Developing Soft and Parallel Programming Skills using Project-Based Learning

Spring 2020

The Commuters
Alaya Shack, Miguel Romo, Arteen Ghafourikia, Andre Nguyenphuc, Joan Galicia

Planning and Scheduling:

Assignee Name	Email	Task	Duration (hours)	Dependency	Due Date	Note
Alaya Shack (Team Coordinator)	ashack1@student.gsu.edu	Edit the video and include the link in the report. Complete individual parallel programming skills task and ARM assembly programming task.	5 hours	(none)	03/04	Review report for grammatical/ spelling errors.
Miguel Romo	mromo1@student.gsu.edu	Complete columns in Github. Complete individual parallel programming skills task and ARM assembly programming task.	5 hours	Github, each member's report, and the video	03/04	Review report for grammatical/ spelling errors.
Joan Galicia	jgalicia2@student.gsu.edu	Get the report formatted correctly (fonts, page numbers, and sections). Complete individual parallel programming skills task and	6 hours	(none)	03/05	24 hours before the due date, please have the report ready for all team members to review.

		ARM assembly programming task.				
Arteen Ghafourikia	aghafourikia1@student.gsu.edu	Serve as the facilitator. Complete individual parallel programming skills task and ARM assembly programming task.	5 hours	(none)	03/04	Review report for grammatical/ spelling errors
Andre Nguyenphuc	anguyenphuc1@student.gsu.edu	Create the planning and scheduling table. Complete individual parallel programming skills task and ARM assembly programming task.	5 hours	(none)	03/04	Review report for grammatical/ spelling errors.

Parallel Programming Skills Foundation: Alaya Shack

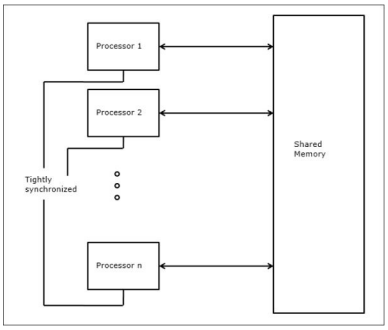
- **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
  - A task is a program or a series of instructions that are executed by a processor.
  - Pipelining is a process in which a task is divided into different steps, and the different steps are worked on by different processor units. This is a type of parallel computing that resembles an assembly line.
  - The way shared memory can be described depends on the perspective it is viewed from. From a hardware point of view, shared memory is a computer architecture in which all processors have access to a common physical memory. From a programming perspective, shared memory is when parallel tasks have the same knowledge of a memory, regardless of knowing the actual physical memory address.
  - Communications is the action of exchanging data through a shared memory bus or over a network.
  - Synchronization is the process of coordinating parallel tasks in real time.This usually involves one task waiting on another task to reach the same point so that the tasks can proceed further.
- **Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.**
  - Single Instruction stream, Single Data stream is a type of serial computer, and it is the oldest type of computer. This classification involves single instruction and single data stream, which means only one instruction stream is being acted on by the CPU during any one clock cycle and that only one data stream is being used for input during any one clock cycle. The execution is deterministic, meaning that if given some input, the output would be the same no matter how many times the operation is executed.
  - Single Instruction stream, Multiple Data Stream (SIMD) is a type of parallel computer, in which all processing units execute the same instruction and each processing unit can work on different data at any given clock cycle.
  - Multiple Instruction stream, Single Data stream(MISD) is a type of parallel computer, in which each processing unit operates on data through separate instruction streams and a single data stream is given to multiple processing units.
  - Multiple Instruction Stream,Multiple Data Stream (MIMD) is a type of parallel computer, in which every processing unit can be executing a different instruction stream and every processor could be working with different data streams.
- **What are the Parallel Programming Models?**
  - The Parallel Programming Models that are commonly used are the shared memory(without threads), threads, distributed memory/message passing, data parallel, hybrid,single program multiple data, and the multiple program multiple data.
- **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
  - Uniform Memory Access (UMA) is a shared memory machine that has identical processors and has equal access and access times to the memory. In this

architecture, if one processor makes an update, the other processors have knowledge of this update.

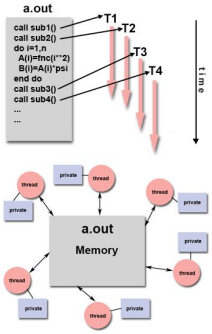
- Non-Uniform Memory Access (NUMA) is a shared memory machine that is made physically by linking two or more symmetric multiprocessor machines (SMP). One SMP can access the memory of another SMP. The processors for NUMA do not always have equal access times to all memories.
- OpenMP uses UMA because UMA allows for if one processor makes an update, then the other processors will be able to see that update.

➢ **Compare Shared Memory Model with Threads Model?**

- In the Shared Memory Model(without threads), the processes and tasks shared a common address space, and they do not read and write to it at the same time. All processes see and have equal access to shared memory.



- Threads Model involves a single heavyweight process that can have multiple light weight processes executing at the same time. Threads communicate with each through global memory, and this requires synchronization.



➢ **What is Parallel Programming?**

- Parallel programming is the process of using multiple processors to solve a problem that can be broken down into smaller steps, and those smaller steps can be executed simultaneously.

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

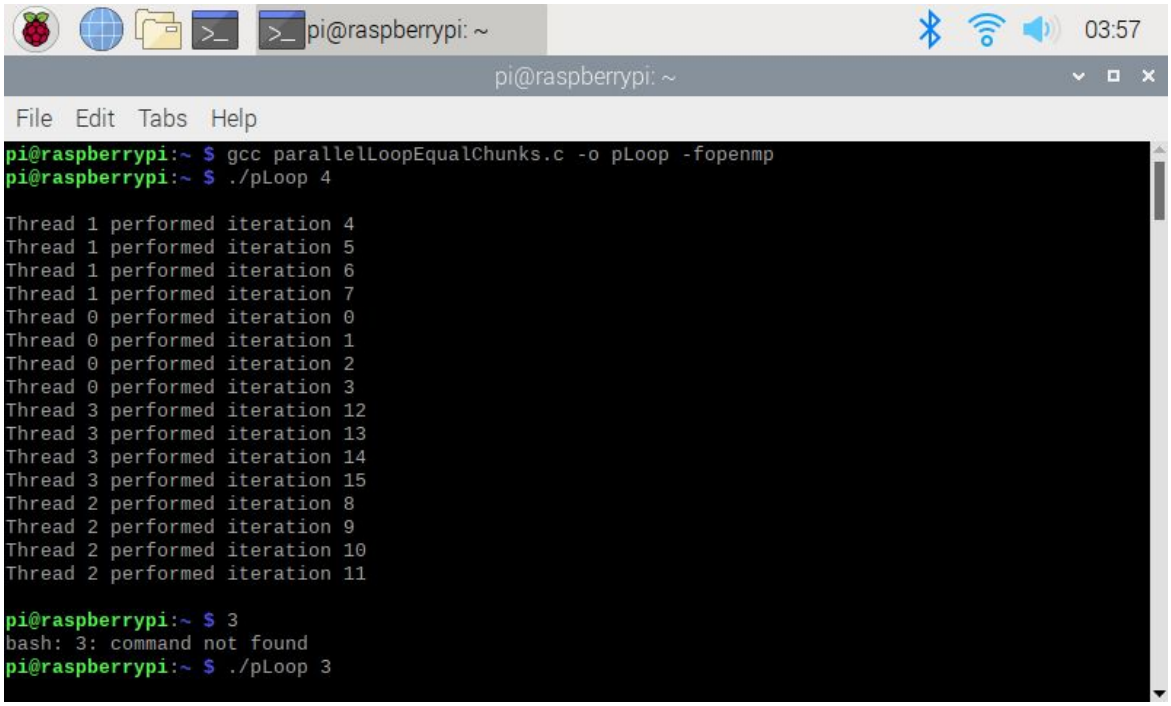
- System-on-a-chip is a single silicon chip that usually contains the following components: a central processing unit(CPU), a graphics processor(GPU), memory, USB controller, power management circuits, and wireless radios. Raspberry Pi does use a SoC unit that has the CPU, RAM, GPU, and other components on it.

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

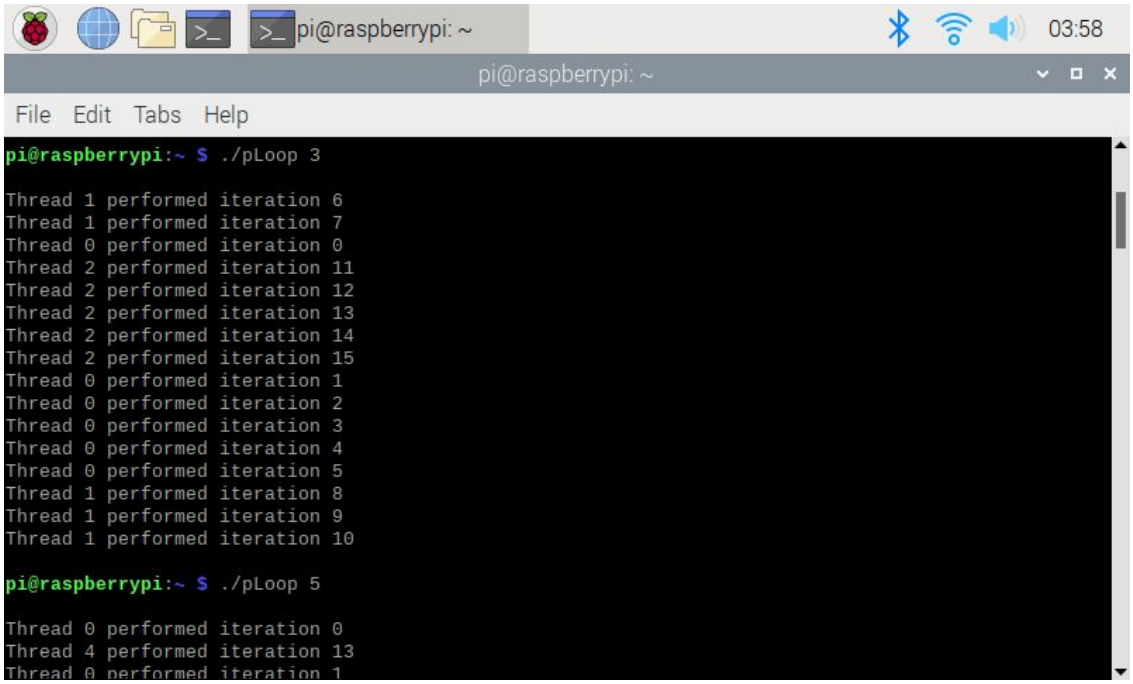
- One advantage of having a SoC rather than separate CPU, GPU, and RAM components is its size. The SoC is only a little bigger than a CPU, but it has a lot more functionality. Also, having a SoC allows for more room for batteries.
- Another advantage of a SoC is that it uses less power than having separate CPU,RAM, and GPU components. Since the number of chips are less with a SoC, this makes building a computer with a SoC less expensive.

