Developing Soft and Parallel Programming Skills Using Project-Based Learning

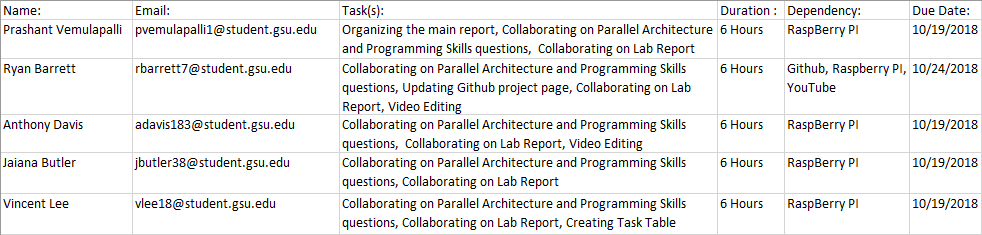
Project Report

Fall-2018

Submitted By PAJ\_VR

Prashant Vemulapalli, Anthony Davis, Jaiana Butler, Vincent Lee, Ryan Barrett

**Planning and Scheduling**



**Parallel Programming Skills**

*Define the following:*

Task: A task is the program or the set of instructions that are executed by a processor, different processors running many different tasks for a program is called parallel programming.

Pipelining: Pipelining is a technique utilized by parallel processing, where multiple processors handle different instructions that all belong to the same task.

Shared Memory: Shared memory is a feature of computer architecture where different processors can access the same physical memory through buses, and where tasks can access logical memory through addresses.

Communications: Communications refer to the various methods in which different tasks in parallel programing use to exchange data with each other.

Synchronization: Synchronization is a parallel processing method where different tasks use communications to coordinate.

*Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them:*

Single Instruction, Single Data (SISD): The CPU only executes one stream of instructions and reads from one stream of data during a single clock cycle. Uses single processor architecture.

Single Instruction, Multiple Data (SIMD): Has multiple processors and each processor executes the same instruction per clock cycle, but each processor can also process a different piece of data per clock cycle.

Multiple Instruction, Single Data (MISD): Has multiple processors with each processor getting input from a single data stream, the data is processed using a separate instruction stream for each processor.

Multiple Instruction, Multiple Data (MIMD): Has multiple processors and each processor may be working on both a separate instruction stream, and a separate data stream.

*What are the parallel programming models?*

Process interaction: shared memory, message passing, Implicit interaction

Problem decomposition: Task parallelism, Data parallelism, implicit parallelism

*List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?*

Uniform Access Memory (UMA): A type of Parallel Computer Memory Architecture in which separate but identical processors can all access the same memory, and all processors will automatically be aware of any changes made to and address in the memory by any single processor; this processor awareness feature is called cache coherency. Each processor can access the memory in the same amount of time and have access to the whole memory. The most common type of machines that uses this type of memory architecture are symmetric multiprocessor machines, or SMP machines.

Non-Uniform Access Memory (NUMA): A type of parallel computer memory architecture in which multiple SMP machines are connected through buses and can access each other’s memories. In NUMA, the access times of different processors may vary, and the speed at which memory is accessed may be slower. NUMA architectures may also utilize cache coherency.

OpenMP, being a tool for parallel programming, uses Uniform Access Memory.

*Compare the shared memory model with the threads model.*

In shared memory model processes and tasks share a common address space where they read and write asynchronously. Various mechanisms are used to control access to the shared memory, and prevent issues.

The advantage of this model is that there is no data “ownership” so there is no need to specify explicitly the communication of data between tasks.

The disadvantage is it becomes more difficult to understand and manage data locality.

In the threads model each thread has its own local data as well as shared data. Threads update each other through global memory which requires synchronization to make sure more than one thread is not updating the same global address at any time.

One of the main differences between shared memory model and threads model is in memory model processes and tasks share memory they read and write to asynchronously while in the threads model each thread interacts with shared memory locations synchronously.

*What is parallel programming?*

Parallel programming is when we use multiple resources at the same time working together to solve a single problem or complete a calculation.

