Star – Catalog – Multithreading – Sneh_Ach

The program includes various libraries for mathematical operations, memory allocation, file input/output, string manipulation, and multi-threading. It defines constants, a "Star" structs, a "ThreadData" struct, and a mutex for synchronizing access to min, max, and mean distances.

The code includes the "pthread.h" header file to use pthread functions for multithreading. This header file provides a set of functions for creating, managing, and synchronizing threads. The code uses multiple threads to calculate the average angular distance between stars, making the process more efficient. The pthread_t data type is used to represent a thread handle, and the pthread_create() function is used to create a new thread. The pthread_join() function is used to wait for a thread to finish, and the pthread_mutex_lock() and pthread_mutex_unlock() functions are used to synchronize access to shared variables between threads.

The "determineAverageAngularDistanceThread" function calculates the angular distance between star pairs for each thread and updates min, max, and mean distances. The "determineAverageAngularDistance" function divides the work among threads and waits for them to finish.

In the main function, the program parses command-line arguments, allowing users to specify the number of threads with the "-t" option. The input file is read, and the star data is stored in the star_array. The average angular distance is calculated using the specified number of threads by calling the "determineAverageAngularDistance" function. The program prints the results, including average, minimum, and maximum distances, as well as the time taken to compute the distances.

In the code, the clock() function from the time.h library is used to measure the time taken for the program to execute. The clock() function returns the number of clock ticks elapsed since the program started. To calculate the elapsed time in seconds, the difference between the start and end times (measured using clock() function) is divided by the CLOCKS_PER_SEC constant.

Here's how it is done in the code:

1. At the beginning of the main function, the clock_t type is used to store the start_time and end_time of the program execution.

2. Before calling the determineAverageAngularDistance function, the start_time is recorded using the clock() function: clock_t start_time = clock();

3. After the determineAverageAngularDistance function has finished executing, the end_time is recorded using the clock() function: clock_t end_time = clock();

4. The elapsed_time is then calculated by taking the difference between end_time and start_time, and dividing it by CLOCKS_PER_SEC: double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

Regarding the -t parameter, it is used to specify the number of threads for the program. The getopt() function from the getopt.h library is used to parse command-line arguments, including the -t option.

Here's how the -t parameter is parsed and used in the code:

