**Parallel Programming Skills and Basics**

**Part 1: Parallel Programming Skills**

* **(5p) Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)** 
  + Task:

A program (or something like a program) set of instructions that the processors execute. In a parallel program, there are multiple tasks being executed by multiple processors at a given time.

* + Pipelining:

A type of parallel computing that break down tasks into steps that will be executed by different processors

* + Shared Memory:

From a hardware perspective, it is a computer architecture where every processor has direct access to a common memory using the bus. From a programming perspective, it is a model where parallel tasks have the same “picture” memory with the ability to directly address and access the memory regardless of its physical location.

* + Communications:

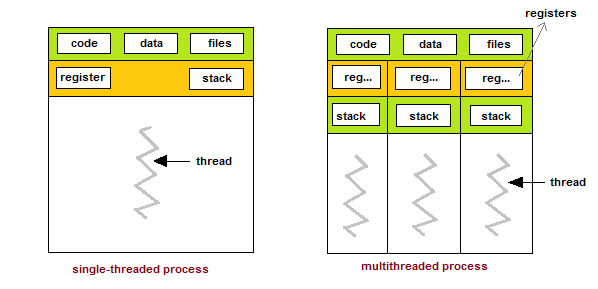
How parallel tasks communication with each other. Methods of communication includes network or a shared memory bus.

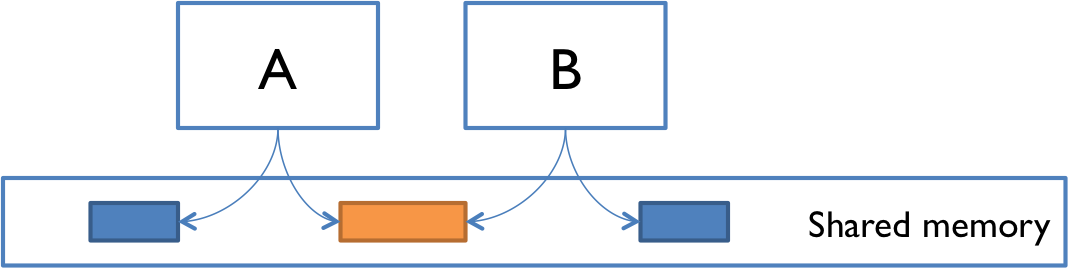
* + Synchronization:

How parallel tasks cooperates with each other, mostly through a form of communication. This is mostly implemented by making a synchronization point in an application that tasks a task to wait for other task(s) to finish before proceeding. This can cause the application’s wall clock execution time to increase.

* **(8p) Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.**
  + Single Instruction Single Data (SISD)
    - Serial computer
    - Only one instruction stream per one clock cycle
    - Only one input data stream per one clock cycle
    - Deterministic execution
    - Ancient computer
    - Examples includes older mainframe generations, minicomputers, workstations, and single processor PCs
  + Single Instruction Multiple Data (SIMD)
    - Parallel computer
    - All processors execute the same instruction at any given clock cycle
    - Each processor can execute different data elements
    - Most suitable for problems with high degree of regularity (like graphic/image processing)
    - Synchronous (lockstep) and deterministic execution
    - Two types: Processor Arrays and Vector Pipelines
    - Most modern computer (mostly those that has GPUs employ SIMD instructions and execution)
  + Multiple Instruction Single Data (MISD)
    - Parallel computer
    - Processors executes data independently by separate instruction streams
    - Single input data stream is fed to multiple processors
    - Computers rarely use this class type
    - Conceivable use may be multiple frequency filters working on a single signal stream, or multiple cryptography algorithms decrypting a code
  + Multiple Instruction Multiple Data (MIMD)
    - Parallel computer
    - Every processor can execute a different instruction stream
    - Every processor can take in different data streams
    - Most popular class of computers
    - Many MIMD also include SIMD execution sub-components
* **(7p) What are the Parallel Programming Models?**
  + Shared Memory (without threads)
  + Threads
  + Distributed Memory/message Passing
  + Data Parallel
  + Hybrid
  + Single Program Multiple Data (SPMD)
  + Multiple Program Multiple Data (MPMT)
* **(12p) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
* **(10p) Compare Shared Memory Model with Threads Model? (in your own words and show pictures)**

In the shared memory model, it does not use threads, and its tasks shares common memory address space that they read and write to asynchronously. To help prevent race conditions and dreadlocks and resolve contention, mechanisms like the locks/semaphores are used to control shared memory access. Shared memory model is the simplest parallel programming model. An advantage of using this model is that there is a lacking “ownership” of the data, which means that we do not to specify the communication of data between tasks since all processes see and have access to shared memory. However, using this model means that data locality will be harder to understand and control. The way how shared memory is implemented depends on the machines. On a stand-alone memory machine, the native operating systems, compilers, and/or hardware gives support for shared memory programming. On a distributed memory machine, the memory is made global by special hard and software.

In thread modeling, which is a type of shared memory programming, a “heavily weighted” task can have multiple “light weighted” concurrent threads. Each thread will have its own local data, but shares the entire resource of the program. This way, the program’s resources does not need to be copied for each thread. The threads will also have benefits of global memory view since it shared the memory space of the program. The threads will also use synchronization constructs to communicate with each other. Implementing threads mostly comprise of a library of subroutines called within parallel source codes and a set of compiler directives imbedded in the serial or parallel source code.

* **(5p) What is Parallel Programming? (in your own words)**

A model of computing where program tasks can be broken down into threads that are then processed by two or more processors.