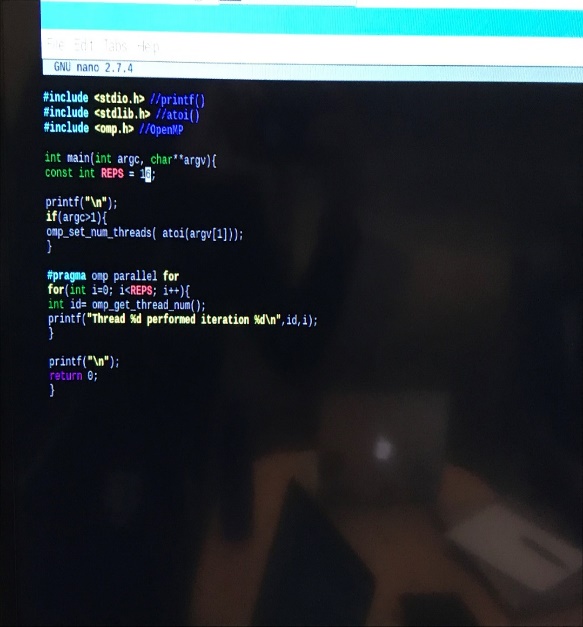
*What is system on chip (SoC)? Does Raspberry PI use system on chip?*

System on chip is an integrated circuit that integrates all the components of a complete computer onto a single chip. Raspberry PI does use system on chip. The CPU, RAM, and GPU are not separated, but on one chip.

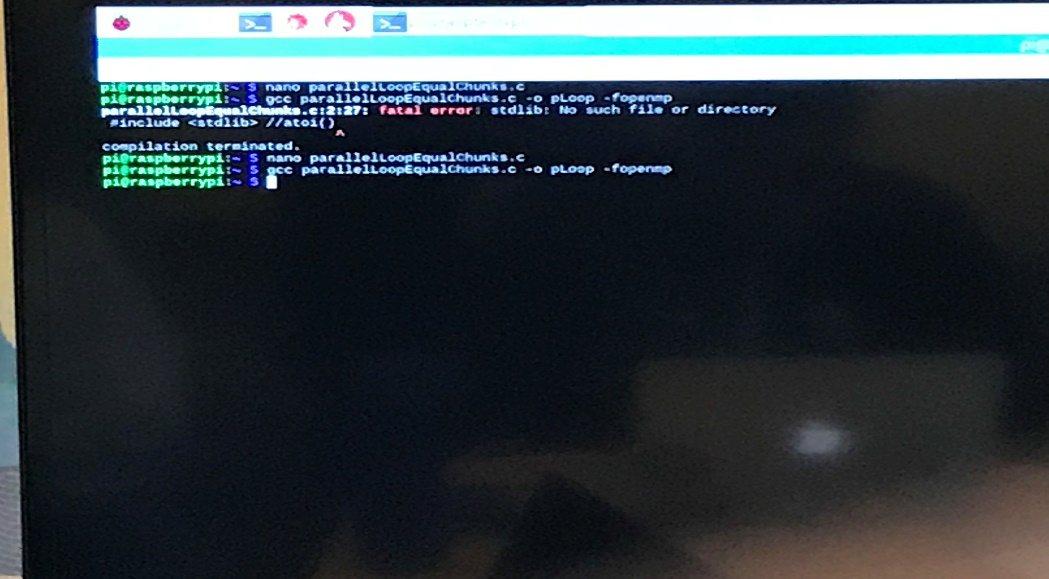
*Explain what the advantages are of having a system on chip than separate CPU, GPU and RAM components.*

A SoC integrates all of the components on one chip. Can do all the functions of a CPU, GPU, and RAM components but contains a lot more functionality. SoC allows us to have computers that fit into the palm of our hands. SoC uses less power to function since its has a high level of integration and much shorter wiring. SoC makes building computers cheaper because of cutting down on the physical chips.