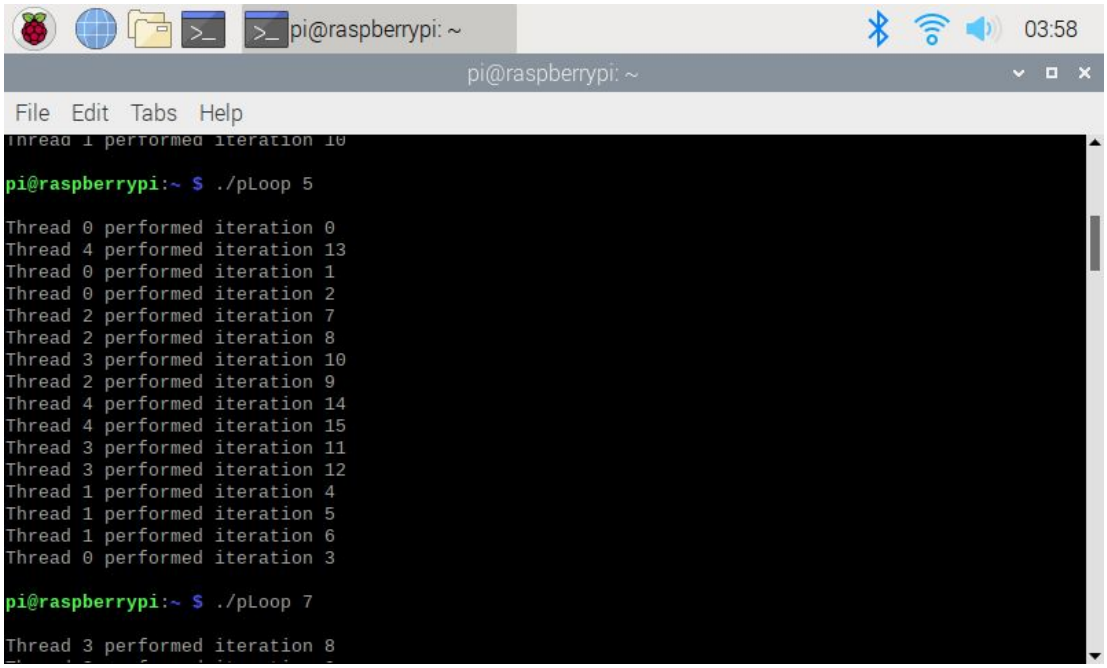
Parallel Programming Basics: Alaya Shack

Part1



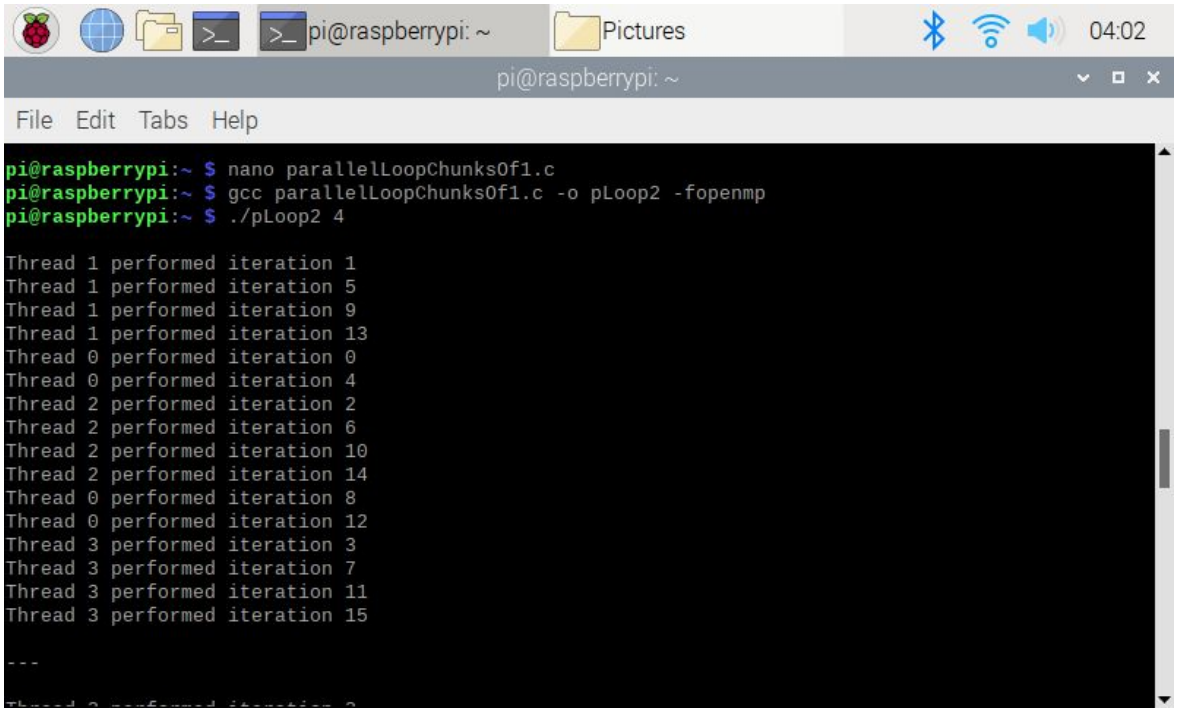
This screenshot shows how I made the parallelLoopEqualChunks program executable with the first line. Then, I ran the program with "./pLoop 4", and 4 signifies the number of threads. I noticed that the four threads all performed 4 iterations. Also, the threads executed the iterations in order, so for example 0 performed 4 iterations, which were iteration 0,1,2,and 3, and 1 performed 4 iterations, which were 4,5,6, and 7.



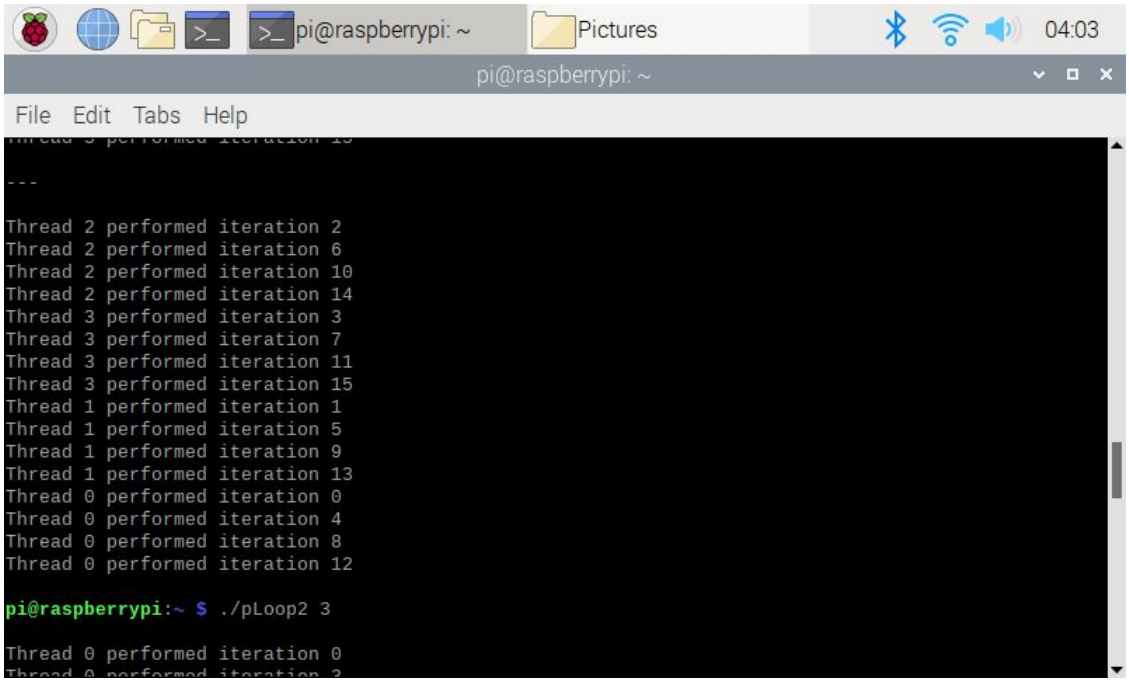


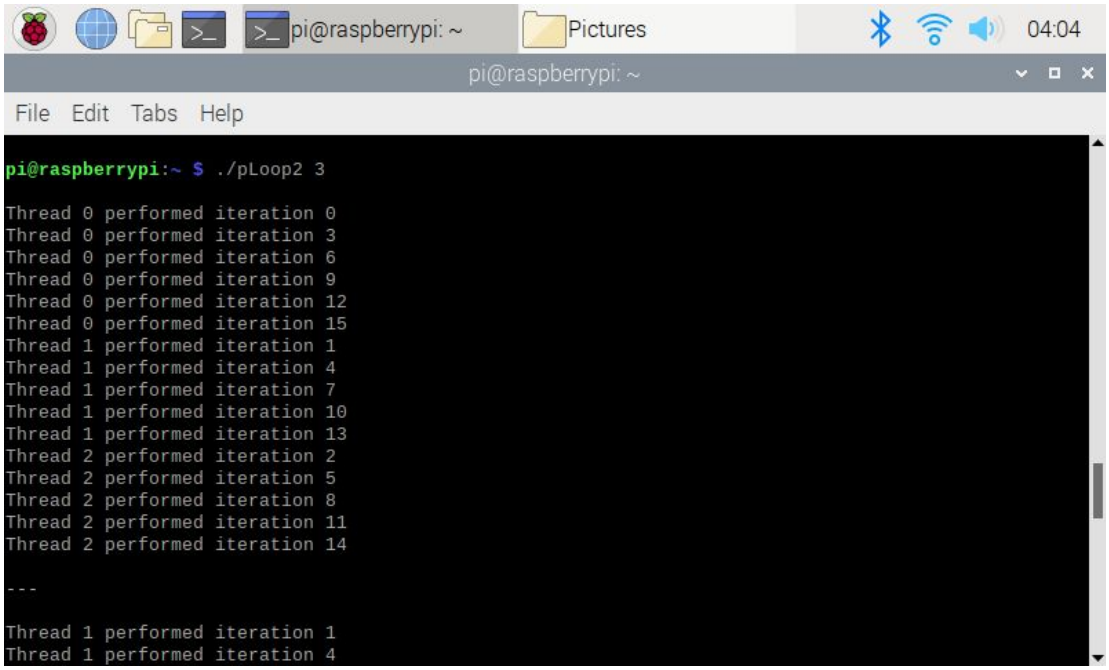
In these two screenshots, I used 3,5, and 7 to see how the compiler splits up the work, when the number of iterations is not evenly divisible by the number of threads. The compiler still has the threads execute iterations in order, but every thread does not perform an iteration the same number of times.

