1. Report the number of loops and what each of the loops is doing.

Ans: There are 6 parallelized loops in the code. Their functions are as follows -

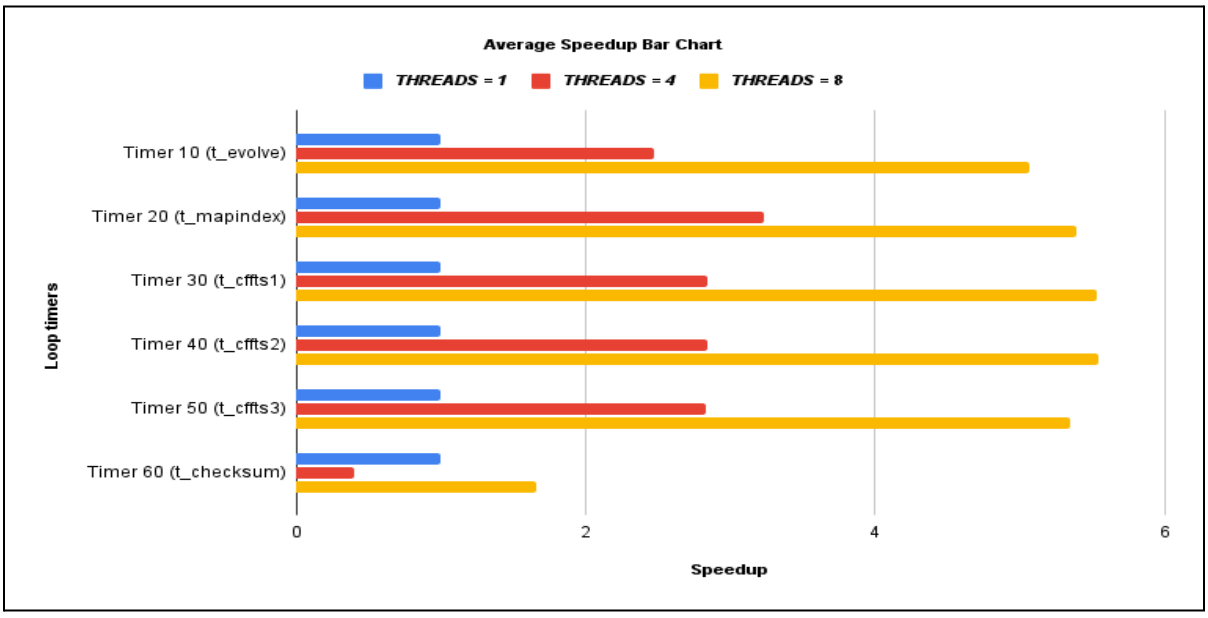
- Loop 1) This loop is part of the evolve() function that transforms data from u0 into u1 in Fourier space over t steps in time. The pragma directive parallelizes the outermost loop (k-loop) of the nested loop. All variables are shared among threads by default except for the loop indices i,j,k. The 'crmul' function performs complex multiplication between u0 and ex arrays and stores the result in u1.

- Loop 2) This loop is part of the compute_indexmap() function. The pragma directive parallelizes the outer 'i-indexed' loop and explicitly defines a set of private variables among the default shared variables. The function computes an index mapping for a 3D grid which will be used later in time evolution calculations for FFT. The loops iterate over the 3D space and calculate transformed indices by effectively shifting the origin of the coordinate system.

- Loop 3,4,5) The pragma loops in three functions (cffts1, cffts2, and cffts3) are each performing parallel 1D FFTs along different dimensions of a 3D complex array. All three share a similar structure where they a) declare private copies of the y0 and y1 arrays for each thread b) use 'omp for' to distribute the outermost loop and c) use a block-based approach (fftblock) to improve cache utilization. Both cffts1 and cffts2 functions parallelize the loops over the Z dimension (k-loop) while the cffts3 function parallelizes the loop over the Y dimension (j-loop).

- Loop 6) This loop is part of the checksum function and parallelizes the computation of a checksum for result verification. All variables are shared by default and a 'nowait' clause allows the threads to proceed without waiting for others to finish the loop. This clause is followed by a critical section that ensures the addition of partial sums is done correctly. The 'omp barrier' synchronizes all threads and 'omp single' ensures that only one thread performs the final calculation and prints the result.

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



The record of execution times and speedup values have been presented in the excel sheet in the same submission.