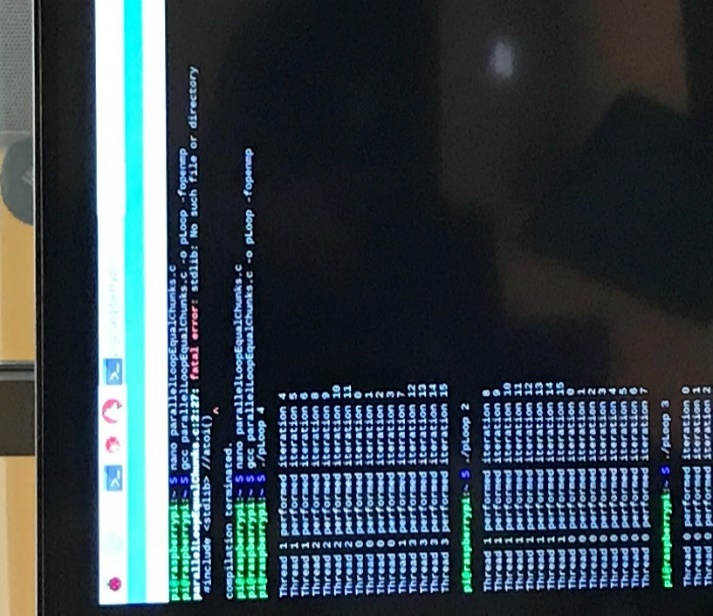
**Parallel Programming Basics**



Explanation: After entering the code “**nano parallelLoopEqualChunks.c”** the code for the parallelLoopEqualChunks program is written.

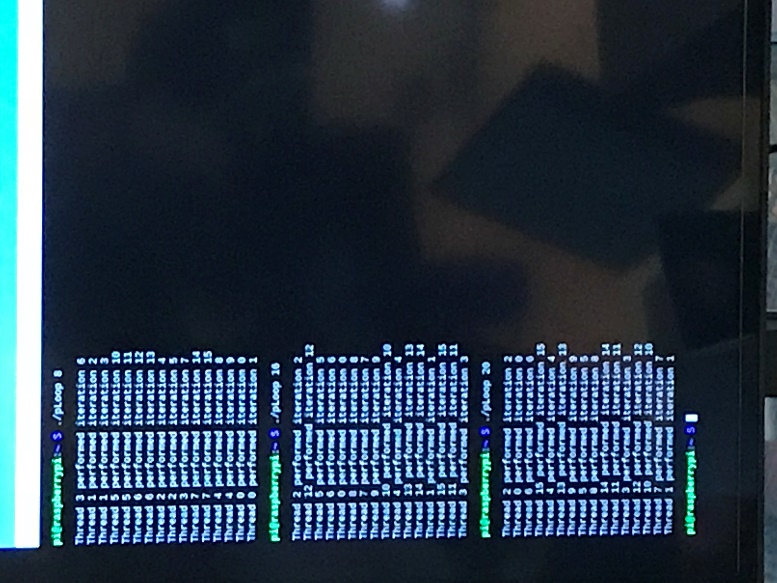


Explanation: After saving the code, and exiting the nano editor, the “**gcc parallelLoopEqualChunks.c -o pLoop -fopenmp**”is used to make the executable for the program and name it pLoop.

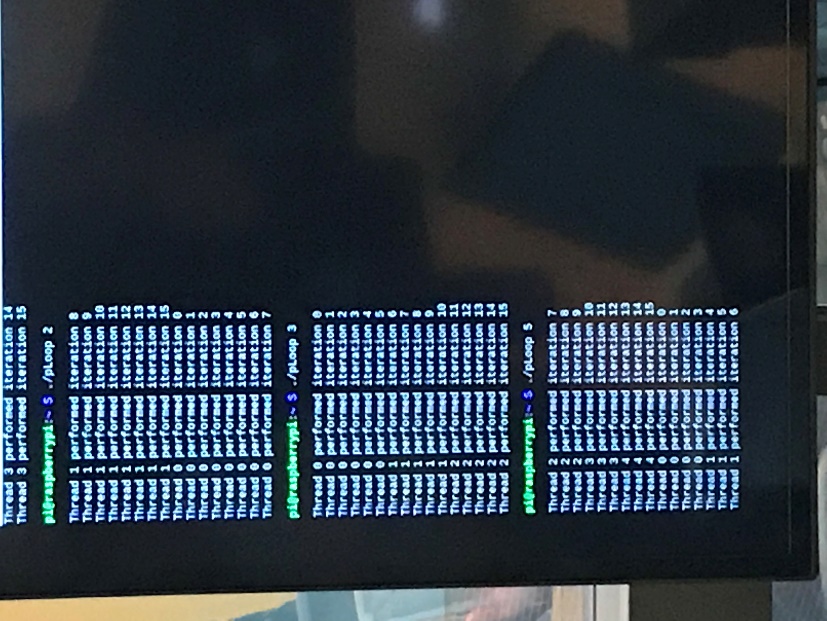


Explanation: We executed the program using the command **“./pLoop 4**”, and tested different values for the program.