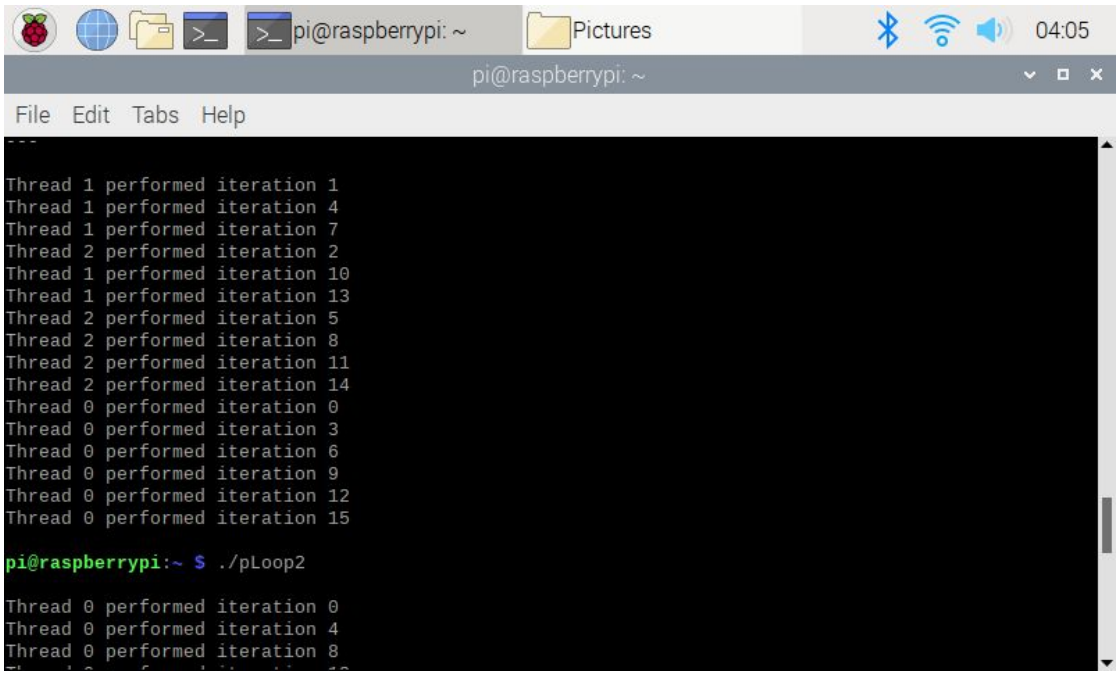
Part2



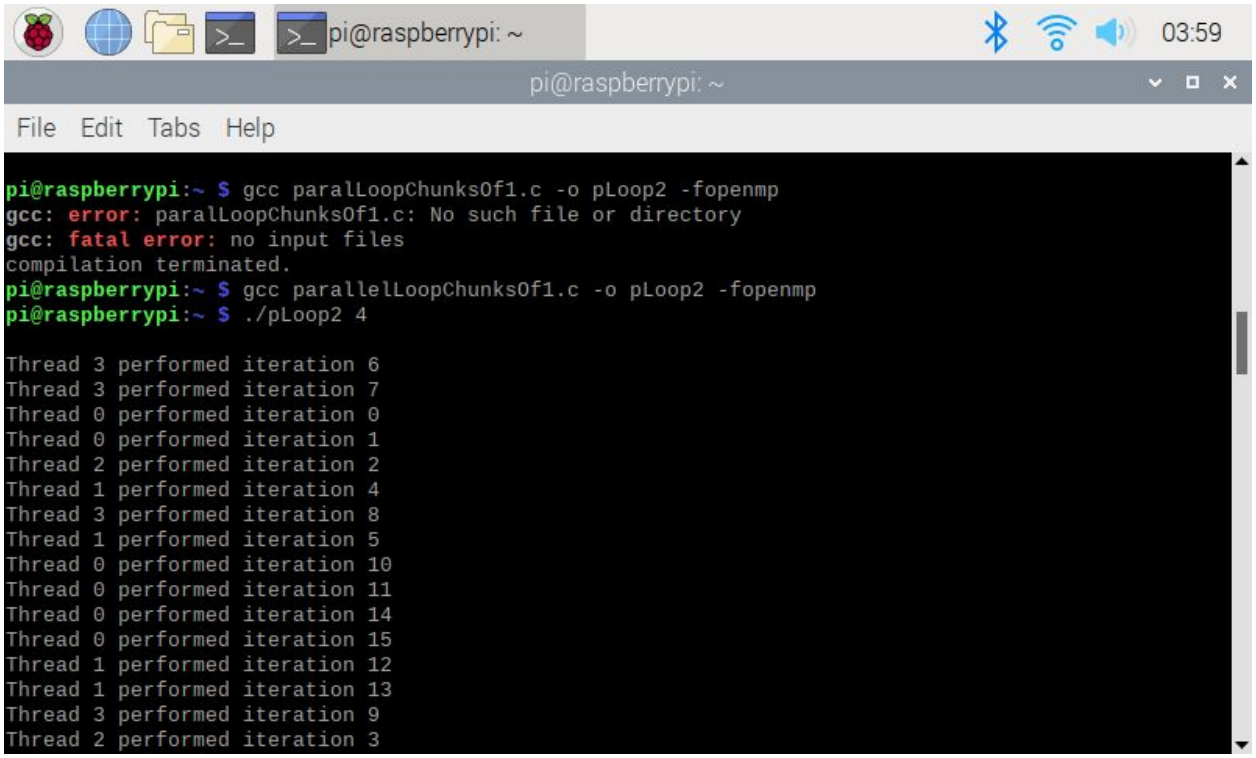
In this screenshot, I edited the parallelLoopChunksOf1 program. Then, I made it an executable file by typing the second line. Then, I ran the program with "./pLoop2 4".

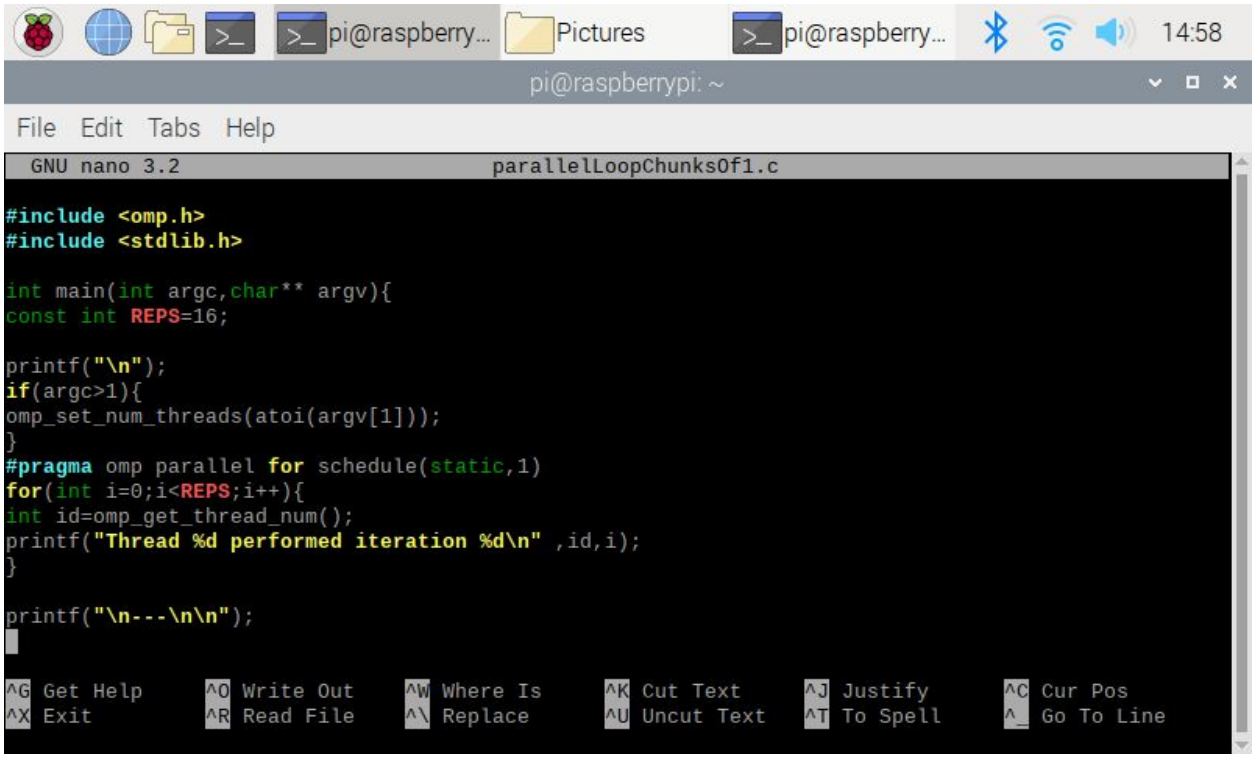


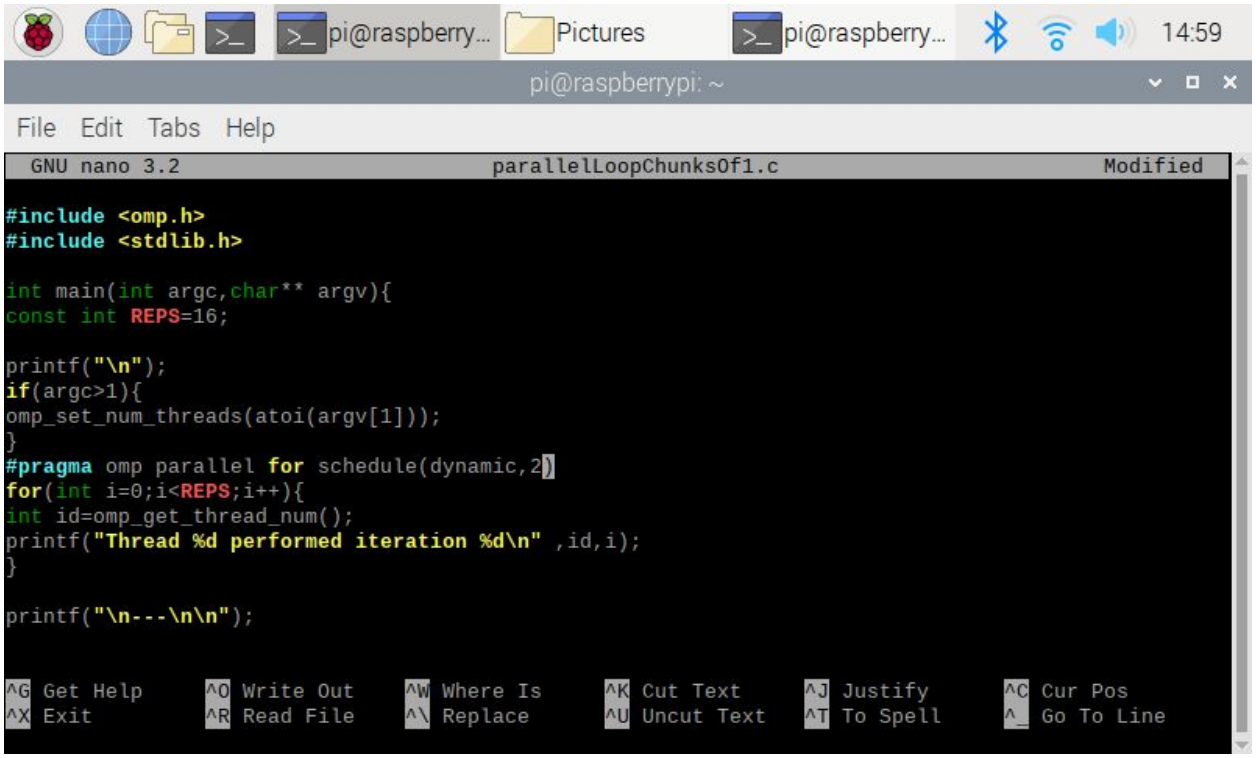




In these three screenshots, I ran the parallelLoopChunksOf1 program, and I used 4,3, and 0 for the number of threads. For each time when I ran the program with a certain number of threads, both of the outputs were the same. I noticed that with this execution of the loops, a thread would perform an iteration, and then the next thread would perform another iteration, and this would repeat until all 16 iterations were completed. For example, 0 would perform an iteration, and then 1 would perform an iteration.

