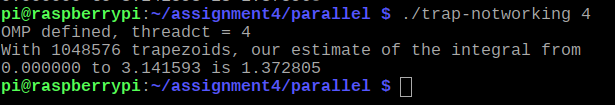
Parallel Programming Report

In the first part of the programming assignment, we are attempting to compute a Calculus value. Specifically, the “trapezoidal approximation of , using equal subdivisions.” First, I created a C file, named trap-notworking by typing the command “nano trap-notworking.c”. Then, I copy-pasted the code which was provided. After exiting from nano editor and typing the command, “gcc trap-notworking.c -o trap-notwroking -fopenmp -lm”, it created an executable program “trap-notworking”.

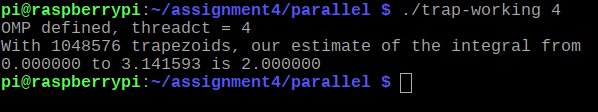
After this, I followed the same procedure for the second code, which was provided also. This time, I changed the name of the file to “trap-working.c” as it was instructed. Then copy-pasted the second code and created an executable program named “trap-working” for the C file by typing “gcc trap-working.c -o trap-working -fopenmp -lm”.

Finally, to run and compare the two programs, I typed “./trap-notworking 4” and “./trap-working 4” one after another. As the raspberry pi already has a four-core processor, it is natural to try four threads.

Picture #1: trap-notworking



Picture #2: trap-working



Both gave us different results after execution. The difference between the two programs was on line 37 and line 38. The program “trap-notworking”.

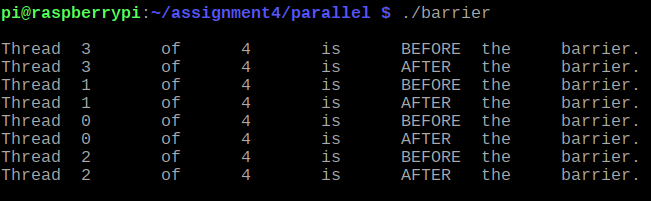
Both programs were created to find the definite integral of sin(x) from 0 to π. It is also stated that the answer should be 2.00. This is correct, but if only one thread is used to calculate the result, it gave the correct output while we edited out code again to run it on multiple threads; it gave us the wrong output. As we can see in picture #1, where “trap-notworking” program gave generated an output of 1.37, which is not the same if we re-run the program again. Every time we run it gives a different value but not the correct one.

On the other hand, “trap-working” gave us the correct value of 2.00. The line of code, “pragma omp parallel private (i) shared (a, n, h, integral), on program “trap-notwroking”, created a data race. Due to this, the “integral” is where the summation occurred, and it happened in the same memory space. On the second program “trap-working”, we added an extra directive, which is “reduction (+: integral)”. It avoided the data race. It gave each thread its own memory, and then it added it all up to the integral variable on the main thread. Even if we run the code with any number of threads, the output gave us 2.00 in picture #2 as a result, which was correct.

For the second part of the assignment, we used the barrier pattern. Which is used in parallel programming to ensure that before execution continues, all threads complete a parallel section of the code. When threads are generating computed data, this could be necessary.

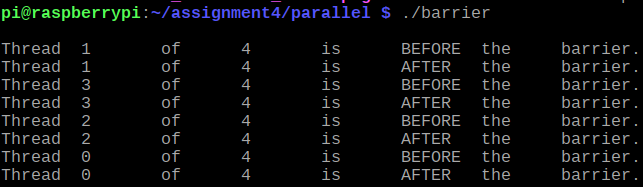
First, I copy-pasted the provided code on a new file named “barrier.c”. Then created an executable program for the file by typing the command, “gcc barrier.c -o barrier -fopenmp”. It generated an executable named “barrier”, then we run the program to see the output.

Picture #3: barrier



Now, we re-compile and re-run the code again, but this time without the commented pragma line “pragma omp barrier”.

Picture #4: barrier

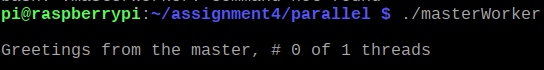


In this program, our main goal was to use the directive, “#pragma omp barrier”. When the directive was not used or commented out, the statements were printed in random order, which can be seen in picture #3. But when we used the directive or uncommented it, the statements printed were much more organized, which can be seen in picture #4. It started from thread 1, and each of the threads printed a before and after statement.

For the final part of this assignment, we will use the Master-Worker strategy. Here, one thread will be called the master, which execute one block of the code when it forks and the other threads called workers will execute different parts of the code when they fork.

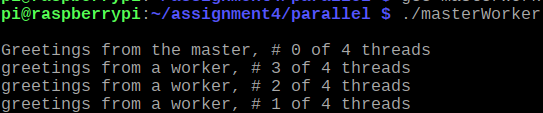
To demonstrate this method, we created a file named “masterWorker.c” and copy-pasted the code provided. Then compiled it by typing “gcc masterWorker.c -o masterWorker -fopenmp”. Which gave us an executable “masterWoeker” to run.

Picture #5: masterWorker



Now, we re-compile and re-run the “masterWorker.c” file, but this time we uncomment the pragma directive “#pragma omp parallel”.

Picture #6: masterWorker



The last part of the assignment was to demonstrate how the master and worker strategy works on parallel programming. Here the master thread runs a different section of code, and workers run the other section. When we commented out the directive, “pragma omp parallel” it only printed the master thread as it was shown on picture #5. Editing the code again and using the pragma directive this time the program gave us a different output. The master creates worker and sends initial values to workers. Pragma limits the execution of a block to a single thread. The master thread would remain the same if the “id == 0”. It would print, “Greetings from the master, # 0 of 1 threads”. The rest would print similar statements, but the word “master” would be replaced by “worker”, as shown in picture#6.