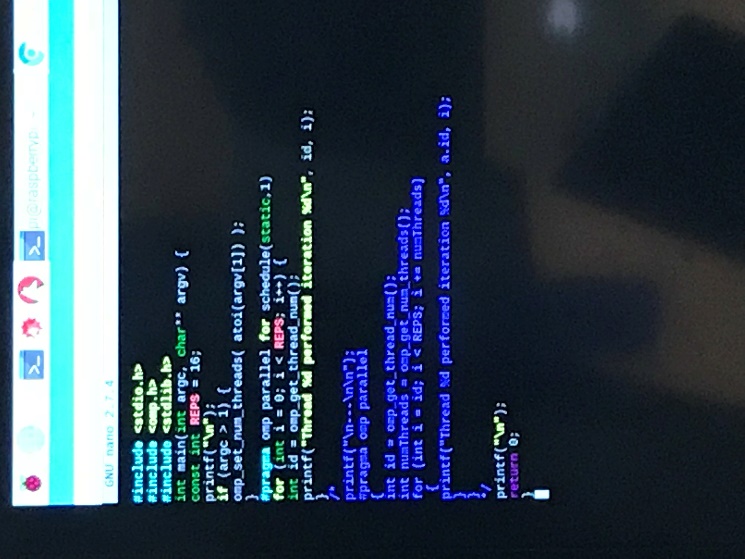
Observation: The #pragma omp parallel directive is forking the for loop to return the thread id, and the iteration number. Each thread is preforming one separate iteration, and some iterations are not in order, because different threads may complete at different times.



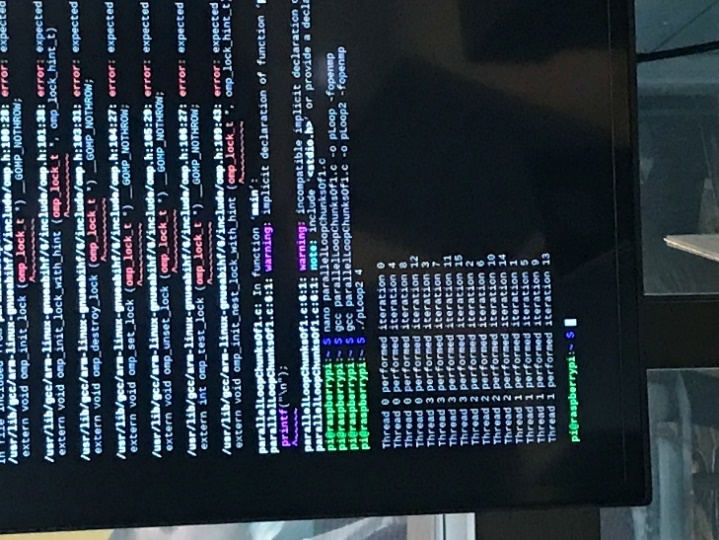
Explanation: Continuing to test different values for the pLoop program.



Explanation: Continuing to test different values for the pLoop program.