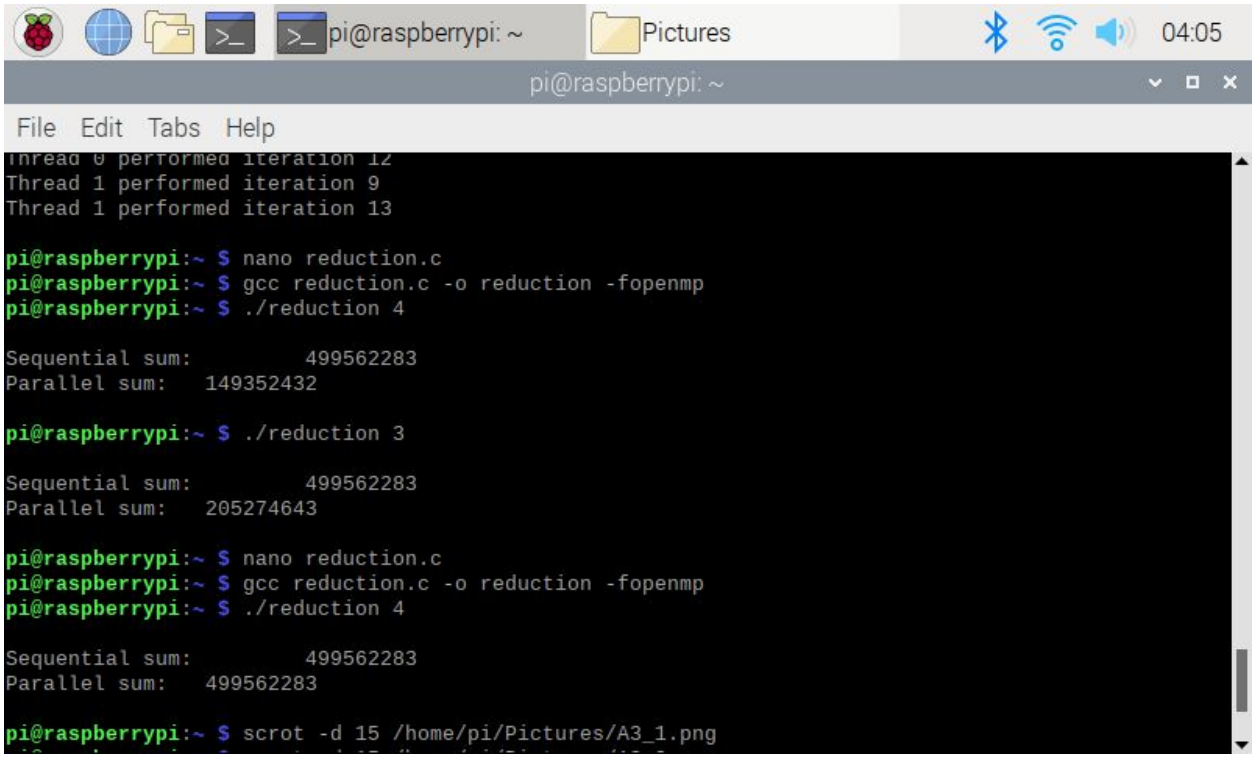

pi@raspberrypi: ~
03:59
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ gcc paralLoopChunksOf1.c -o pLoop2 -fopenmp
gcc: error: paralLoopChunksOf1.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4
Thread 3 performed iteration 6
Thread 3 performed iteration 7
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 2 performed iteration 2
Thread 1 performed iteration 4
Thread 3 performed iteration 8
Thread 1 performed iteration 5
Thread 0 performed iteration 10
Thread 0 performed iteration 11
Thread 0 performed iteration 14
Thread 0 performed iteration 15
Thread 1 performed iteration 12
Thread 1 performed iteration 13
Thread 3 performed iteration 9
Thread 2 performed iteration 3

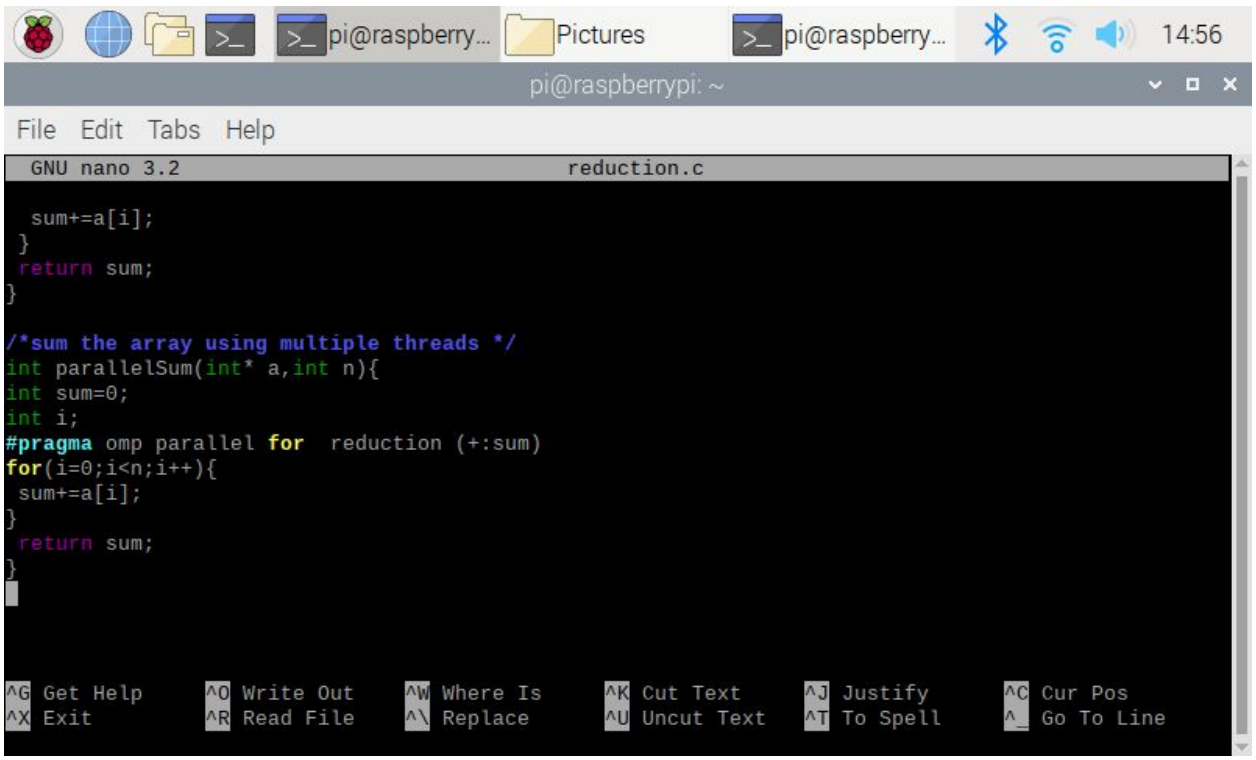

pi@raspberry...
Pictures
pi@raspberry...
14:58
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopChunksOf1.c
#include <omp.h>
#include <stdlib.h>
int main(int argc,char** argv){
const int REPS=16;
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel for schedule(static,1)
for(int i=0;i<REPS;i++){
int id=omp_get_thread_num();
printf("Thread %d performed iteration %d\n" ,id,i);
}
printf("\n---\n\n");
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



In these three screenshots, I changed the parallelLoopChunksOf1.c program. I changed the "#pragma omp parallel for schedule(static,1)" to "#pragma omp parallel for schedule (dynamic,2)". Then, I ran the program with "./pLoop2 4". I examined how changing the keyword to dynamic instead of static changed the output and how changing the different sized chunks to 2 changed the output. I noticed that the threads did not perform in any particular order. I learned that dynamic makes it so that threads that take a shorter time to complete their task can work on the next piece, while threads that take a longer time can still work on their piece.

Part3



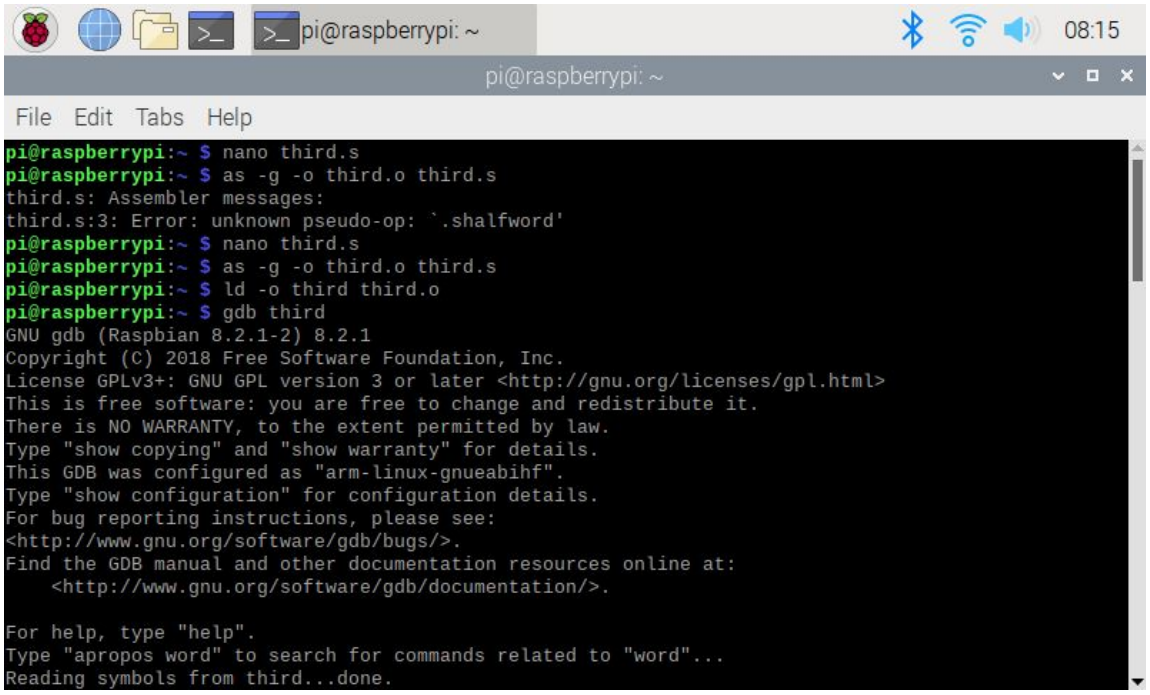


These two screenshots show how I edited the reduction program with nano. The first edit that I made was to uncomment "#pragma omp parallel for" so that the program would actually run in parallel. I made the program executable, and I ran it with "./reduction 4" and with "./reduction

