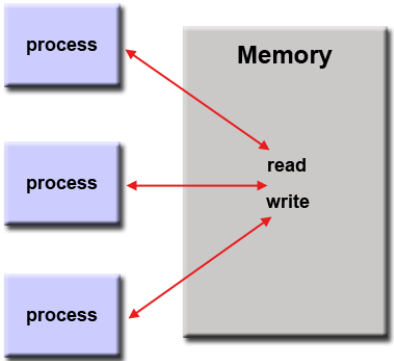
Zoe Kosmicki

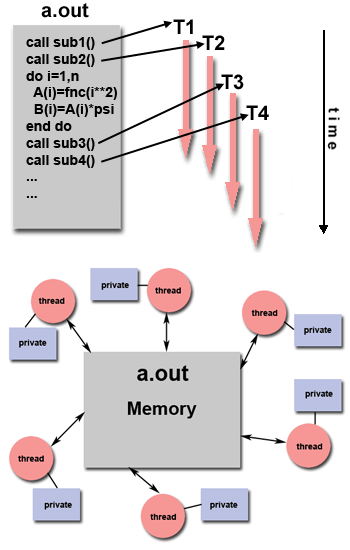
Assignment 3

Task 3

Part 1: Foundation

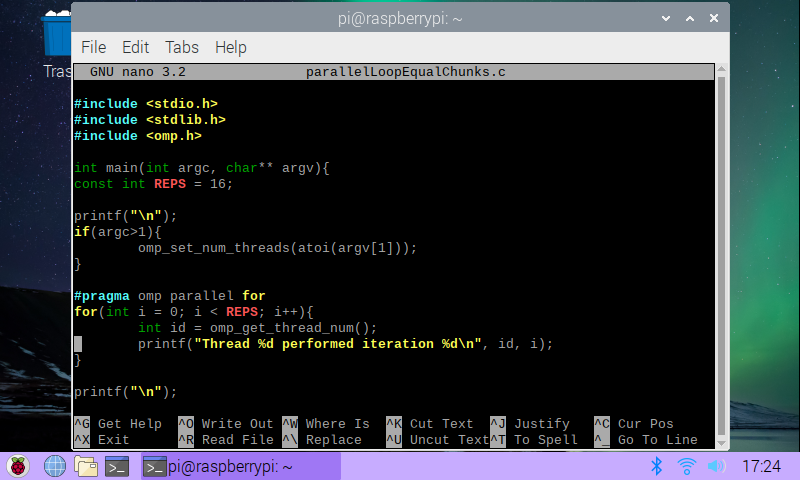
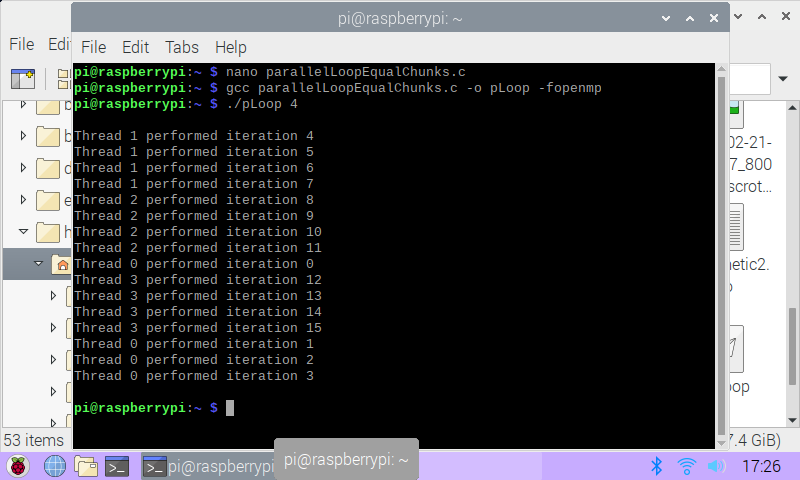
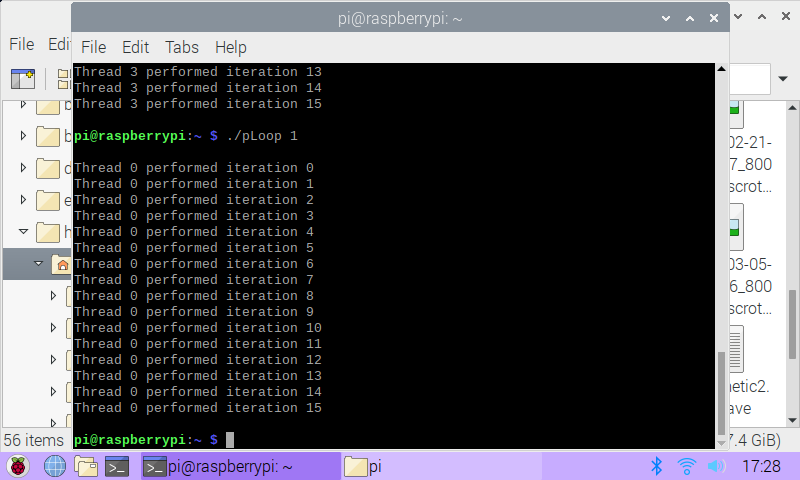
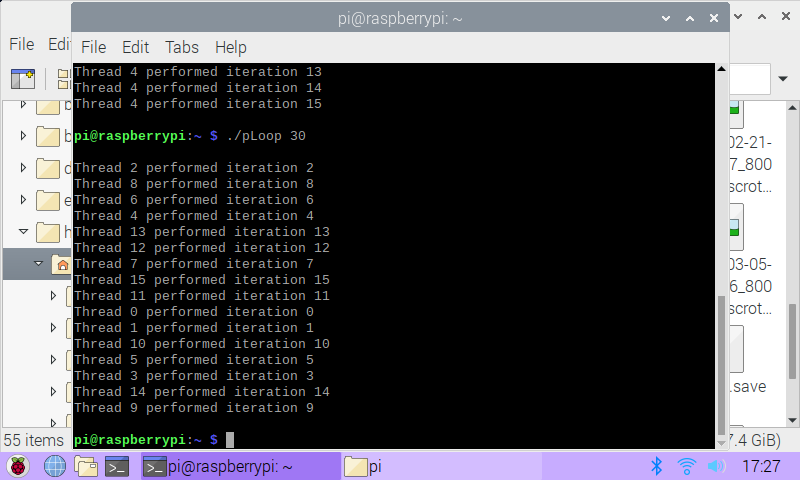
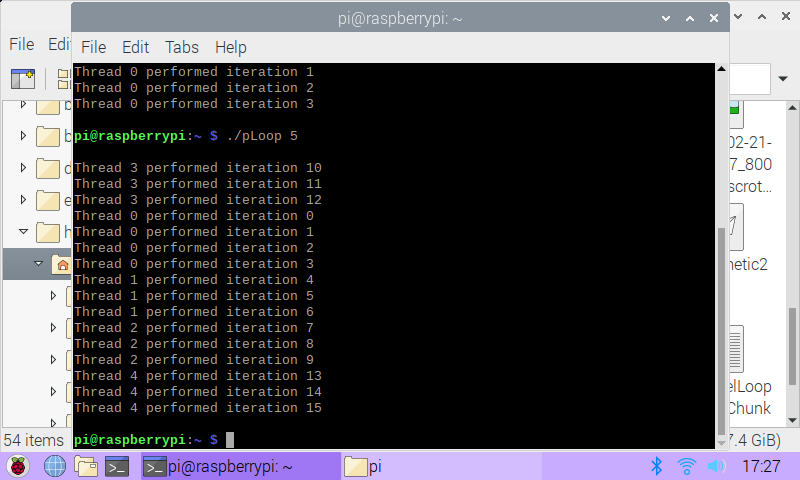
* *Define the following:*
  + Task: a set of instructions that are generally organized like a program, to be executed by a processor.
  + Pipelining: the way a program is broken down by various processors to be executed in a parallel fashion. Comparable to an assembly line.
  + Shared Memory: a type of computer architecture that lets all processors have direct access to memory. This isn’t dependent on where the physical memory is located; each processor has the same access to the logical memory.
  + Communications: how parallel tasks “talk” to each other as they are being executed.
  + Synchronization: how a parallel program coordinates with itself as it’s running.
* *Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.*
  + SISD (Single Instruction stream; Single Data stream): SISD computers are serial, so they can only process and execute one data/instruction stream at a time. Generally now only found in older hardware.
  + SIMD (Single Instruction stream; Multiple Data stream): Parallel computer that can execute the same instruction over various data threads at the same time. The two main varieties are processor arrays and vector pipelines.
  + MISD (Multiple Instruction stream; Single Data stream): A type of parallel computer that executes more than one set of instructions on one stream of data at the same time. This is the most common type of parallel computer currently.
  + MIMD (Multiple Instruction stream; Multiple Data stream): A parallel computer that executes multiple instruction and data streams at the same time. This type of computer makes up most modern supercomputers and network clusters.
* *What are the Parallel Programming Models ?*
  + Shared memory, threads, distributed memory, data parallel, hybrid, SPMD, and MPMD
* *List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?*
  + UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access) are the types of Parallel Computer Memory Architectures. UMA machines have one shared general memory that multiple CPUs have access to, while NUMA machines have several memories connected to several CPUs, which are all interconnected via buses. OpenMP can use both, since it is built for shared memory machines.
* *Compare Shared Memory Model with Threads Model? (in your own words and show*