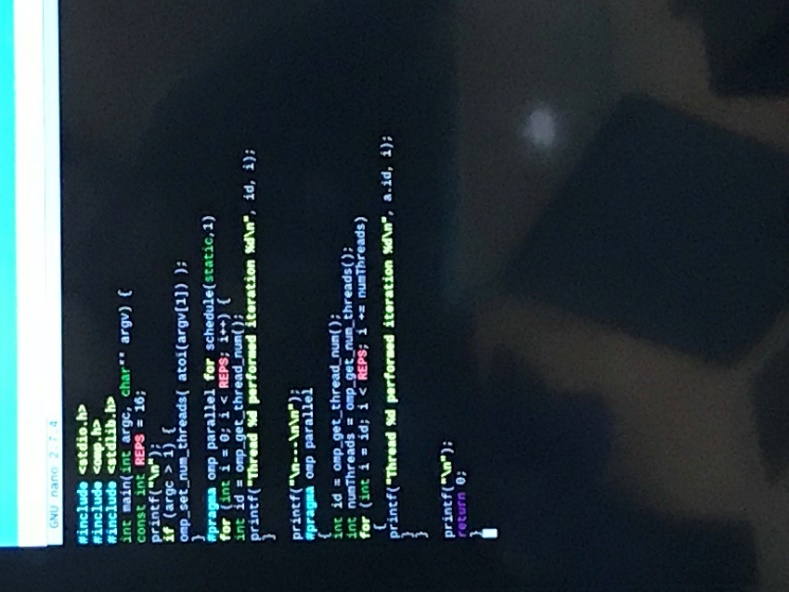


Explanation: After entering the code “**nano parallelLoopChunksOf1.c”** the code for the parallelLoopChunksOf1 program is written.

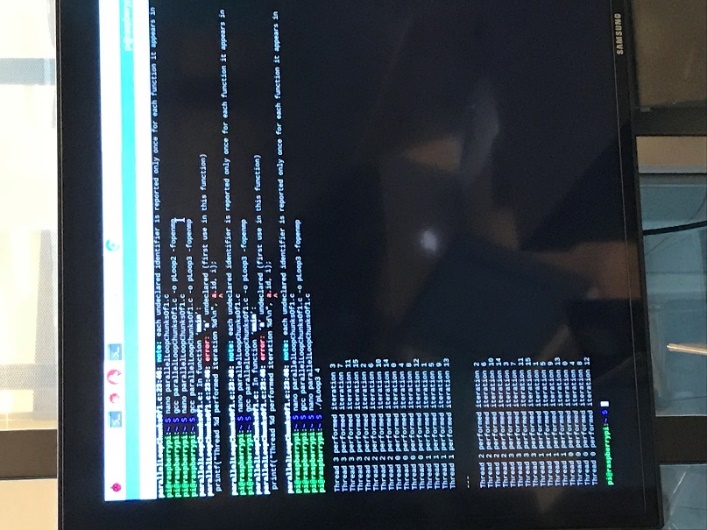


Explanation: After saving the code and exiting the nano editor, the code “**gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp**” is written to create the code and name it as pLoop2. The program is executed by the code “**./pLoop2 4**”.

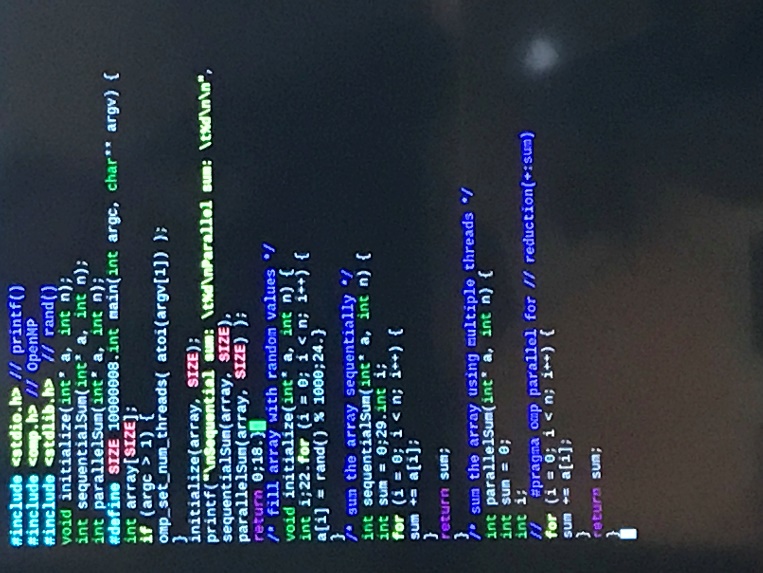
Observation(s): Each thread still preform separate interactions, but now the iterations are in increments of 4 due to the static keyword specifying the pattern for which the threads are to follow.



