the average sequential time upon average parallel time. The speedup obtained is 3.96 which is greater than the required speedup (2).

GAUSSIAN ELIMINATION

AIM: To implement parallel version of Gaussian elimination using OpenMP programming and measure the speedup of parallel version on 8 CPUs.

ASSUMPTIONS: The Matrix A and the vectors y and b are initialized with reasonable values. The number of CPUs (cores) are less than the size of the matrix A.

IMPLEMENTATION: Some changes have been made for the given Gaussian elimination implementation with the help of OpenMP compiler directives. We parallelized the division step by the usage of the parallel for directive using static scheduling with the default chunk size (which means the iteration is split into as many sequential blocks as there are threads).

The elimination step is performed using parallel for with static scheduling and a chunk size of one (meaning each block is one row, and these blocks are then distributed cyclically among the threads).

TIME MEASUREMENTS:

The matrix size chosen: 2048 x 2048
The number of threads: default (i.e. 8 on kraken)

Measurements of the execution time of sequential and parallel implementation at 5 runs each are as follows:

	Sequential implementation	**Parallel implementation**	**Speedup**
Run 1	11.38	6.31	
Run 2	11.38	6.36	
Run 3	11.40	6.30	
Run 4	11.38	6.24	
Run 5	11.41	6.33	
Average	11.39	6.31	1.81

CONCLUSION: The implementation of parallel program is a way simpler and easier with the OpenMP than MPI and Pthreads. We note down the values of the sequential version and our parallel implementation at five different runs. Having found that the average of the five runs for both sequential and parallel version makes it reliable and is useful for calculating the speedup. Our parallel implementation with OpenMP has a speedup of 1.81 (on 8 CPUs) as compared over the sequential version, which is sufficient with regards to the requirement of 1.5