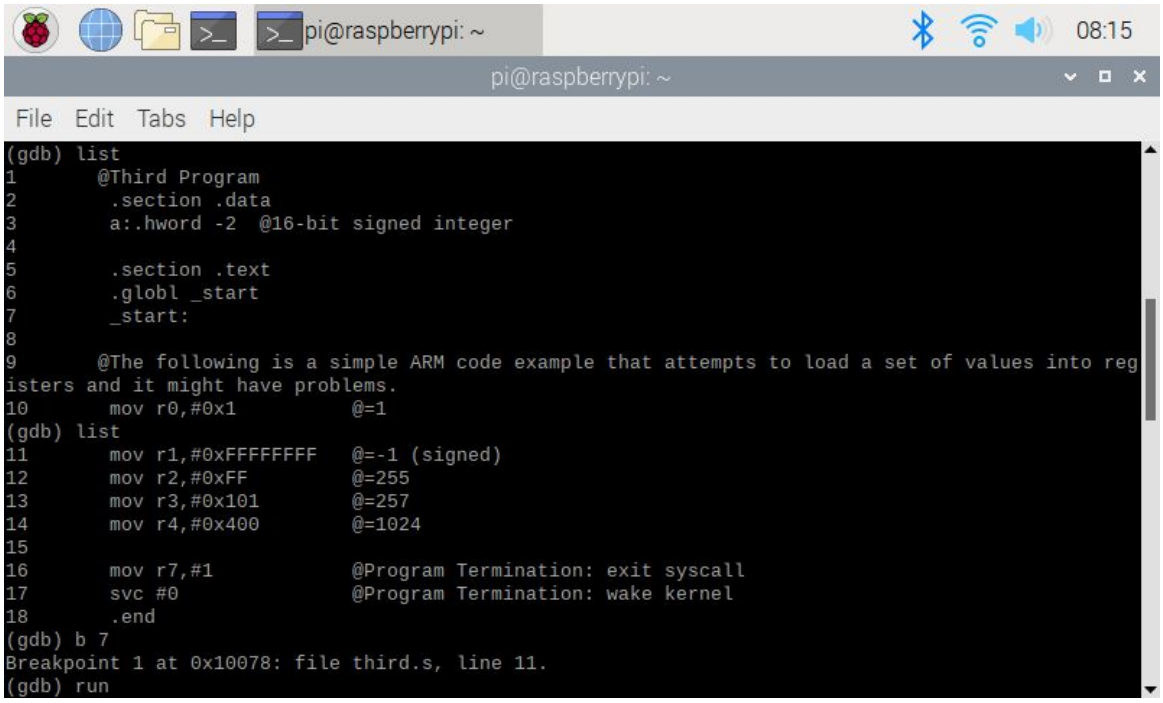
3", the 3 and 4 change the number of threads. With both cases, the sequential and parallel sum are not equivalent, but they should be equivalent. For the second time that I used nano, I uncommented "reduction (+:sum)". Then, I made the program executable, and I ran the program with "./reduction 4". With this case, I noticed that parallel and sequential sum were equivalent. This shows that we need the reduction clause if we want an accurate answer of the sum of the digits from 0 to 1000000(exclusive). We need the reduction clause because we want sum to be private to each thread, and then when each thread is finished, the sums of their individual sums are added.

ARM Assembly Programming: Alaya Shack

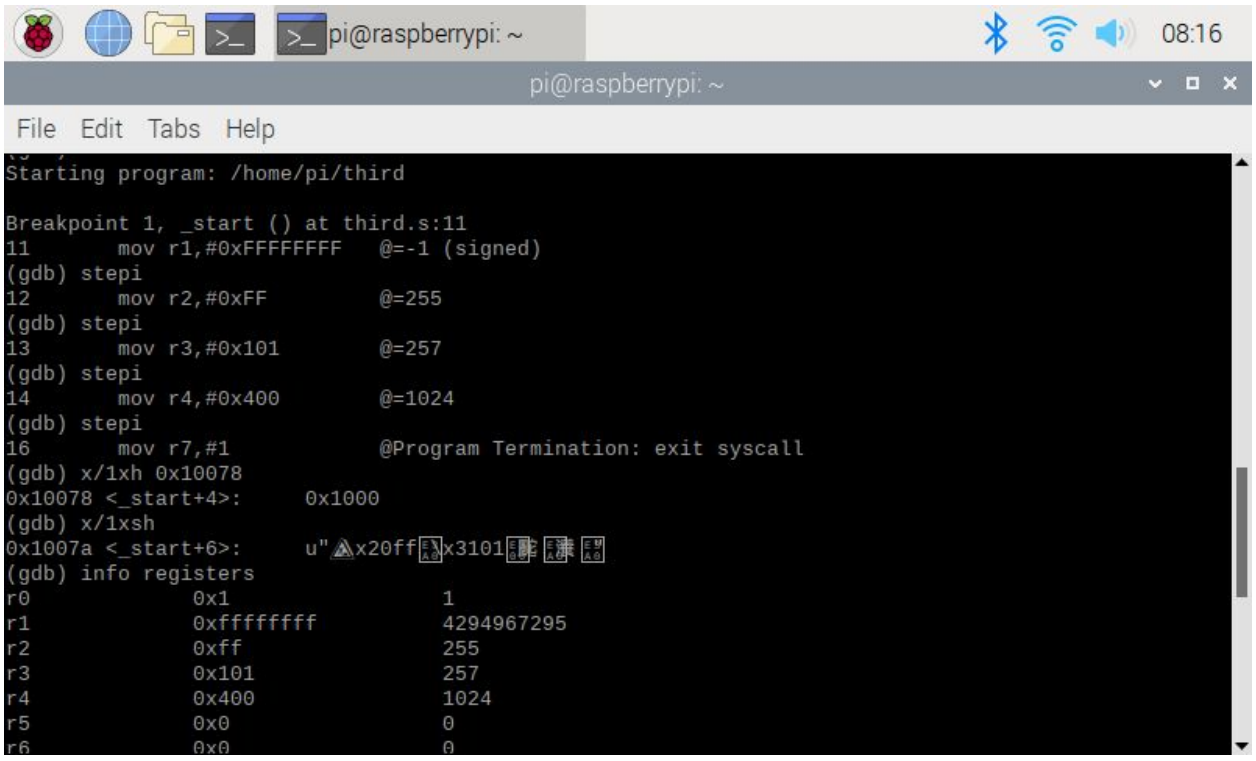
Part A



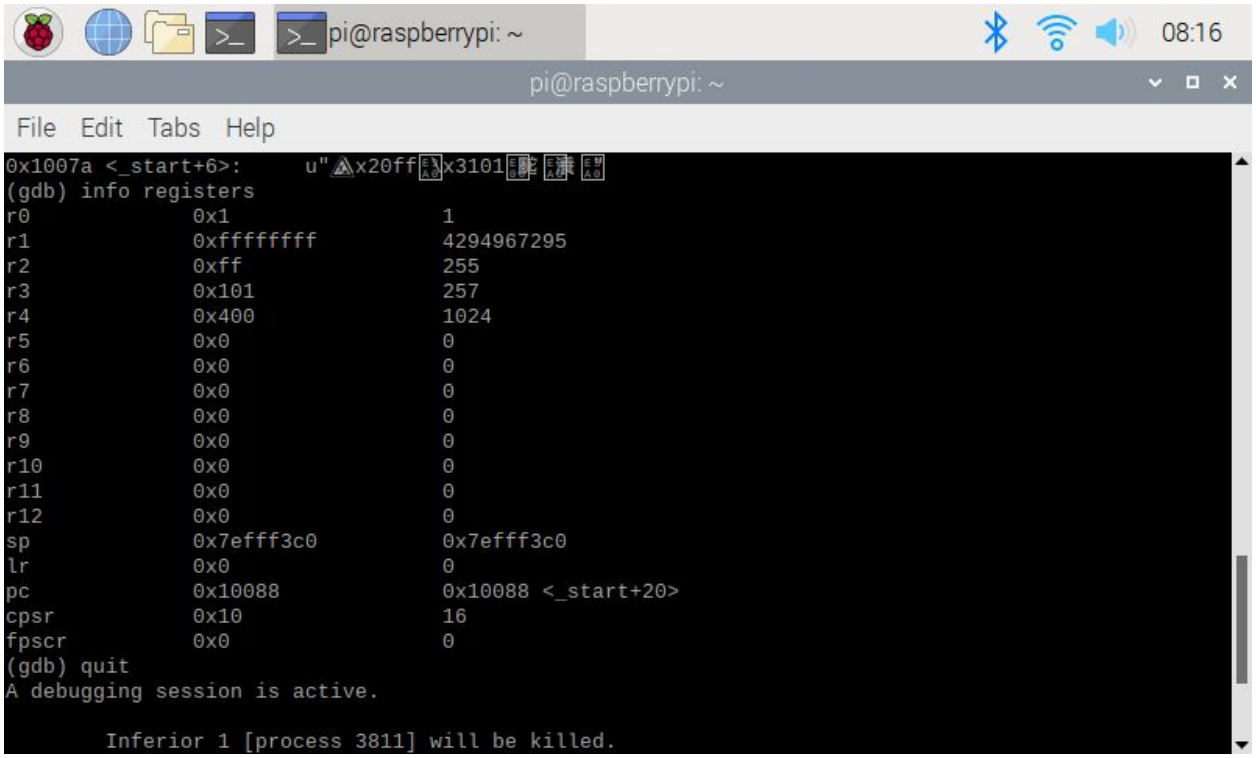
In this screenshot, I edited the third file, and then I assembled the file. When I assembled the file the first time, I got the error message "unknown pseudo op: .shalfword". I got this error message because this is not how you declare a signed 16-bit integer. Then, to correct this error, I changed the declaration of a to ".hword". Finally, I was able to assemble and link the file. Then, I launched the debugger with "gdb third".



In this screenshot, I used the list command to "list" the first ten lines of code, and then I used the same command to list the next ten lines of code. Then, I created a breakpoint at line 7, and I ran the program.

