Ashwin Sarathi Krishnan
asarath@ncsu.edu

ECE 506 Project 1 Report

1. Number of parallel loops and it's functionality.

There are a total of **6 parallel loops**. Their functionality is as follows:

a. **Evolve**: It performs time evolution on an input array u0 by t steps to produce an output array u1. The function evolves a complex field in Fourier space by applying a time dependent phase rotation to each Fourier mode. In this loop, OpenMP seems to parallelize the outermost loop k, keeping i, j and k as private variables and the others are shared. There is an implicit barrier at the end of the loop to aid synchronization.

b. **Compute Indexmap**: This loop parallelizes the process of mapping local i, j, k coordinates to values representing $i^2 + j^2 + k^2$ in Fourier space. In this case, the outermost loop i is being parallelized. Because of this parallelization, variables such as i, j, k, ii, ii2, jj, ij2 and kk can have conflicts and are hence privatized. There is an implicit barrier at the end of the loop to aid synchronization.

c. **Cffts1**: This loop helps parallelize 1D FFT along one dimension of a 3D array. By declaring dcomplex y0 and y1 inside the parallel region, each thread gets its own copy avoiding race conditions. The variables i, j, k and ii are privatized and the remaining variables are shared by default. The function parallelizes the outermost loop k. The data is processed in blocks of size fftblock to improve cache utilization. There is an implicit barrier at the end of the loop to aid synchronization.

d. **Cffts2**: The idea is the exact same as cffts1, but just performs 1D FFT along a different dimension.

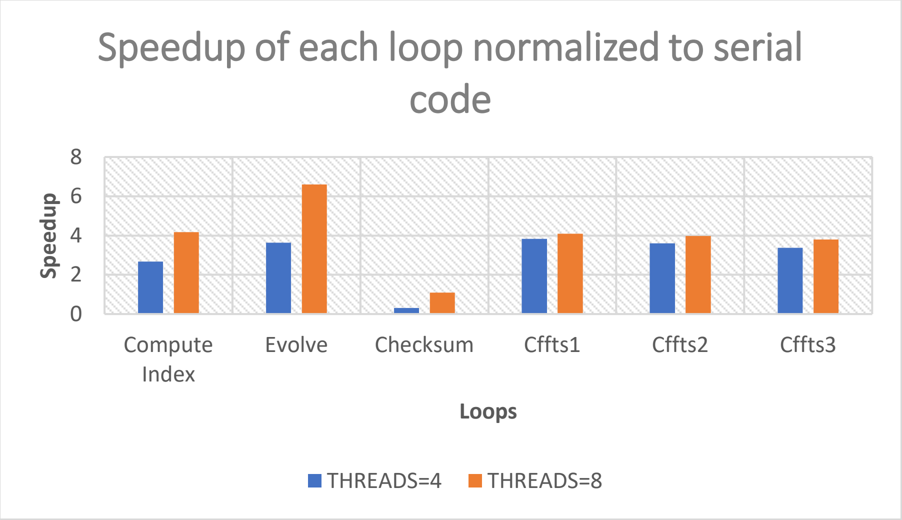
e. **Cffts3**: The idea is the exact same as cffts1 and cffts2, but just performs 1D FFT along a different dimension.

f. **Checksum**: This function uses OpenMP to parallelize the computation of a checksum for the 3D array. First a zone for parallelization is created with all variables being shared. The threads parallelize the loop over j and proceed without waiting for other threads to finish. Then upon reaching the critical section, only one thread is given control at a time to ensure atomicity. After the sum is calculated, a barrier is implemented to ensure that all threads have finished checksum calculations. Then the final calculations and output is executed on a single thread and other threads wait implicitly.

2. Bar graph of speedup of each loop (with THREADS=4 and 8) normalized to serial code (THREADS=1).

The graph below is a representation of the speedup on a per loop basis as we increase the number of threads.

Ashwin Sarathi Krishnan
asarath@ncsu.edu

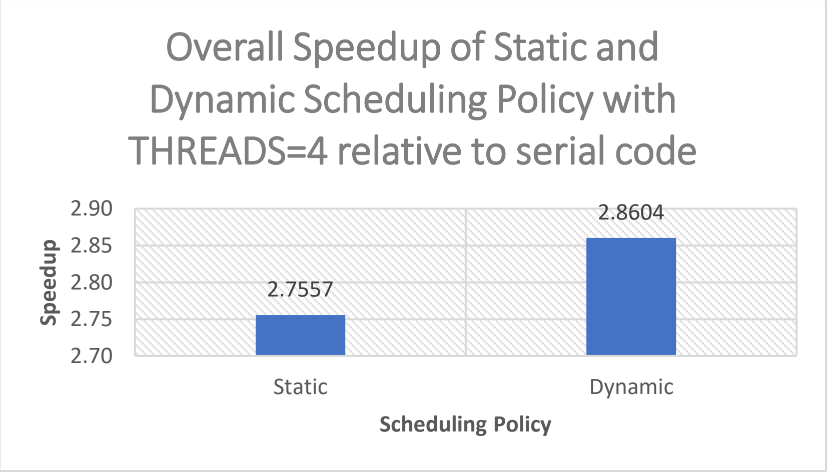


3. Explanation of default scheduling policy in OpenMP with reference and explanation.

The default scheduling policy in OpenMP is **static mode**, with the iterations being evenly distributed into chunks. These chunks are then allocated to various threads. If iterations are not a multiple of thread count, the chunk size is approximated. The assignment does not change during run time. This policy is useful when all iterations take approximately the same time to execute, there's minimal overhead in dividing work upfront and the workload is predictable and evenly distributed. I took my reference from - OpenMP_Dynamic_Scheduling.pdf (ufsc.br)

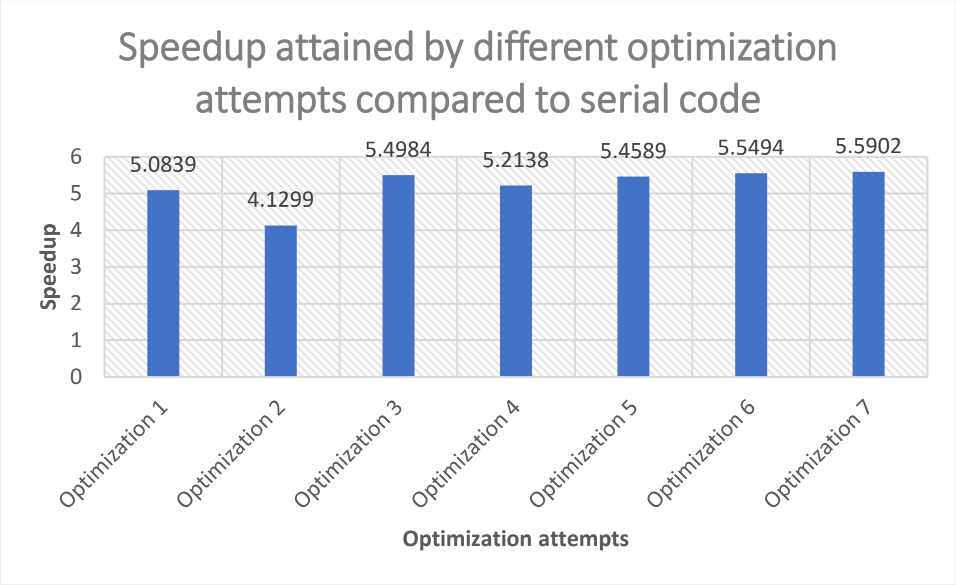
There is also an approach to get the default scheduling policy from within the program during run time. A variable of type omp_sched_t is created. Then inside the parallel portion of the loop, this variable is assigned the name of the current scheduling mode by using the **omp_get_schedule()** function. If no scheduling mode is predefined, the value held by this variable should be "omp_sched_static".

4. Bar graph of overall speedup of Static and Dynamic Scheduling policy with THREAD=4 relative to THREAD=1.



Ashwin Sarathi Krishnan
asarath@ncsu.edu

5. Bar graph of speedup with the proposed optimization relative to THREAD=1 along with an explanation.



The unique features in each run are as follows:
Optimization 1: 10 threads, static scheduling, chunk size 10, 32 cores
Optimization 2: 10 threads, dynamic scheduling, chunk size 10, 32 cores
Optimization 3: 10 threads, static scheduling, chunk size 2, 32 cores
Optimization 4: 10 threads, dynamic scheduling, chunk size 2, 32 cores
Optimization 5: 10 threads, static scheduling, chunk size 2, 12 cores
Optimization 6: Removing timer checks in main() with optimization 5 conditions
Optimization 7: Parallelizing the n iteration loop in cfftz() with optimization 6 conditions

10 threads seem to be the most optimal thread count for this program. This could be because under 10 threads, there is underutilization of the processing power and over 10 threads, the switching overhead is too high.A slight boost was observed when the number of cores was dropped from 32 to 12 cores. This is probably because assigning threads to cores was made easier with lesser cores.

It can be observed through runs 2 and 4 that a dynamic policy is detrimental to the system. This could be because dynamic scheduling works best when the workload is unbalanced and unpredictable, but the workload per iteration in this program is fairly constant. Lower chunk sizes resulted in a significant performance boost because there is better cache utilization at a smaller chunk size, and fewer consecutive iterations are processed before switching. This also allows for finer grained load balancing in case of system level fluctuations.

Removing the timer checks in the main function helped improve performance slightly. Finally, the most optimal conditions were achieved in optimization run 8 when the n iteration loop in cfftz() function was parallelized while having 10 threads, static scheduling with a chunk size of 2 and 12 cores. A maximum speedup of 5.59 compared to serial code was achieved in these conditions.