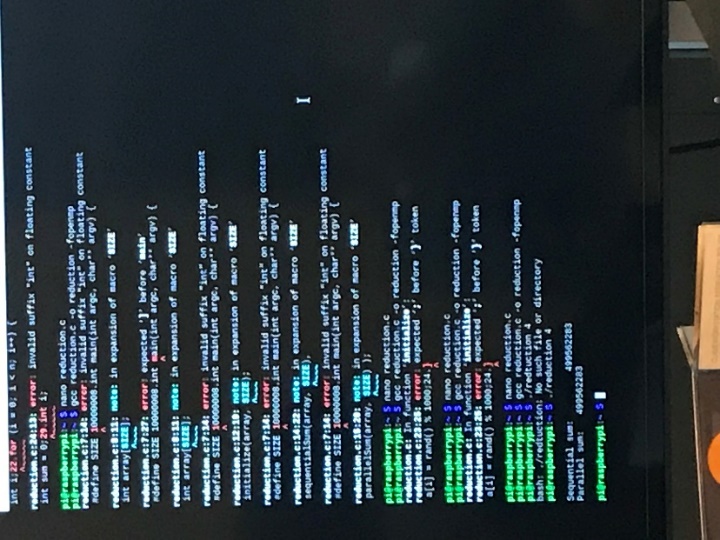
Explanation: We entered the nano editor again and uncommented the code that was originally commented.



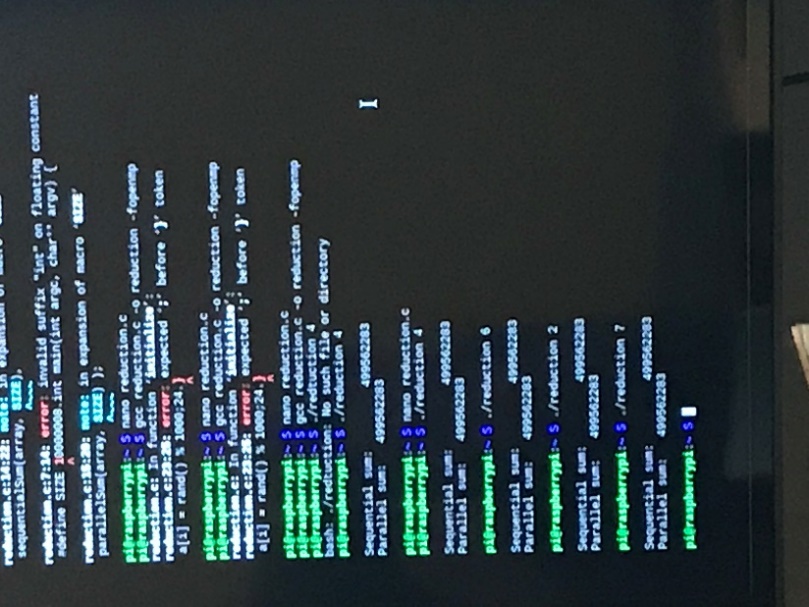
Explanation: After svaing the code and exiting the nano editor, for the version of the code with the uncommented section, we created the executable and named it as pLoop3 using the command “**gcc parallelLoopChunksOf1.c -o pLoop3 -fopenmp**”. Then we tested values for it using “**./pLoop3 4**”.



Explanation: We entered the nano editor using the code “**nano reduction.c**” and then wrote the code for the program.