* **(5p) What is system on chip (SoC)? Does Raspberry PI use system on SoC?**

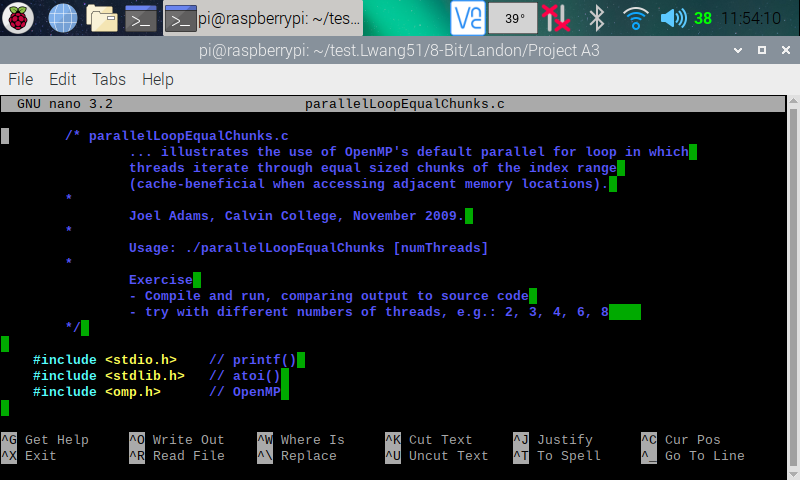
SoC is a computer that has almost all of the components that a CPU would have integrated into a single silicon chip. An SoC usually has GPU, memory, USB controller, power management circuits, and wireless radios. Raspberry Pi computer would be an example of a System on Chip computer.

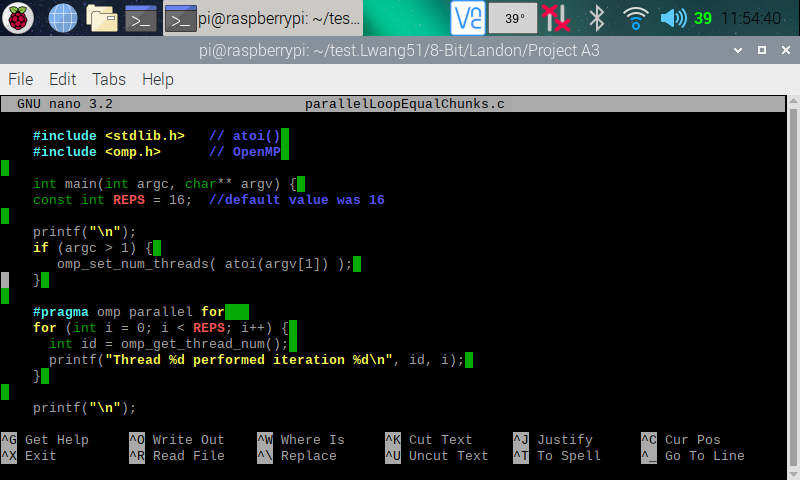
* **(5p) Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

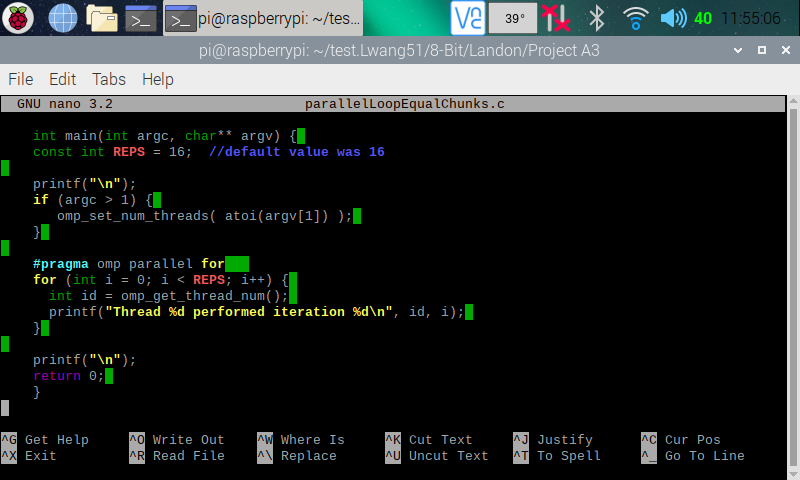
SoCs consumes less power, because it has a high level of integration, so it uses shorter wires. Furthermore, it will be cheaper to build, because far fewer physical chips are needed to.