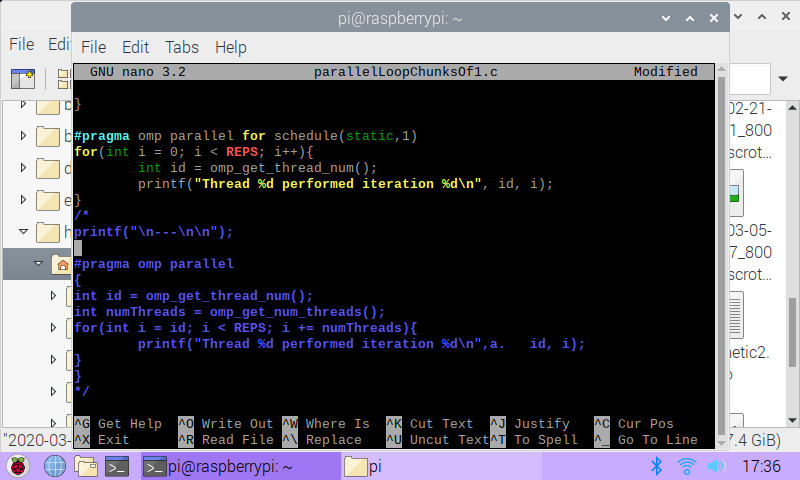
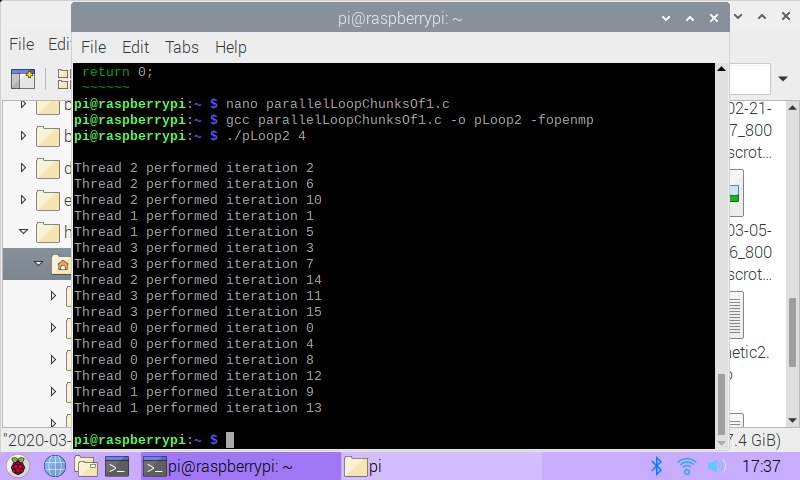
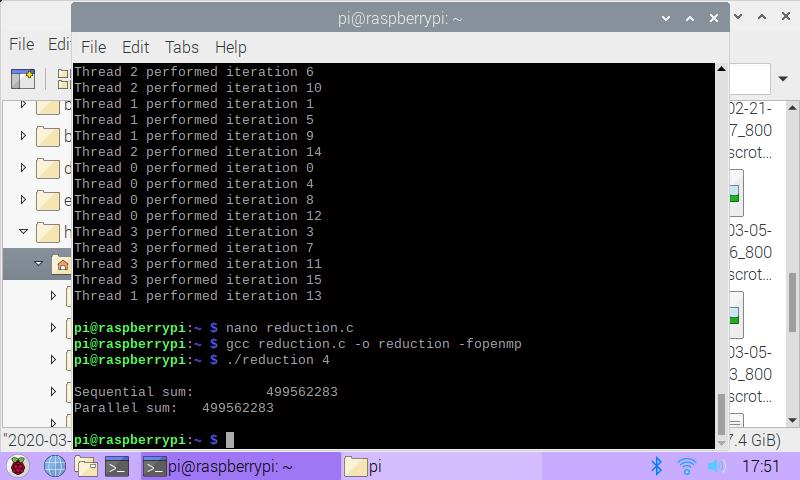
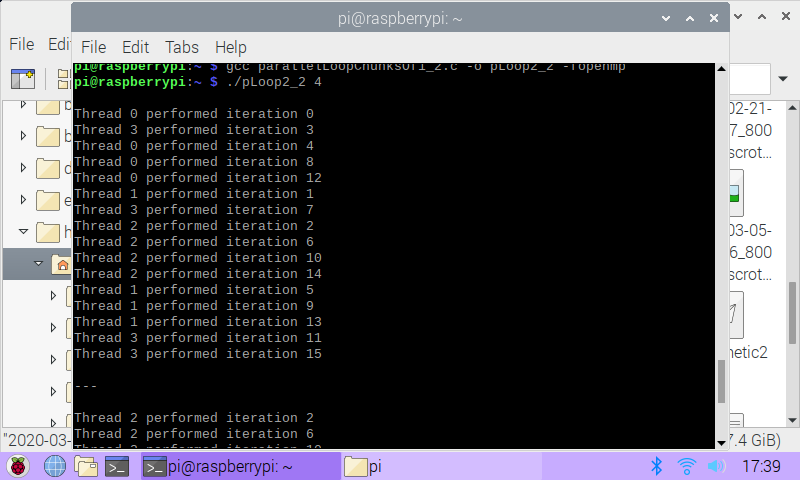
*pictures)*