3. Explanation of default scheduling policy of OpenMP. Provide a reference of where you found it and briefly explain it.

Ans: With most OpenMP runtimes, including GCC, the default scheduling when no schedule clause is present is a static policy with a chunk size equal to the number of iterations divided by the number of threads. The chunks are distributed among thread at the start of loop execution. This default policy aims to balance the workload evenly across threads while minimizing overhead. If there are any spare iterations (i.e #iterations % #threads != 0) then they are handed out one each to the first #remainder threads. [1] I found this by searching for a more precise way to point out the default policy by checking the implementation of the OpenMP '*def-sched-var'* internal control variable (ICV). [4] To do so, I echoed the OpenMP version as follows. `$ echo |cpp -fopenmp -dM |grep -i open` The terminal output showed `#define _OPENMP 201511`. After referencing the docs for OpenMP 4.5 [2] (which was released in 11/2015), the ICV initial value for *def-sched-var* is said to be *'implementation defined'* and there are no ways to modify or retrieve the value for this ICV. However, it does note that the schedule should be efficient for cases where all iterations have approximately the same amount of work, which static scheduling achieves. Finally, I checked the GCC docs for the actual implementation. The version on Henry cluster is: **gcc (GCC) 11.3.1 20221121 (Red Hat 11.3.1-4)** which builds upon gcc-9.3.0's [3] last decision to implement static scheduling as the default policy. The clauses in the documentation that state the default policy as dynamic scheduling likely refer to the default OMP_SCHEDULE env var which overrides in runtime situations, not when no schedule clause is specified. This hypothesis was confirmed in [1].

References:

[1] https://stackoverflow.com/questions/61638747/default-scheduling-in-openmp-gcc-compiler
[2] OpenMP 4.5 Complete Specifications
[3] https://gcc.gnu.org/onlinedocs/gcc-9.3.0/libgomp.pdf
[4] http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

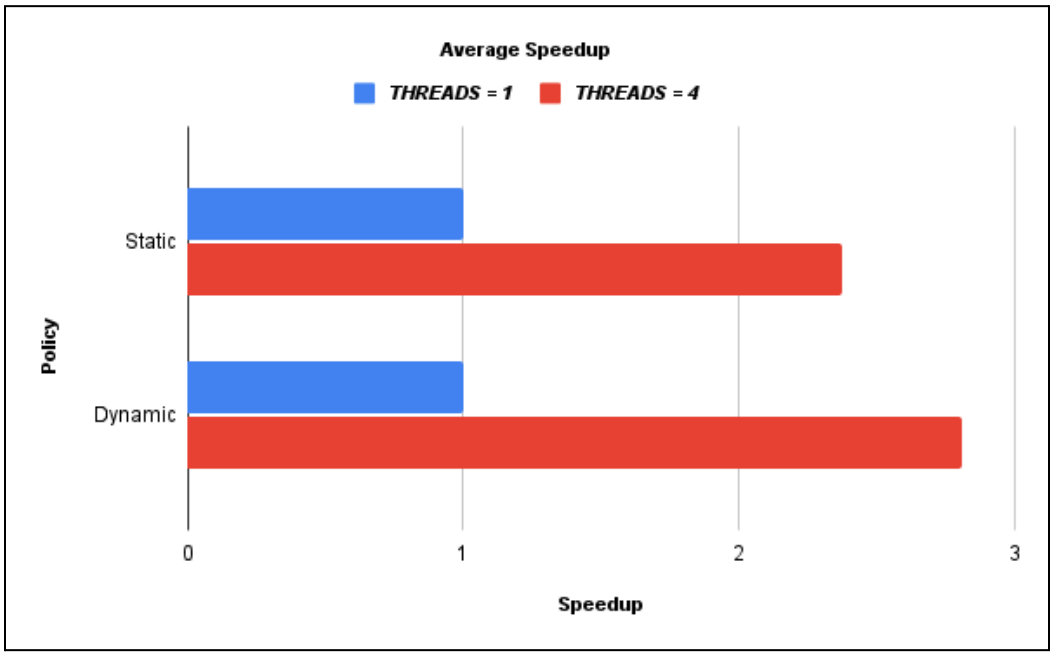
4. Bar graph of overall speedup of Static and Dynamic Scheduling policy with THREAD=4 relative to THREAD=1.

Ans: On the next page. Also present in the Excel sheet

5. Bar graph of speedup with the proposed optimization relative to THREAD=1. Provide a brief explanation. (only for CSC/ECE 506).

Ans: I managed to achieve a speedup of ~6.2652 relative to the serial execution of the program. I implemented a dynamic scheduling policy with chunk size = 16, and ran the program on 16 threads. I chose the dynamic scheduling policy to allow work to be distributed more evenly across threads. I tried various chunk sizes to get a good balance between fine-grained load balancing and reduced scheduling overhead. Among chunk sizes 10, 16, 18, 20, and 24, 16 yielded the best results across runs. I think it is likely that it also aligns well with the cache

(Task 4) Bar Graph of speedup for Static v/s Dynamic scheduling



sizes to get efficient usage of the cache. I initially thought that larger thread sizes (24/32) could better take advantage of the underlying 32 core compute node I was executing on. However, setting THREADS=16 sees a significant reduction in execution time. This could be a result of the reduced scheduling overhead and reduced synchronization costs (for the t_checksum parallel for). I didn't try parallelizing some other loops so I believe that a better speedup can be achieved, especially since the benchmark was specifically designed to test parallel performance. A speedup of ~6.26 using 16 threads isn't linear scaling in any world, but it's okay considering that only 6 of the 53 loops have been subjected to parallel processing.