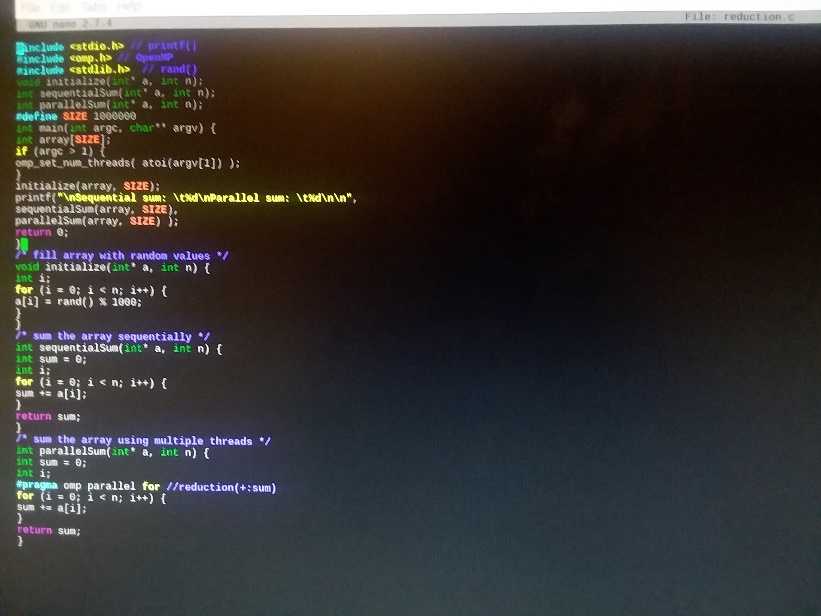


Explanation: After saving the code and exiting the nano editor, we tested the program using the code “**./reduction 4**”.

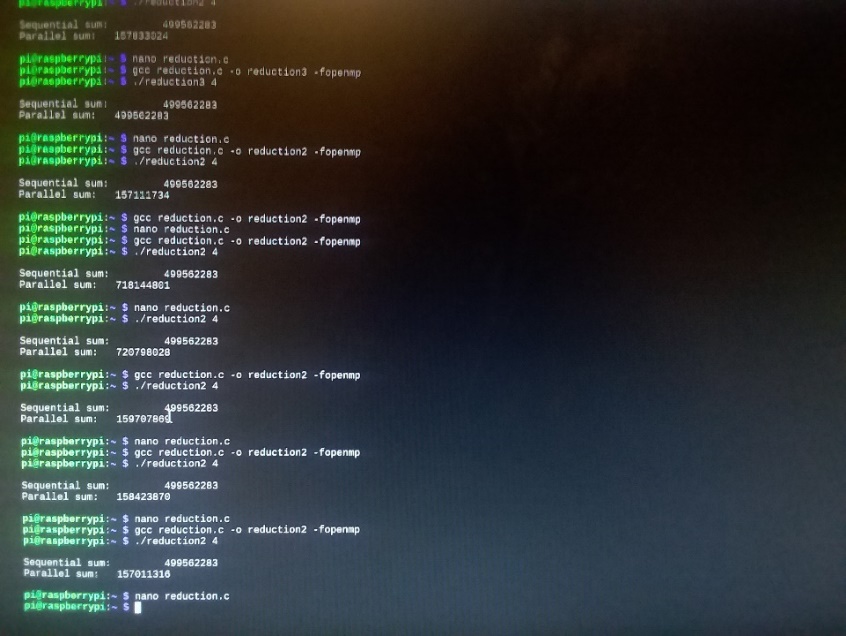


Explanation: Testing different values for the program.

Observation(s): Because of the commented out portion of the code, it seems that the parallel sum is the same as the sequential sum, due to the part of the code that should fork the process is commented out. The code for the parallel sum function is the same as the sequential sum function, so without the #pragma omp parallel directive, it just sums the values sequentially instead of in parallel.