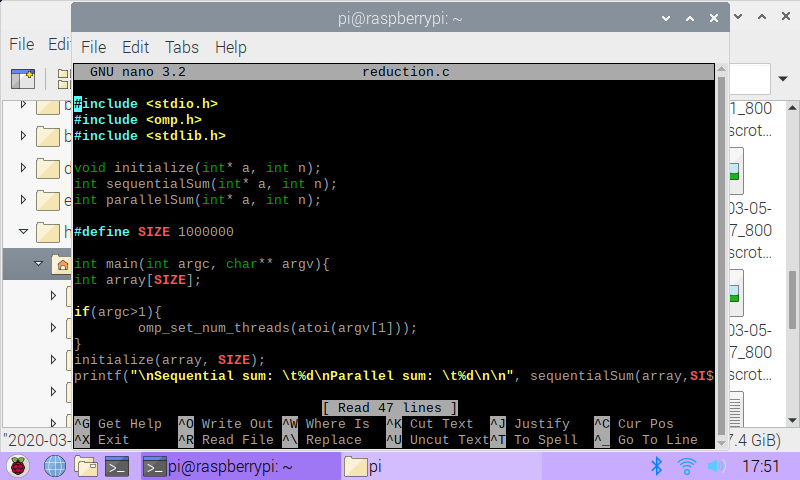
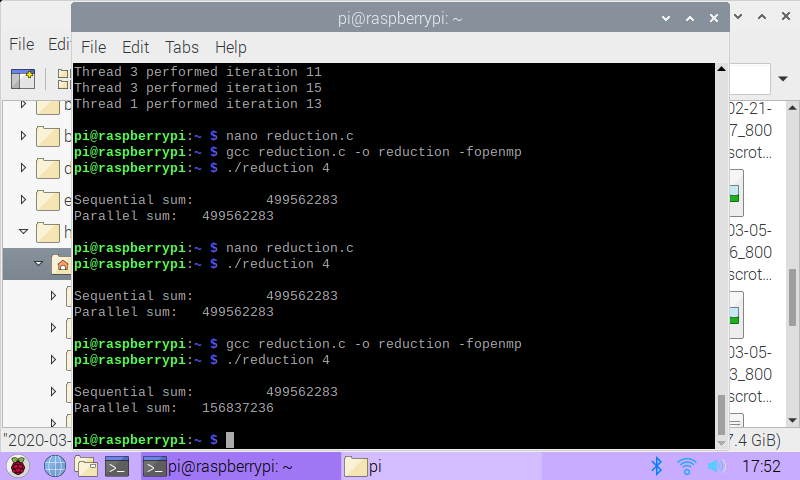
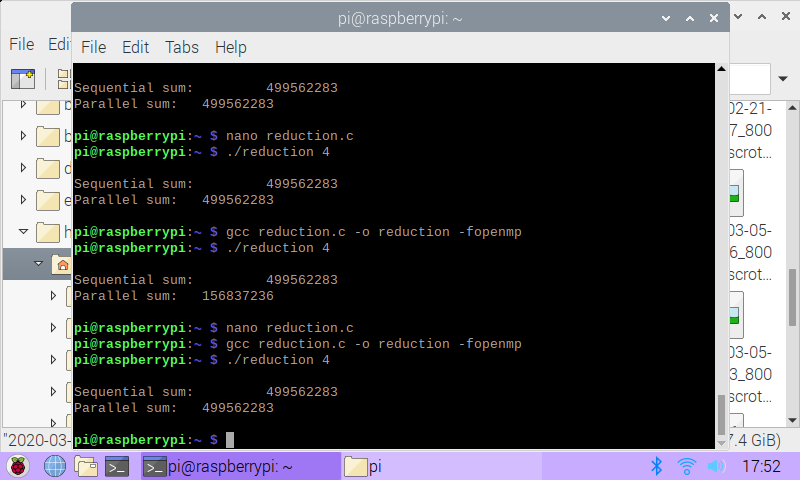
* + Shared Memory Model: the memory is a shared space, with each individual process able to read and write to shared addresses within the memory.
  + Threads Model: a program has its main body that allocates resources to run the program in full. After the resources have been found and set aside, the main body creates several threads/tasks that operate independently and in parallel. These threads all operate within a shared memory, so they can all see what the other threads are doing and have access to the resources other threads might be using.
    - These models are both forms of parallel processing, and they both utilize a shared memory space. However, the shared memory model is much simpler than the threads model. The shared memory model consists of just a shared memory space that several processes share, while the threads model utilizes a shared memory allocated by a parent process, and only accessed by threads from within the parent process.
* *What is Parallel Programming? (in your own words)*
  + Programming that takes advantage of a computer’s ability to complete tasks in parallel with each other rather than just serially.
* *What is system on chip (SoC)?*
  + A type of computer that has the CPU, GPU, and RAM put onto a single chip, instead of all of those components separately placed
* *Does Raspberry PI use SoC?*
  + Yes.
* *Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.*
  + SoCs are smaller than having separate components, so they are better for smaller devices as well as are cheaper to make due to its size. They also generally use less power, since wired information processing is literally/physically shorter compared to a PC motherboard.