**Part 2: Parallel Programming Basics**

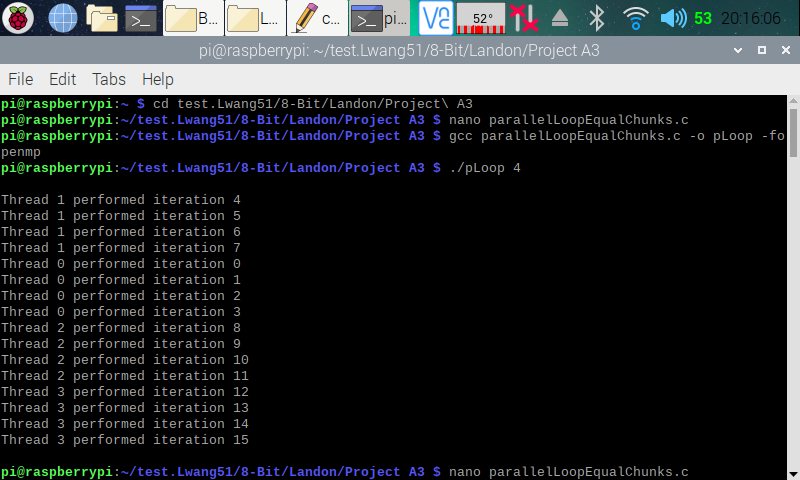
**Parallel Loop Equal Chunks Program**

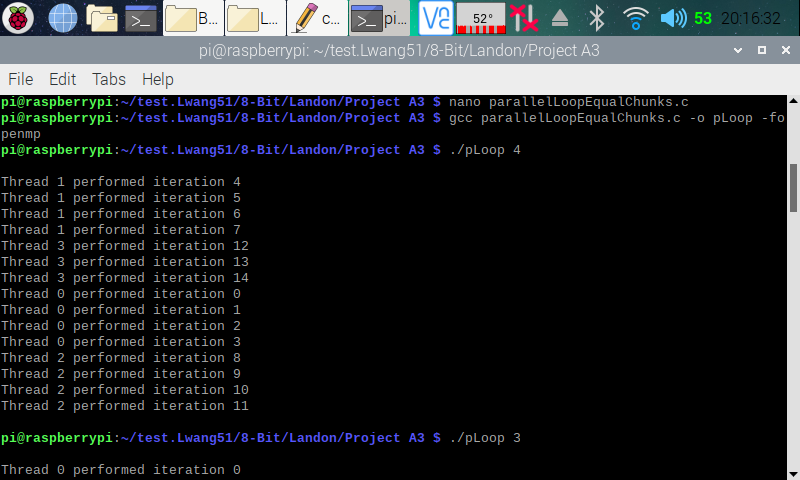




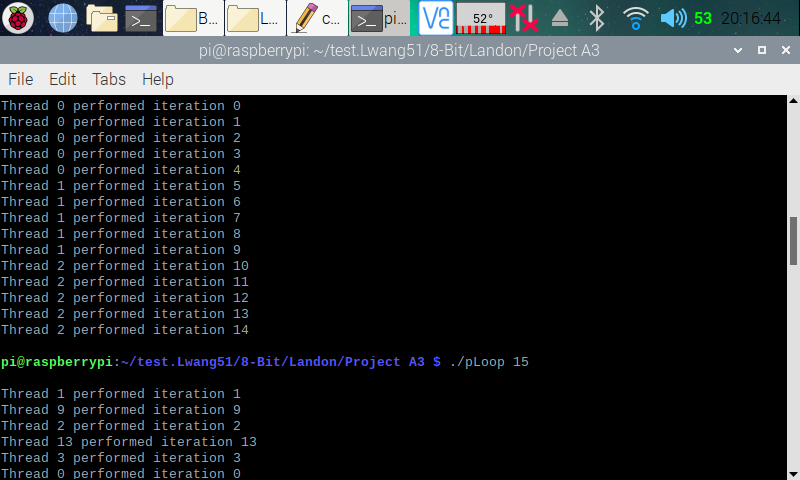


Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.

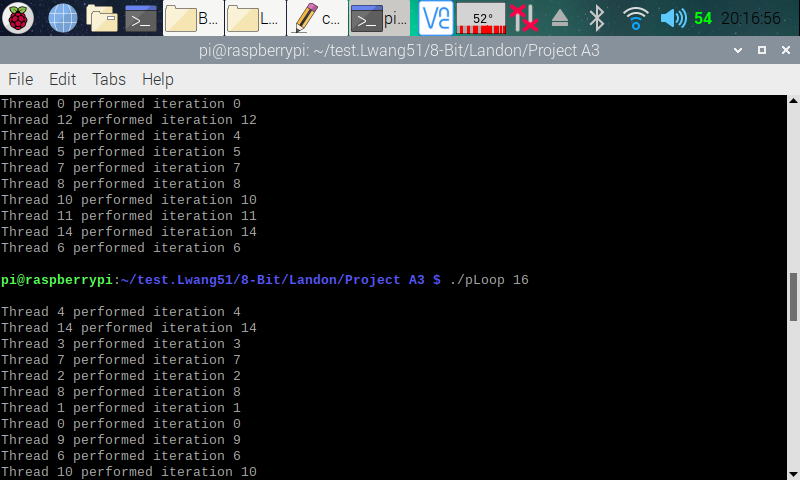
 Here (in the screenshot above), I exited from nano and made an executable program by using **gcc parallelLoopEqualChunks.c -o pLoop -fopenmp** instruction (this created the pLoop executable program). I then ran pLoop with a command line argument indicating how many threads to fork (**./pLoop 4**). We can see from the output that the program has forked four thread to complete 16 iterations.

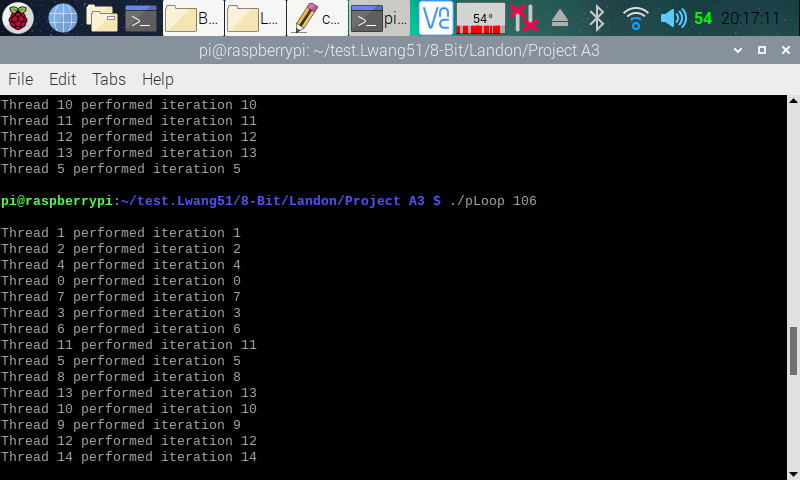


Here (in the screenshot above), I went back into the code and changed the number of reps to 15 instead of 16. I then ran the program indicating 4 threads to fork. We can see that threads 1 to 2 processes four iterations while thread 3 only processed three iterations. This is what we get when we have iterations that’s not evenly divisible by the number of threads.

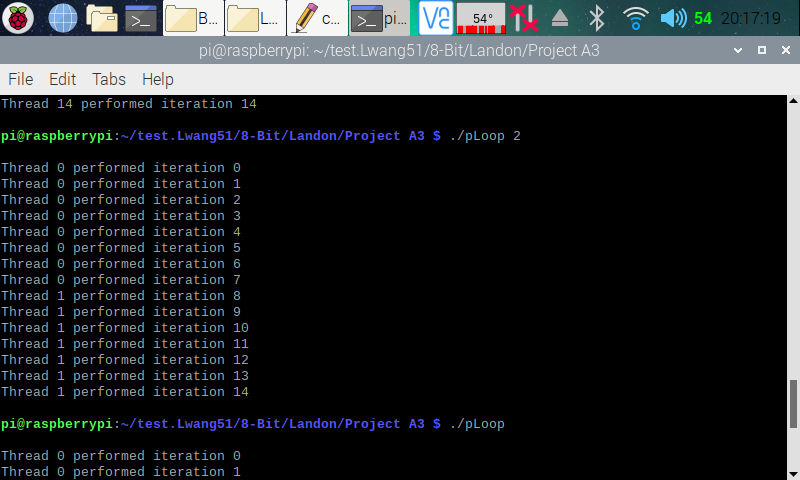


Here (in the screenshot above), I ran the program indicating 3 threads to fork. We can see that threads 1 to 2 each processed five iterations. Each thread processed an even number of iterations, because the number of iterations (15) is divisible by 3.

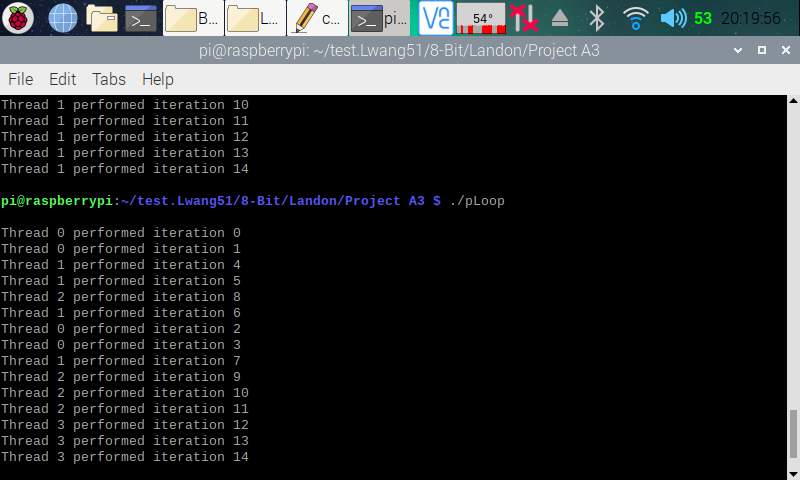




Here (in the two screenshots above), I ran the program indicating 16 and 106 threads to fork. We can see that although 16 and 106 threads were forked, only 15 of them were used for the 15 iterations.

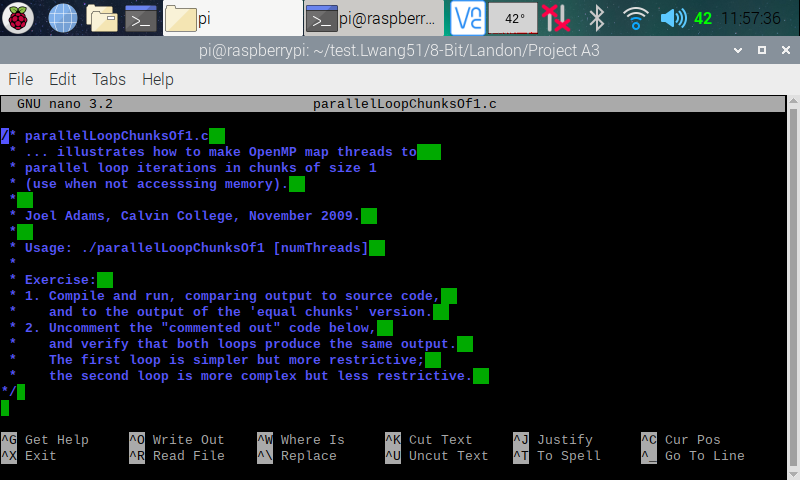


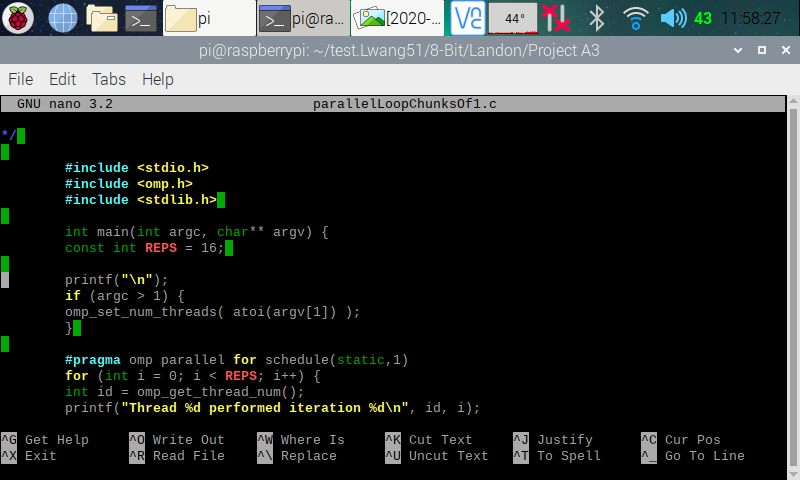
Here (in the screenshot above), I ran the program indicating 2 threads to fork. We can see that thread 0 processed eight iterations, and thread 1 processed seven iterations. The reason behind this is the same as when we ran the program with four threads to fork.

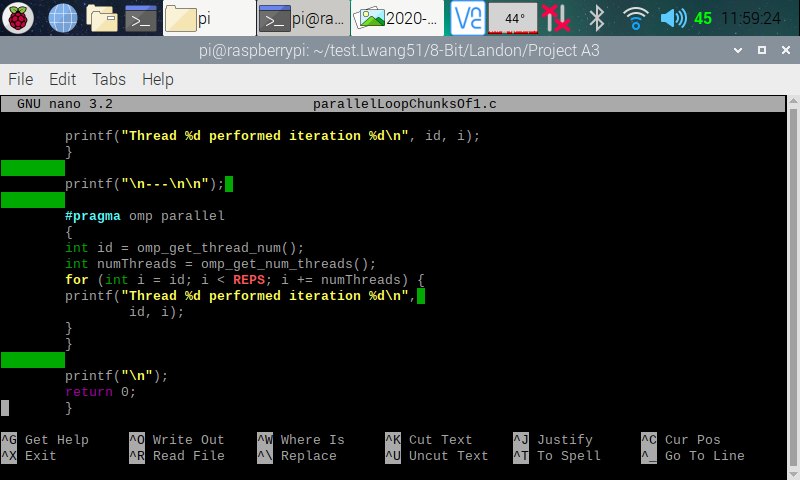


Here (in the screenshot above), I ran the program without indicating how many thread(s) to fork. The processor automatically goes with fours threads to fork since we have four processors available.

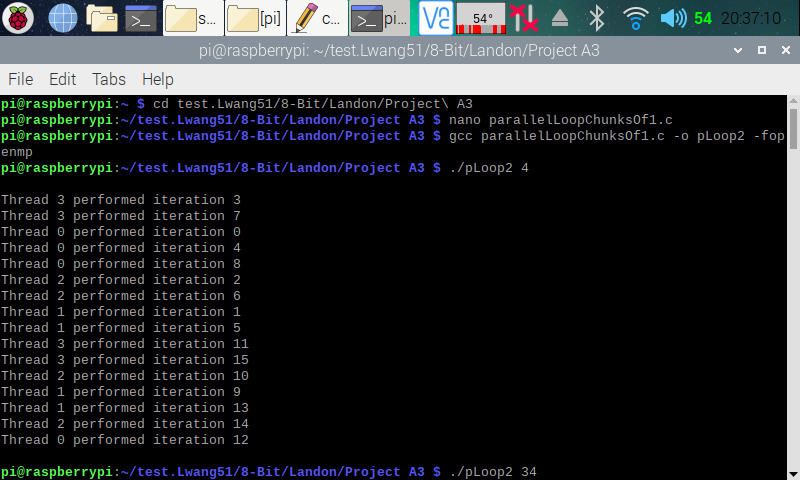
**Parallel Loop Chunks of 1 Program**



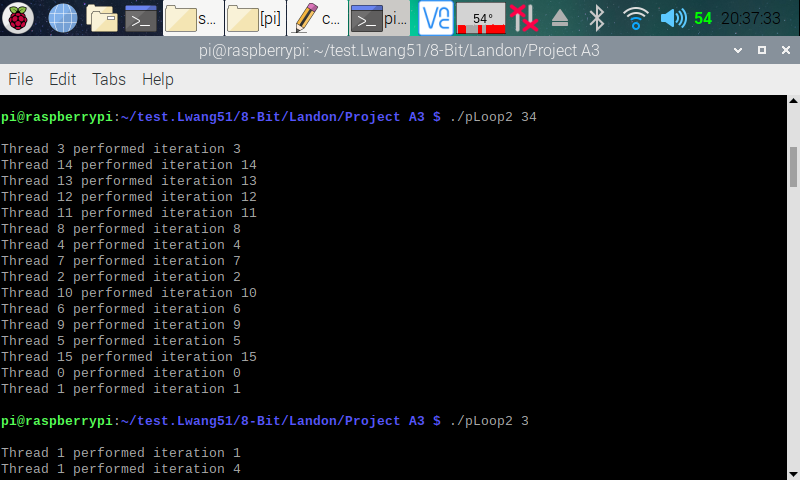


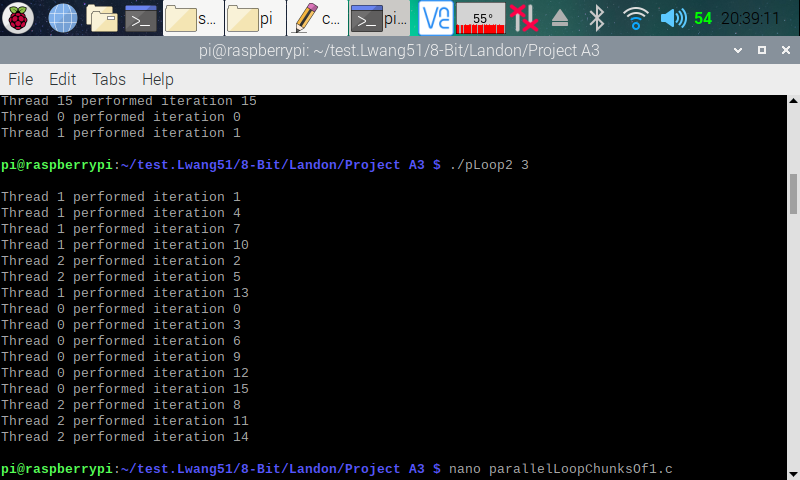


Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.

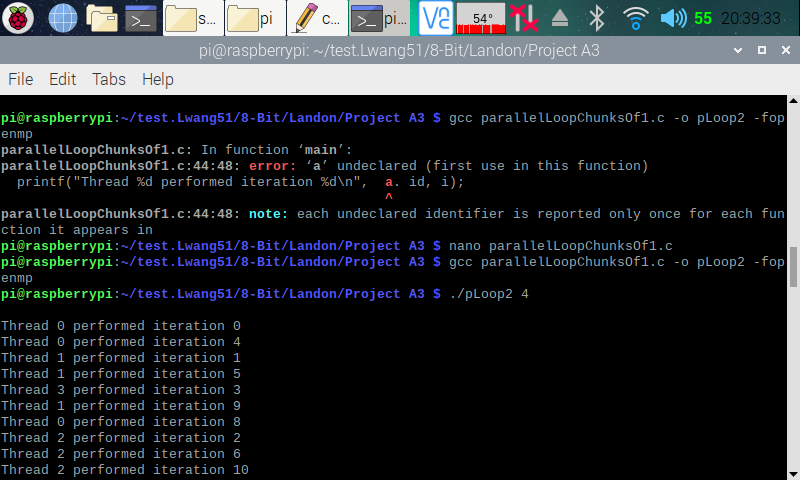


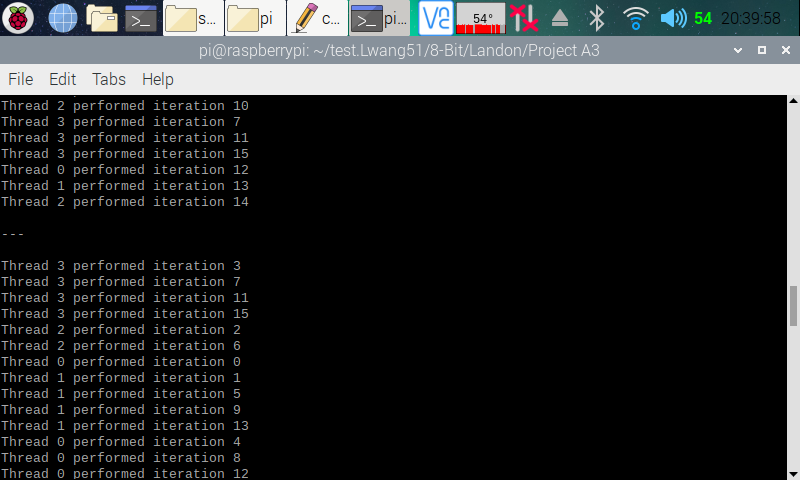
Here (in the screenshot above), I exited from nano and made an executable program by using **gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp** instruction (this created the pLoop2 executable program). I then ran pLoop2 with a command line argument indicating how many threads to fork (**./pLoop2 4**). We can see from the output that the program has forked four thread to complete 16 iterations. This type of threading is a little different from the previous program. It doles out one loop iteration to one thread, then the next iteration to the next thread and so on. As a thread completes its iteration, it goes to the next one. The threads still preform the same amount of work, but just not in a consecutive iteration like how it did in the last program.



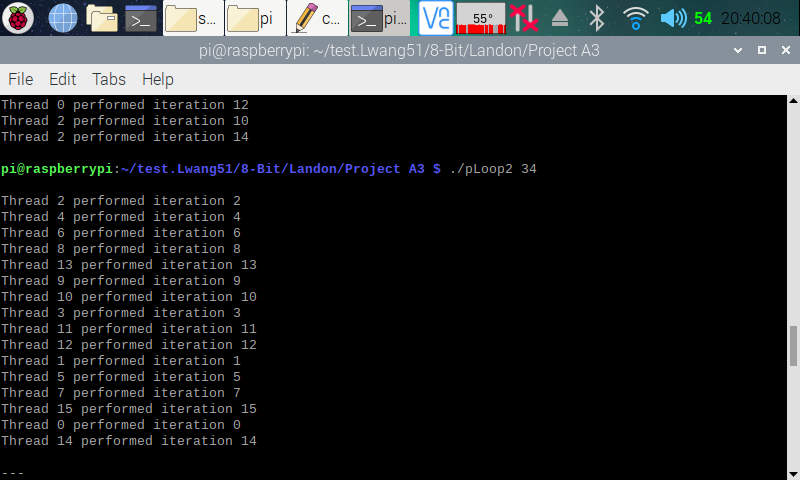


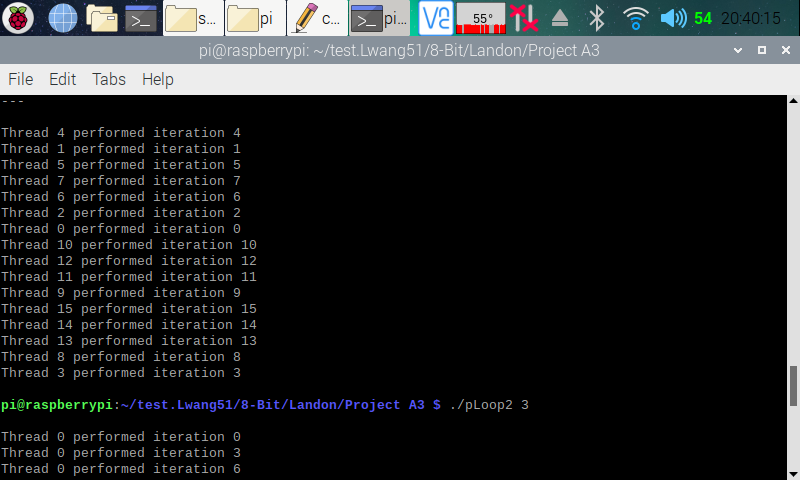
Here (in the screenshot above), I ran the program with 34 and 3 threads to fork. The concept of the amount of work each thread preformed is the same as the one explained in the previous program. How the threads preform is the same as if we ran it with 4 threads to fork.

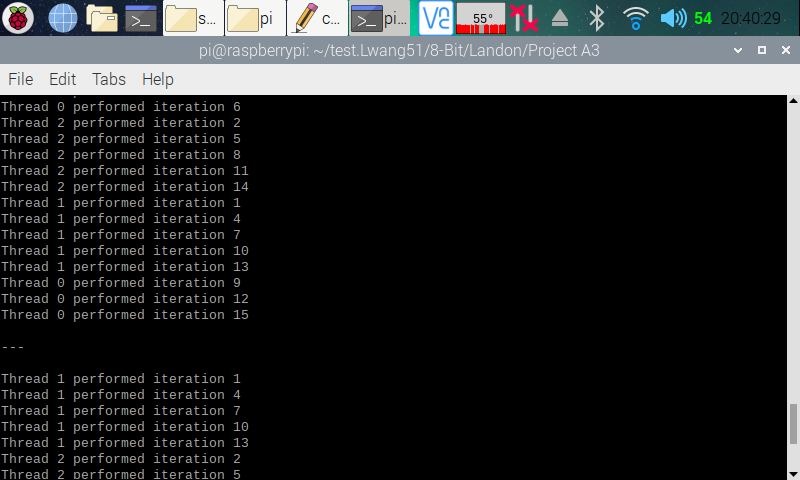


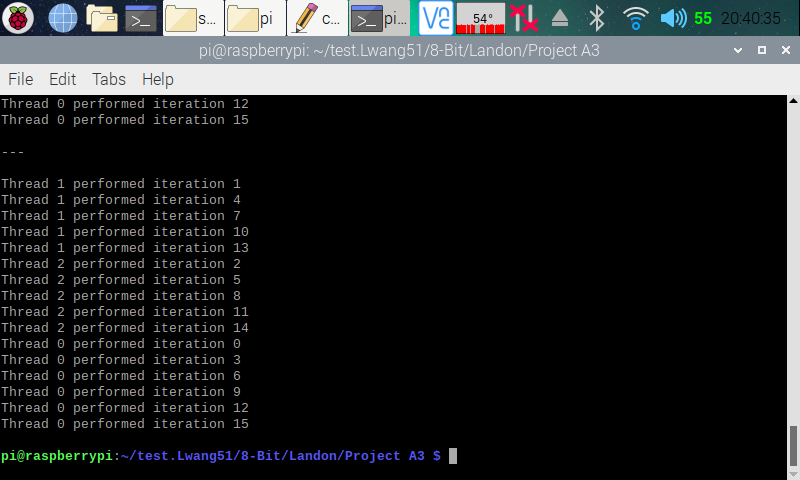


Here (in the screenshot above), I went back into the code and removed the commented section of the code as stated in the instructions. I then fixed up an error and updated the executable file. Then, I ran the program with 4 threads to fork. Now, both for loops produces the same type of output, but like stated in the instruction, the second for loop is more complex but less restrictive. Part of this program output is on the next screenshot.



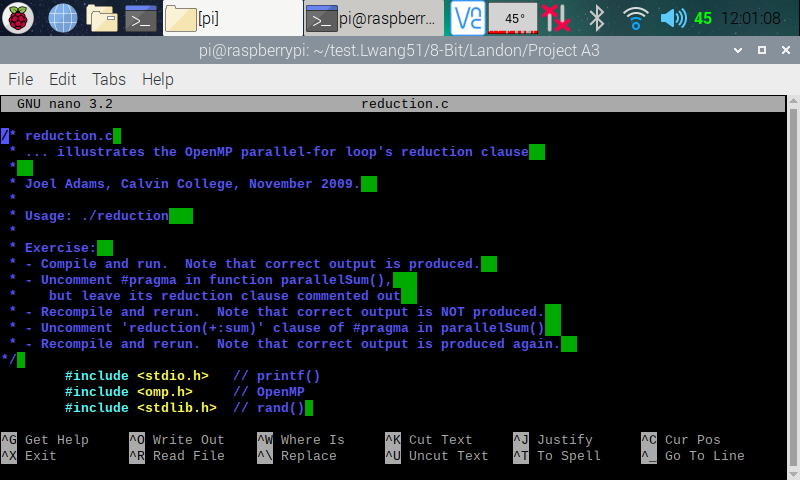


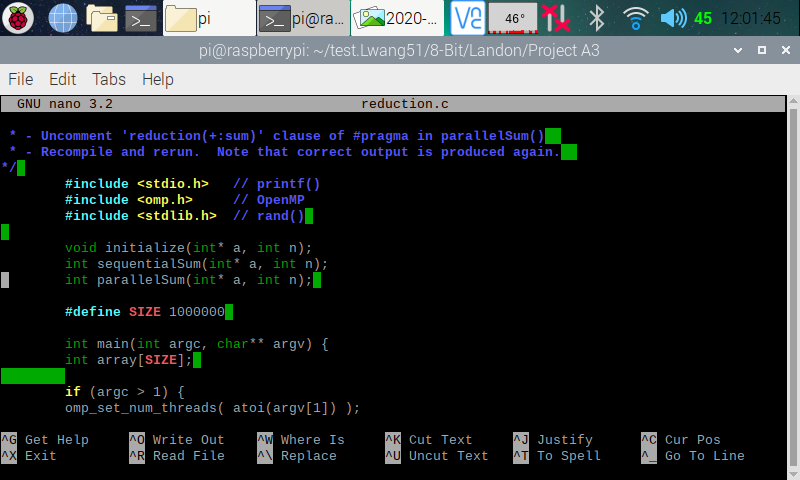


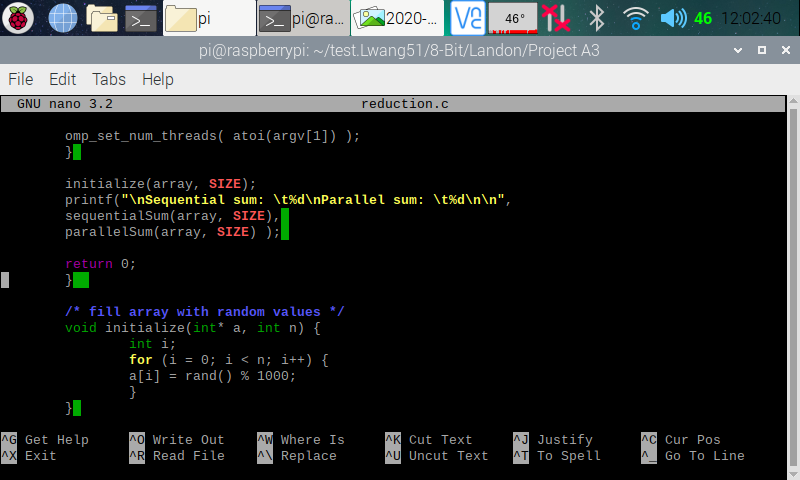


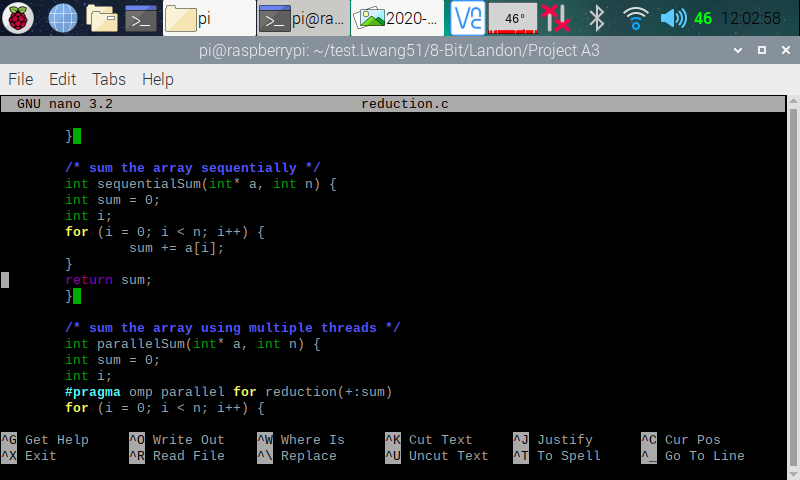
Here (in the four screenshots above), I ran the program with 34 and 3 threads to fork, the output and the reason why each program threaded in a way is the same as in the previous examples.

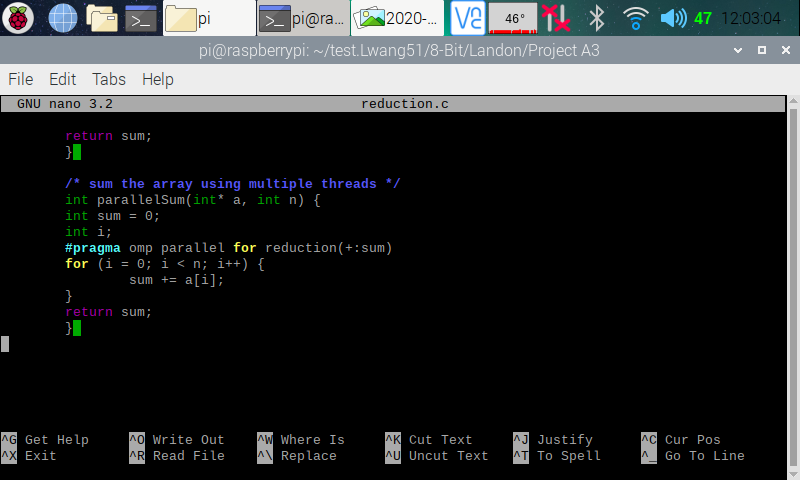
**Reduction Program**



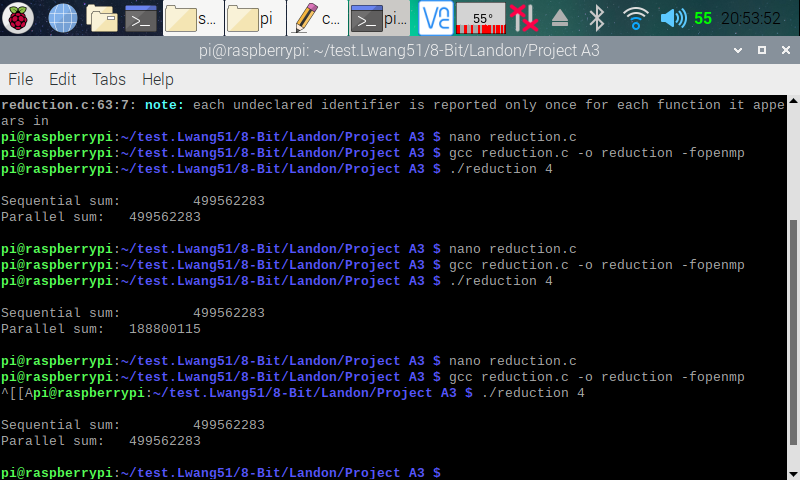


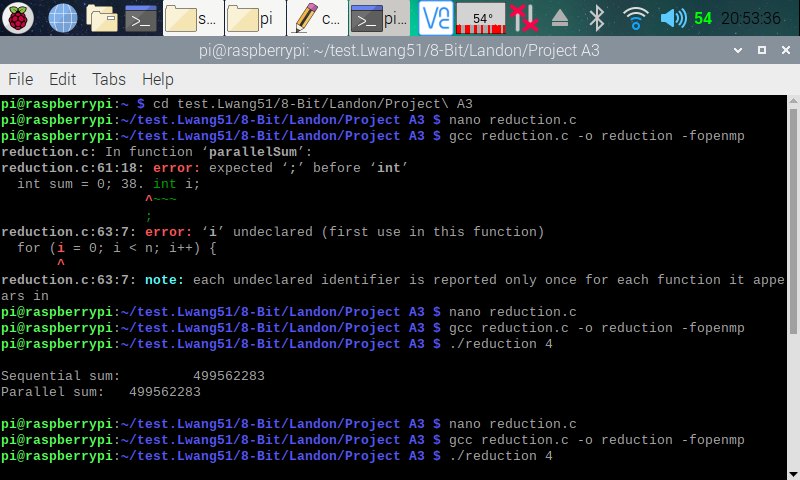






Here (in the five screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.





Here (in the two screenshots above), I exited out of the nano, and made the executable of the program. After compiling and running the program for the first time, we can see that both of the output is the same. However, from looking at the code and reading the parallel programming task document, I concluded that the parallel sum from the first run is not actually calculated using parallel computing. I say this because the “// #pragma omp parallel for //reduction(+:sum)” line of code is commented out, and we need that to do parallel programming.

After going back and removing the first comment of the “// #pragma omp parallel for //reduction(+:sum)” code, I ran the program for the second time. After the program executed, we can see that the results from the sequential sum and the parallel sum does not match. I then went back and removed the second comment from the “#pragma omp parallel for //reduction(+:sum)” code. I then ran the program for the third time, and now, we can see that the sequential and a parallel sum both matched. Furthermore, the parallel sum is now calculated using parallel computing.

The reason why parallel for pragma without the reduction clause did not produce the correct result was because the variables used were not private, and that each thread did not have its own copy of the variable.