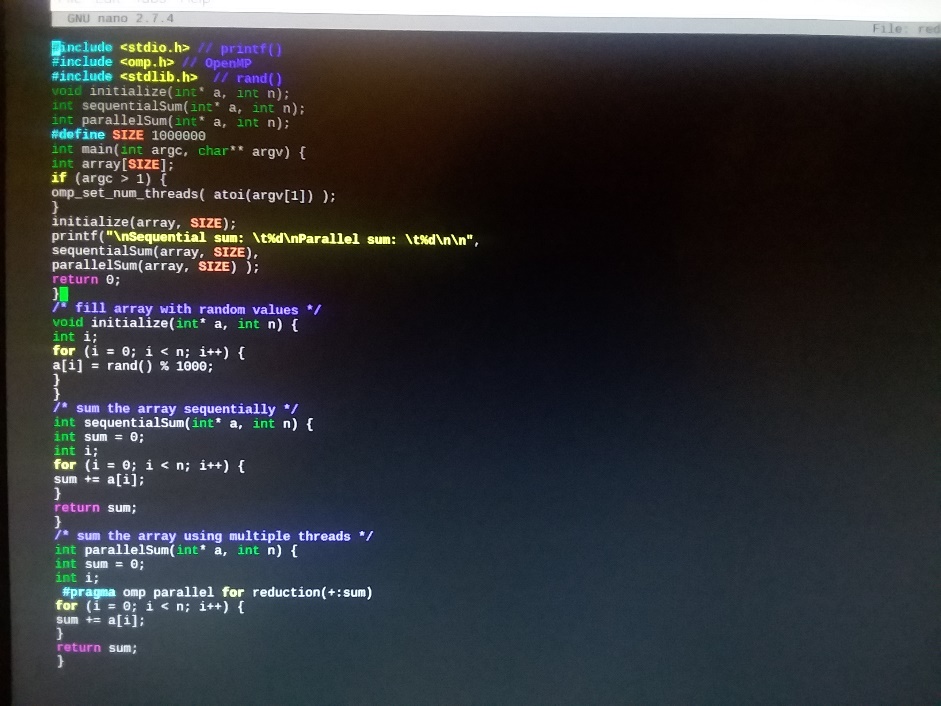


Explanation: We went back into the nano editor and uncommented the code snippit “**#pragma omp parallel for**”.

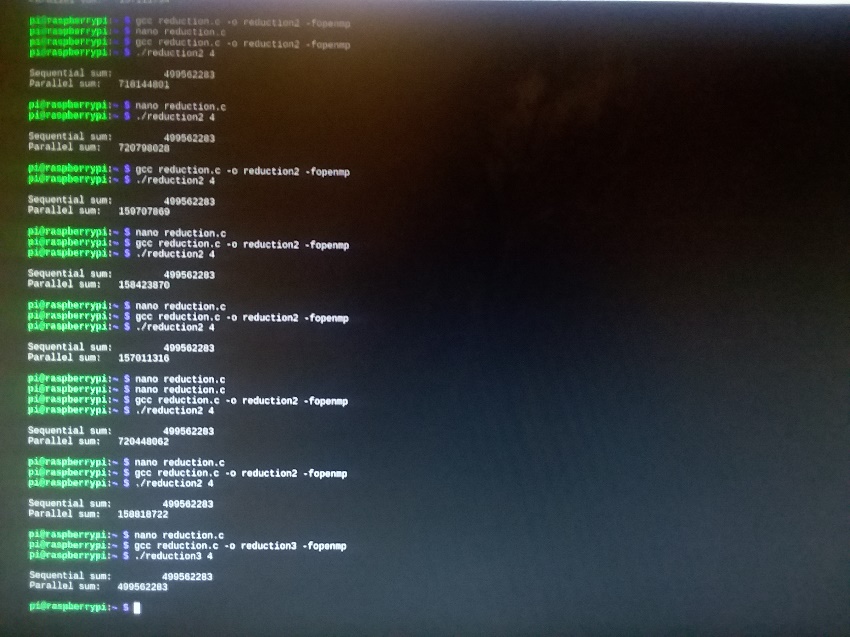
****

Explanation: after creating the new executable and naming it “reduction 2” using the command “**gcc reduction.c -o reduction2 -fopenmp**” we test the executable with “.**/reduction2 4**”

Observation: The values for the parallel sum are now different because of the portion of code that is commented back in, as it uses the #pragma omp parallel directive the fork the portion of the code that returns the parallel sum, but the values are incorrect due to the last part of commented code.



Explanation: We went back into nano, and uncommented the code snippet “**reduction(+:sum)**”.



Explanation: After saving the code, we exited nano and created the executable, naming it reduction 3, then we tested it using the command **“./reduction3 4**”.

Observation(s): The parallel sum is once again the same as the sequential sum, due to the reduction variable. The parallel sum function now sums the values using multiple threads instead of sequentially.

**Appendix**

Video: https://youtu.be/GlFktgJx0eE

Slack: pajvr.slack.com

Github: https://github.com/PAJVR/Project\_3

Github project page screenshot:

