Assignment 2: OpenMP

1. Matrix Multiply in OpenMP

Each thread gets a portion of outer loop. i, j and k needed to be made private (declared inside the loop header) because we want each thread to have their own portion of iteration variable. We do not want each thread to write on the global index.

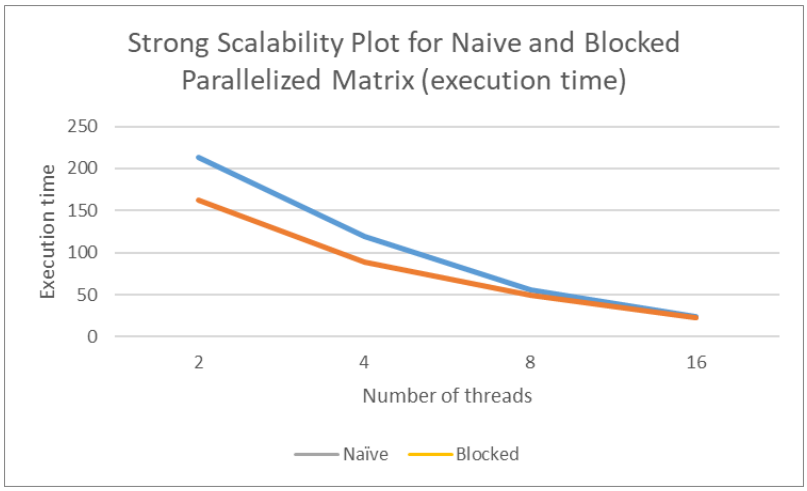

Figure 1: Execution time plot of matrix multiplication in regard to thread count

2. Scheduling in OpenMP

(b)

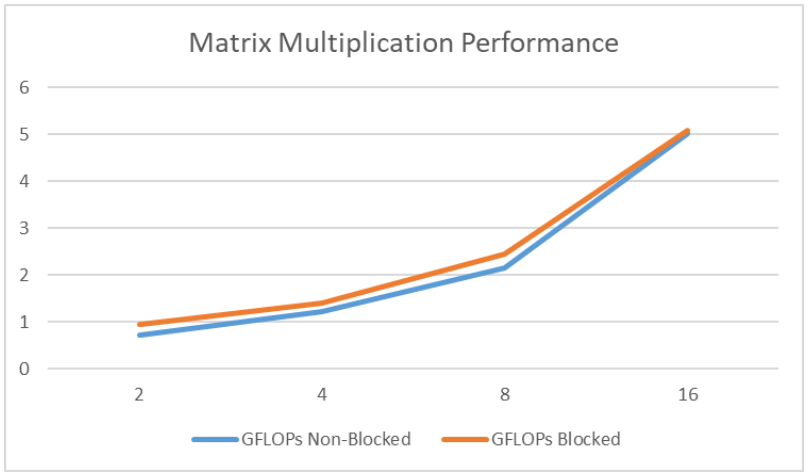

Figure 2: Performance measure for sparse non load balanced matrix multiplication (blocked and naive). It is clear that naive implementation is less performant then blocked version.

(c) Guided scheduling was used instead of dynamic for it performed better. The reason is that chunkSize = 1 (dynamic case) causes false sharing.

(d)

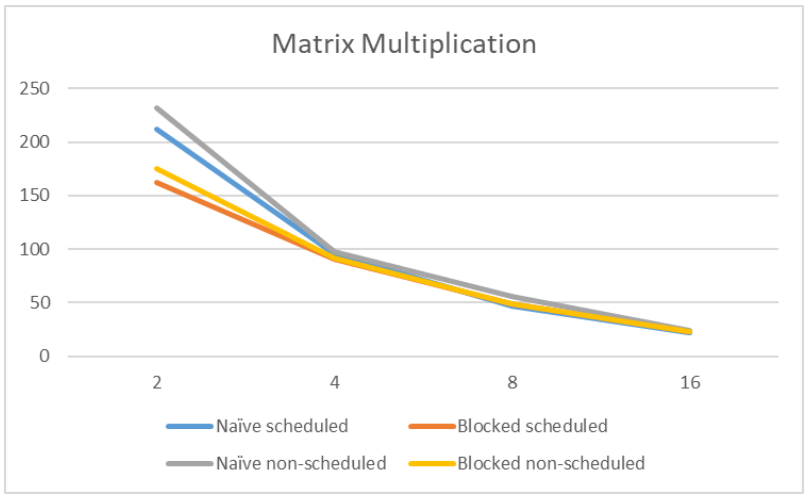

Figure 3: A strong scalability plot (measuring execution time) for both the methods with and without using load balancing with numThreads = 2, 4, 8, 16, and size = 4000. It is clear that scheduling has most impact on sparse matrix multiplication when thread count is low (as with more threads work is spread more, thus diminishing scheduling returns).

3. Task programming in OpenMP

20 runs have been chosen to average runtime more correctly.

(a) Enclosing taskloop construct in a parallel single region is required. Encapsulating taskloop in the single region forces only one of all threads once create all the tasks for parallel loop. The parallel region is needed for threads to be created, which later take tasks from taskpool.

(b)

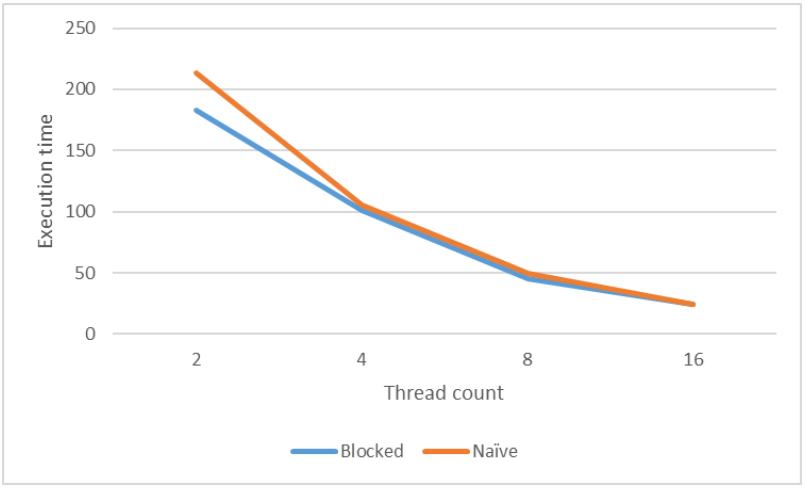

Figure 4: The strong scalability report of execution time of matrix multiplication performed under different thread count (implemented via OMP tasks) using naive and block-based approaches. It is clear that the fewer threads are used, the greater the execution time difference between both methods exists.

(c) In general, task programming allows dividing matrix multiplication problems into sub-problems. Each sub-problem is treated as a task, which can be executed in parallel by threads. This enhances performance and the runtime. In the case sparse or upper/lower triangular matrices, task programming allows flexibility in deciding sub-problem size. In this case, it is possible to make sub-problems small enough for performance to be increased (i.e. sparse workload is distributed among threads as equally as possible, therefore all threads do nearly similar amount of works). It works in a similar fashion as dynamic/guided scheduling for parallel loops.