Part 2: Parallel Programming Basics

I began by writing out the code for parallelLoopEqualChunks.c in the nano editor.

After creating an executable, I ran the program using ./pLoop 4 and below is my output.I then tried a few other numbers of threads to fork to see what the output would look like.To me, it seemed like if there are enough threads (at least 16), then the iteration would pair to it’s similarly numbered thread. Otherwise, there would be a sectioning of threads that would attempt to partition each iteration as cleanly as possible.

Next, I wrote down the code for parallelLoopChunksOf1.c in the nano editor.I first ran pLoop2 with the pragma section commented out, just to make sure it ran fine. Below is the output I got.After running that, I uncommented the pragma section and renamed the executable pLoop2\_2, just to keep track of it. Then I ran it with the command ./pLoop2\_2 4 and below is the output.Both sections of code give equal outputs, just in different orders. The second chunk of code seems more flexible than the first, because in the for loop, i is set to id instead of starting at 0, meaning it’s easier to adapt the code to a different number of threads.

Next, I wrote out the reduction.c program in the nano editor.After compiling and linking it, I ran it with the value 4. Both the sequential and parallel sums were the same, since the parallel portion was still commented out. This program creates an array of size 1,000,000 and fills it with random numbers and adds it up sequentially and parallelly. Next, I uncommented line 39, ran the program again, and then uncommented reduction(+:sum) and ran it once more and below are the outputs I got, respectively.Before the reduction(+:sum) clause was uncommented, the parallel and sequential sums were different. After adding the reduction clause back in, they went back to being the same value. I think they were different at first because without the reduction clause, the parallel processes weren’t communicating with each other correctly, with many different sum variables being calculated and not added together at the end. This would end up giving a different answer than the sequential sum would give.