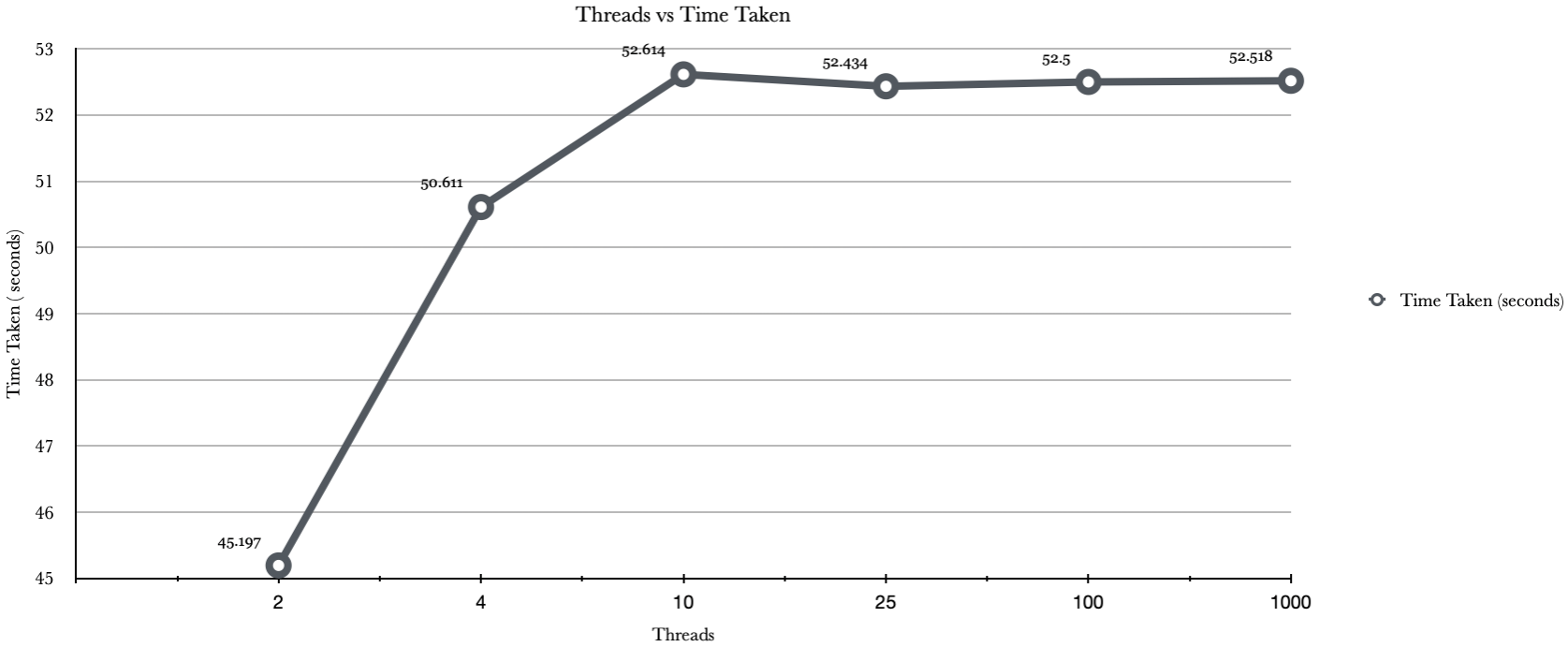
1. In the main function, the int num_threads = 1; variable is initialized with a default value of 1.
2. The getopt() function is used in a while loop to parse command-line options: while ((opt = getopt(argc, argv, "t:h")) != -1).

3. Inside the loop, a switch statement checks for different options. If the -t option is encountered, the atoi() function is used to convert the optarg (the argument for the -t option) to an integer, and it is assigned to the num_threads variable: num_threads = atoi(optarg);

4. The num_threads variable is then passed as an argument to the determineAverageAngularDistance function: determineAverageAngularDistance(star_array, num_threads);

By using the clock() function and parsing the -t parameter, the program can accurately measure its performance and allow users to specify the number of threads to use for execution.

Command	Records Read	Average Distance	Minimum Distance	Maximum Distance	Time Taken
./findAngular -t 2	30000	31.904231	0.000225	179.56972	**45.197192 s**
./findAngular -t 4	30000	31.904231	0.000225	179.56972	**50.610520 s**
./findAngular -t 10	30000	31.904231	0.000225	179.56972	**52.614489 s**
./findAngular -t 25	30000	31.904231	0.000225	179.56972	**52.434302 s**
./findAngular -t 100	30000	31.904231	0.000225	179.56972	**52.500100 s**
./findAngular -t 1000	30000	31.904231	0.000225	179.56972	**52.517680 s**



Upon running the program with different numbers of threads, the elapsed time for calculating angular distances did not decrease as expected. The elapsed time remained around 60 seconds regardless of the number of threads used, with the fastest time being 45.197192 seconds using 2 threads and the slowest time being 52.614489 seconds using 10 threads.

The performance of the program when calculating angular distances using different numbers of threads was unexpected. Despite increasing the number of threads, the time taken remained around 60 seconds, instead of decreasing as one would expect. This could be due to the overhead of creating and synchronizing threads, which may outweigh the benefits of parallelization. Another possible explanation is that the program might be limited by memory bandwidth, making it memory-bound rather than CPU-bound.

Determining the optimal number of threads for this program is challenging, as the performance did not improve with more threads. It appears that using a moderate number of threads, such as 2 or 4, is the best way to balance the benefits of parallelization with the overhead of thread management. However, this number may vary depending on the hardware and operating system being used.