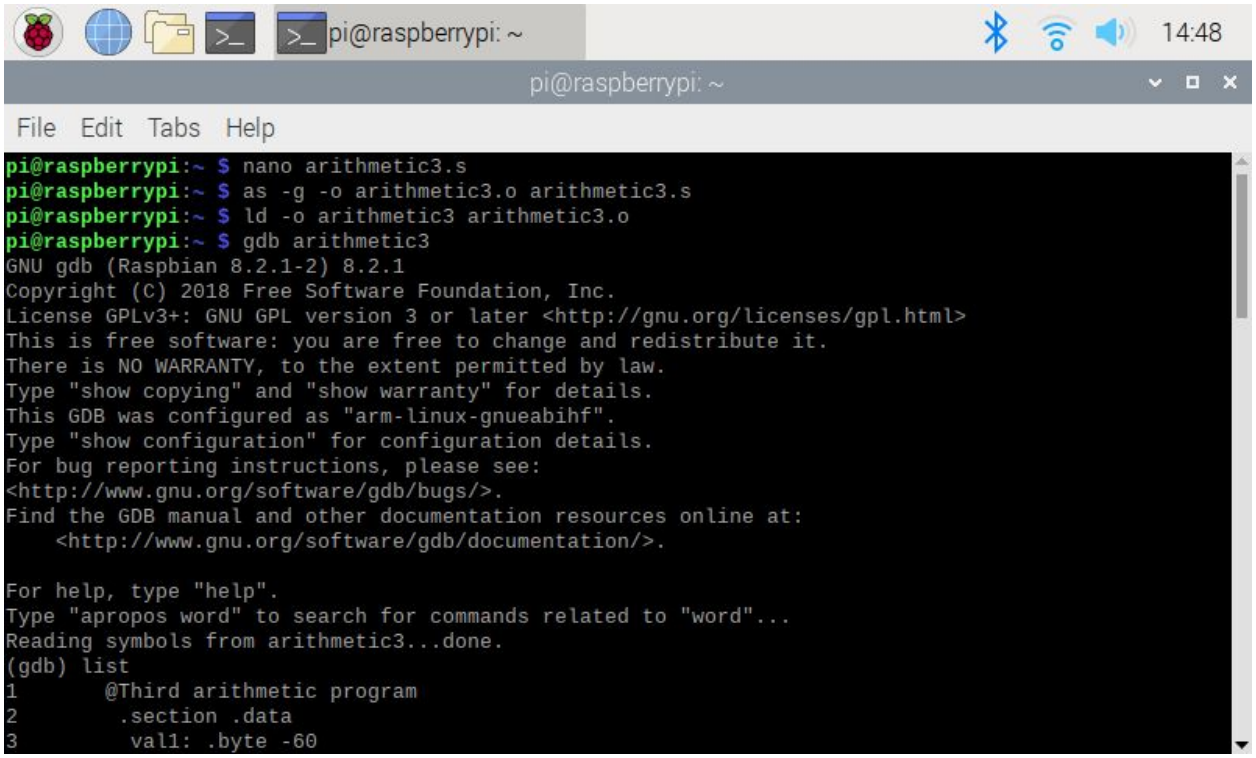


In this screenshot, I used the stepi command to step through the code one line at a time. Also, I used “gdb x/1xh 0x10078” to examine the memory. I tried to use “gdb x/1xsh 0x10078” but there was some encrypted result.

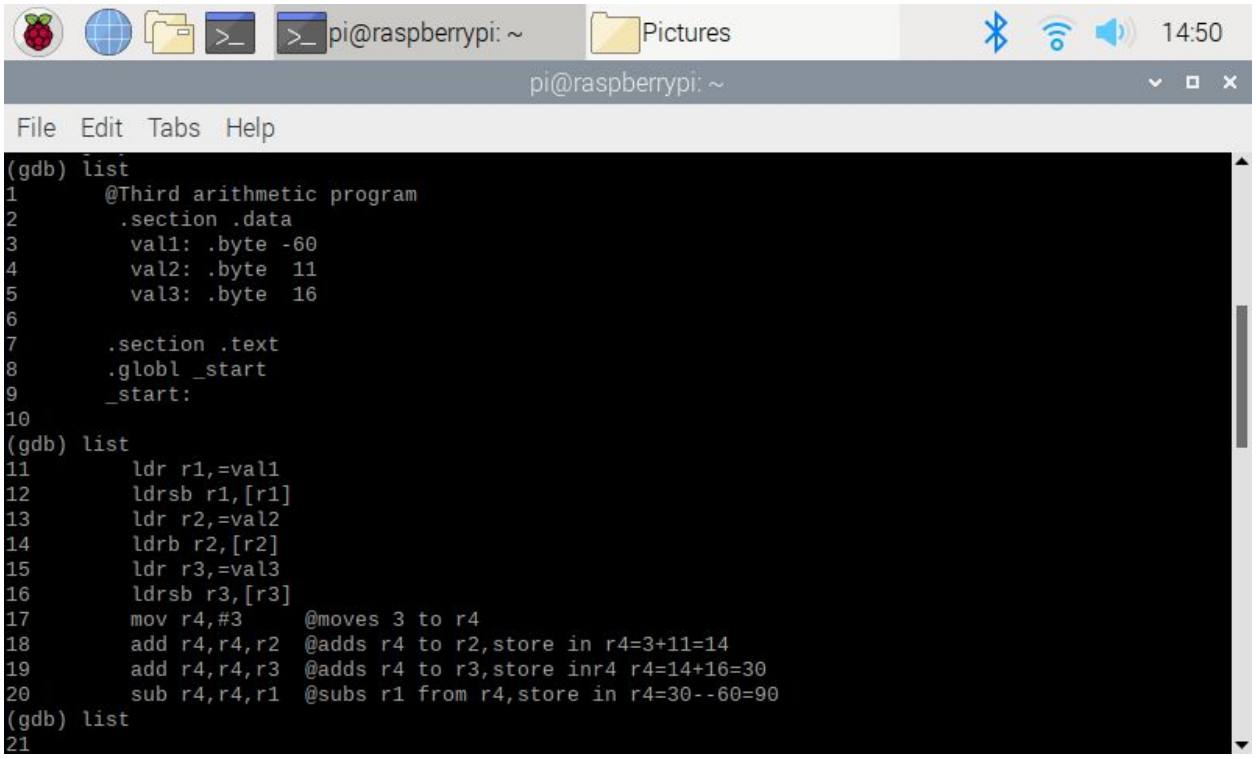


This screenshot shows the registers for the third program. The values in the registers r0,r2,r3,and r4 are what I expected because we moved those values to those specific registers. The only exception is for r1, the hexadecimal representation of -1 is correct, but the decimal value is not correct.

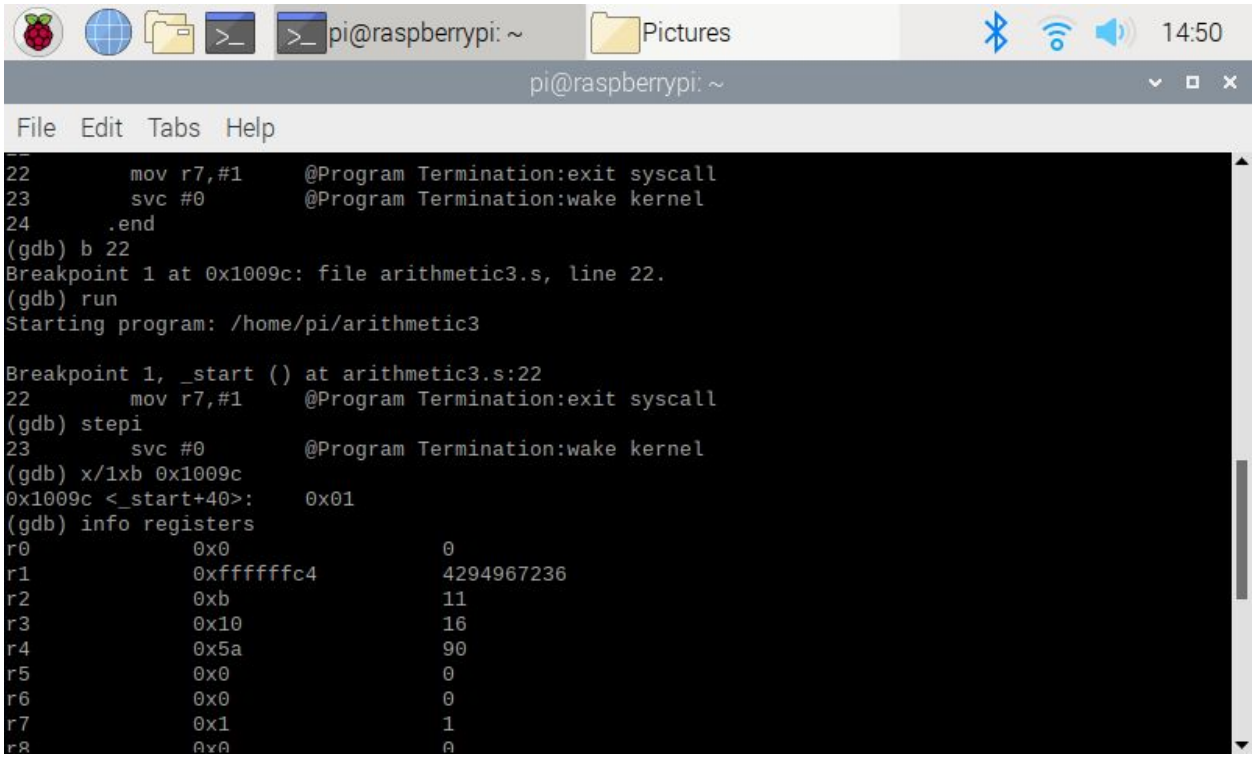
Part B



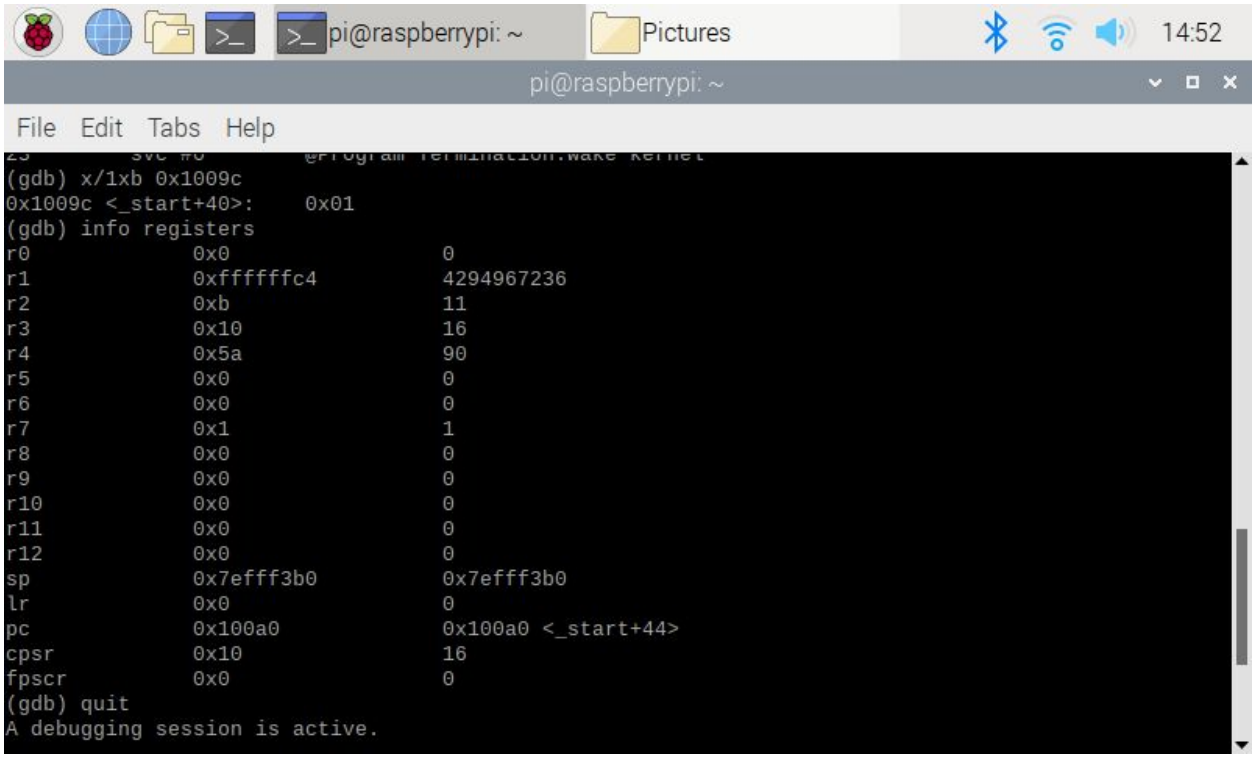
In this screenshot, I edited my arithmetic3 file, and I then assembled and linked the file. I launched the debugger with the “gdb arithmetic 3” command.



This is where I used the list command three times to list the first 30 lines of code. In my code, I declared val1, val2,val3 as bytes because they are 8-bit memory variables, and I then assigned their respective values. I used “ldr” to load memory addresses of the variables. However, for loading the actual value, I used “ldrb” for val2 because it is an unsigned byte. For val1 and val3, I used “ldrsb” because they are signed bytes. For the rest of the code, I used comments to explain what I was trying to accomplish.



In this screenshot, I inserted my breakpoint at line 22, and I ran the program. Then, I used the “stepi” command to step through one line of code. Then, I used “gdb x/1xb 0x1009c” to display one byte in hexadecimal starting at the memory address.

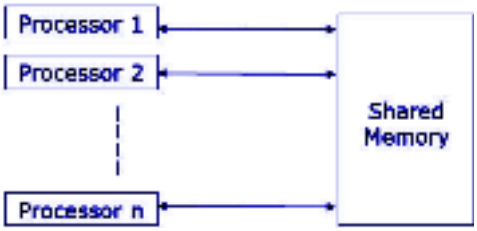


In this screenshot, I used info registers to look at the registers so that I could verify if my code was done correctly. R1 has the correct hexadecimal representation of -60, but the decimal value is not correct. R2 has 11 because I stored the value 11 in it. R3 has 16 because I stored the value 16 in it. R4 has the value 5a, which is 90 in decimal because I added 3+11+16, and then subtracted -60 from that, which results in 90. Also, in my cpsr register, which represents the flags, there is 0x10, which means the result is valid.

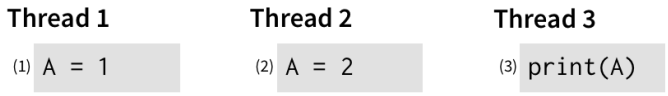
Parallel Programming Skills Foundation: Joan Galicia

- **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
  - A task is a set of instructions given to the computer to execute on the CPU.
  - Pipelining is a type of parallel computing that breaks down a task into parts that can be run on different processors
  - Shared Memory is a location in the computer that can be accessed by using the bus and issued by tasks.
  - Communication is the transfer of data between Parallel tasks usually by the bus if a network
  - Synchronization is the part of parallel computing of when a task is completed it cannot proceed further until the rest of the tasks have also been completed
- **Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.**
  - The following classifications have two dimensions
  - SISD is a single computer where only one instruction and one data is being executed by the CPU during a clock cycle
  - MISD is a type of parallel computing where each processing unit operates on data independently on a separate instruction stream as single data is fed into multiple processing units.
  - SIMD is a type of parallel computer where the processing units execute the same instruction at ant clock cycle
    MIMD is a type of parallel computer where each processor may be executing different instruction streams and multiple data streams.
- **What are the Parallel Programming Models?**
  - There are several parallel programming models shared memory, threads, distributed memory/message passing, data parallel, hybrid, single program multiple data, and multiple program multiple data.
- **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
  - Uniform Memory Access is memory used equally by tasks and if a part of the memory is updated then the other processors also know that it is updated.
  - Non Uniform Memory Access links two or more SMPs and this allows the direct access of memory from one SMP to another. The only issue is that it does not allow equal share time of memory.
  - Open mo uses the Uniform Memory Access model because all the processors share the same physical memory and all tasks have an equal access time to all memory words.
- **Compare Shared Memory Model with Threads Model?**
  - Shared Memory is where tasks share a common address space where they can be read and written, and most or all of the program's variables are accessible. Threads is a type of Shared Memory Programming. It contains a single "heavy weight" process that can have multiple "light weight", concurrent execution paths, but it does not have a common state.

Shared Memory Model



Thread Model



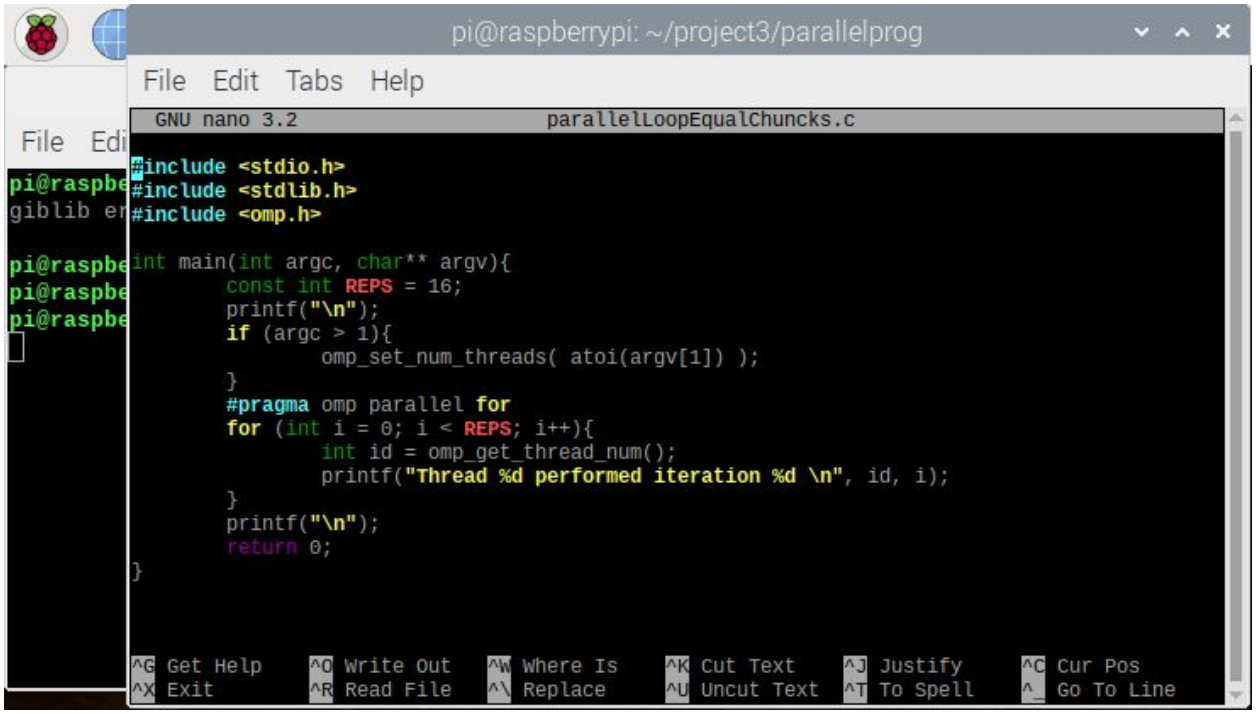
➢ **What is Parallel Programming?**
- Parallel programming is the way the computer breaks down a task into similar parts that can be executed simultaneously in threads.

➢ **What is system on chip (SoC)? Does Raspberry PI use system on SoC?**
- SoC is a new processor that is very similar to the CPU, but the difference is that the SoC has almost all the chips the CPU uses. The CPU is the core of the computer, but the SoC would be the computer. SoC contains the memory, GPU, USB controller, power management circuits , and wireless radio.
- Yes, the raspberry pi is powered by a SoC which is the Broadcom BCM2835 SoC.

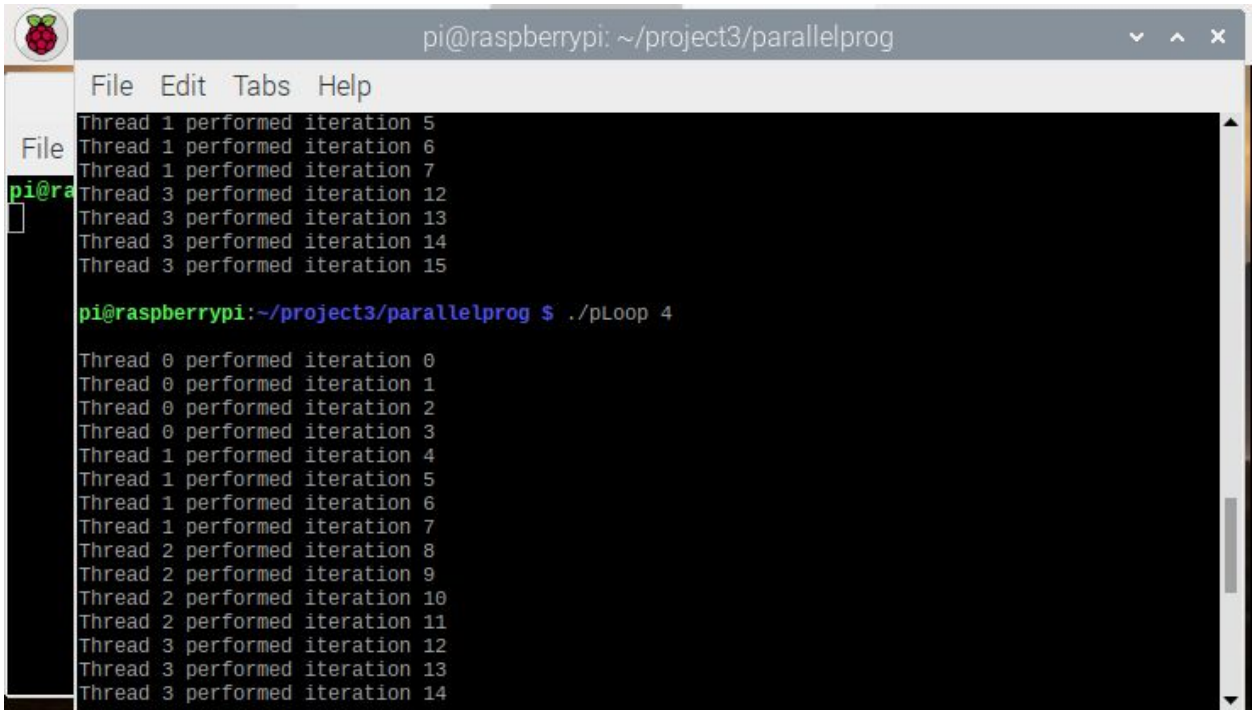
➢ **Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**
- The advantage of the SoC is its lower power consumption, shorter wiring which allows data to be transferred faster, and it becomes less expensive to produce.

Parallel Programming Basics:

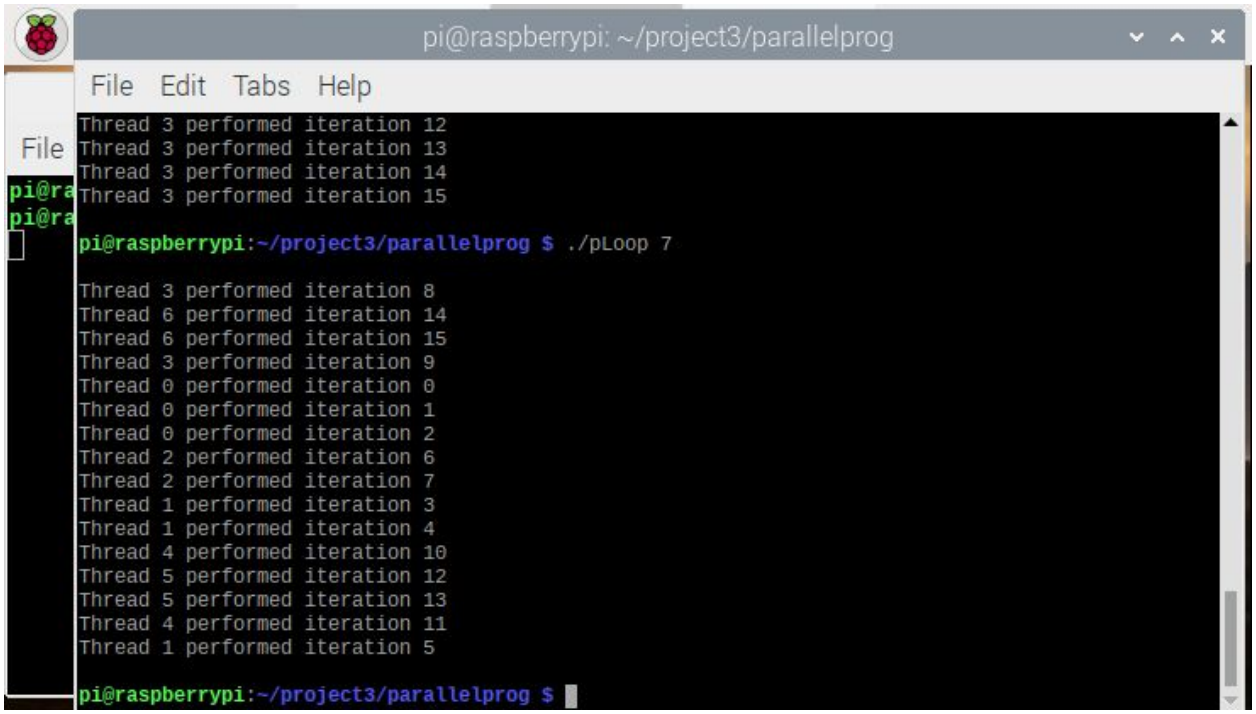
Part1



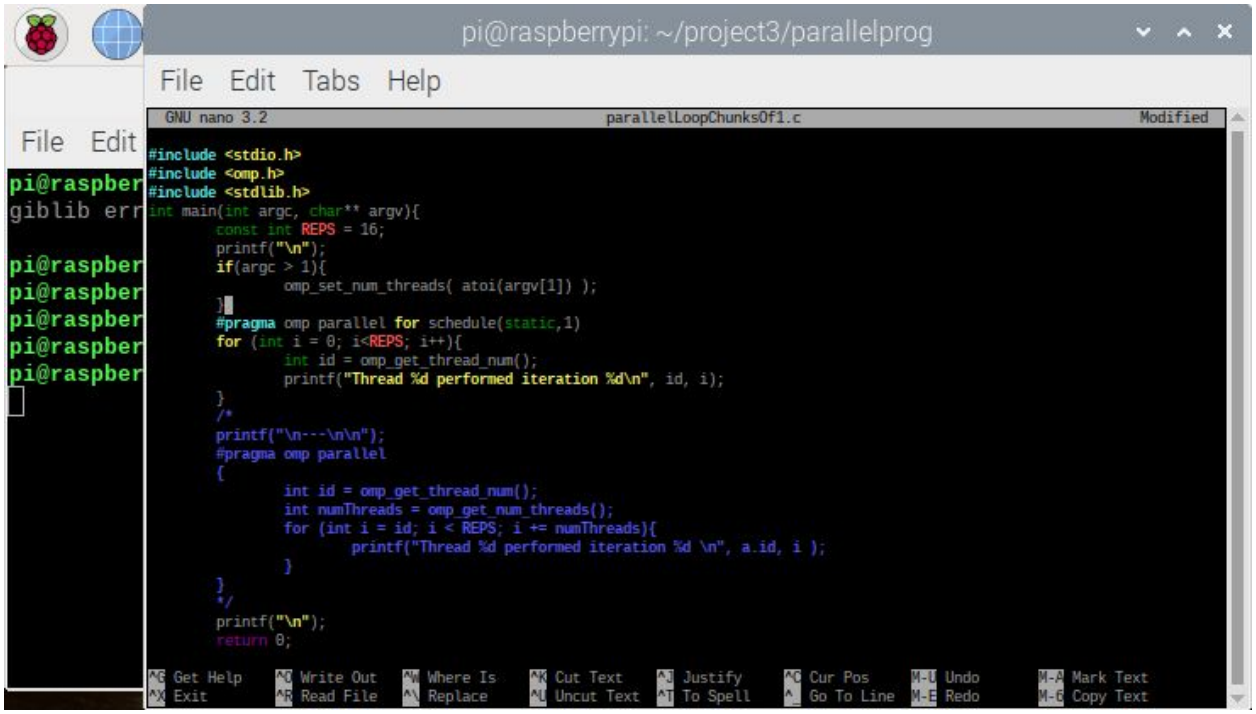
This screenshot is the program for running a parallel computation with a loop. For this program, it's to show how an alternative method of data decomposition is run and the executions can be seen below.



For this execution, the program was iterated 16 times with 4 forked threads. The order of the iterations do not matter as they are not dependent on calculations.

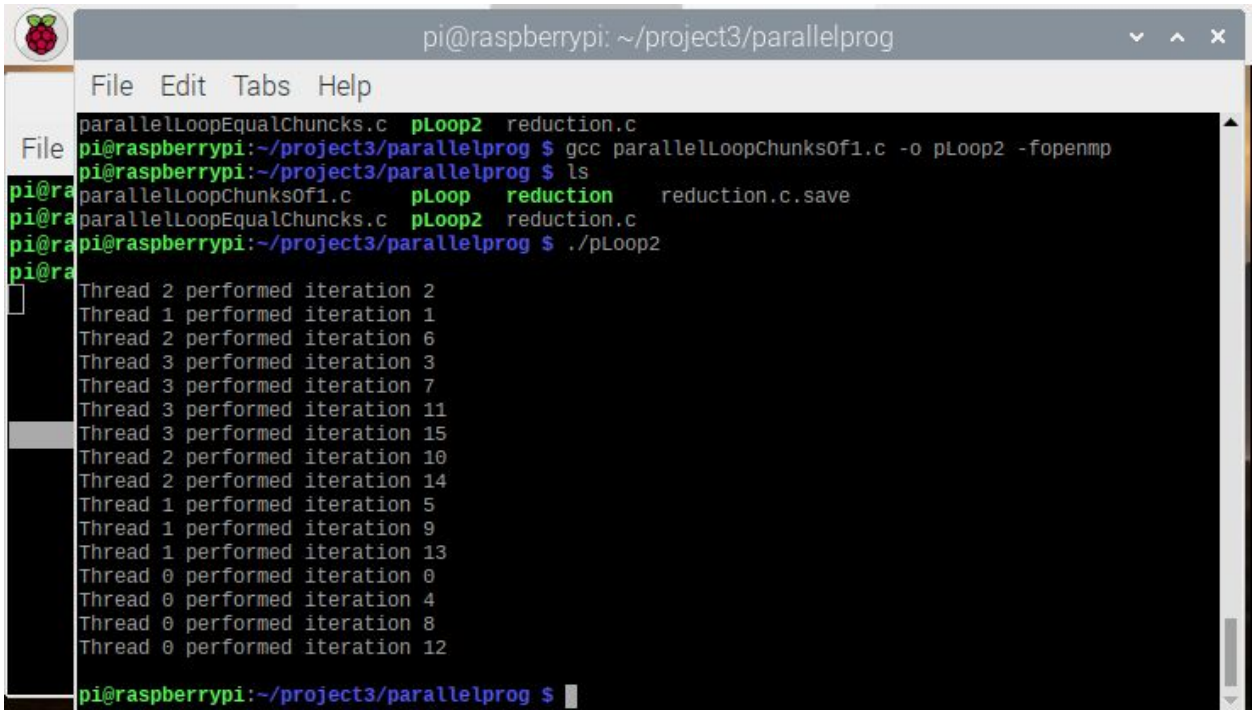


For this execution, the program was iterated 16 times with 7 forked threads. The order of the iterations do not matter as they are not dependent on calculations.

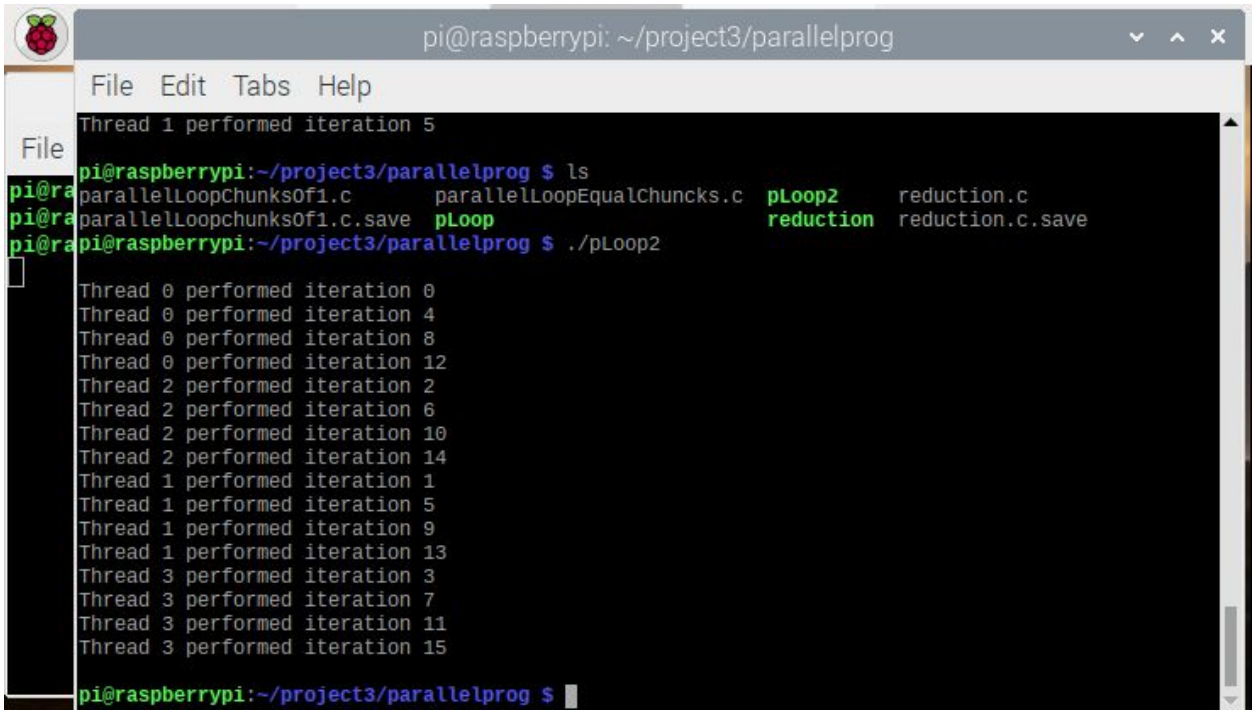


In this screenshot, it shows another loop program that shows an alternate way of parallel computing, but for an unequal amount of iterations. The difference from parallelLoopChunks is

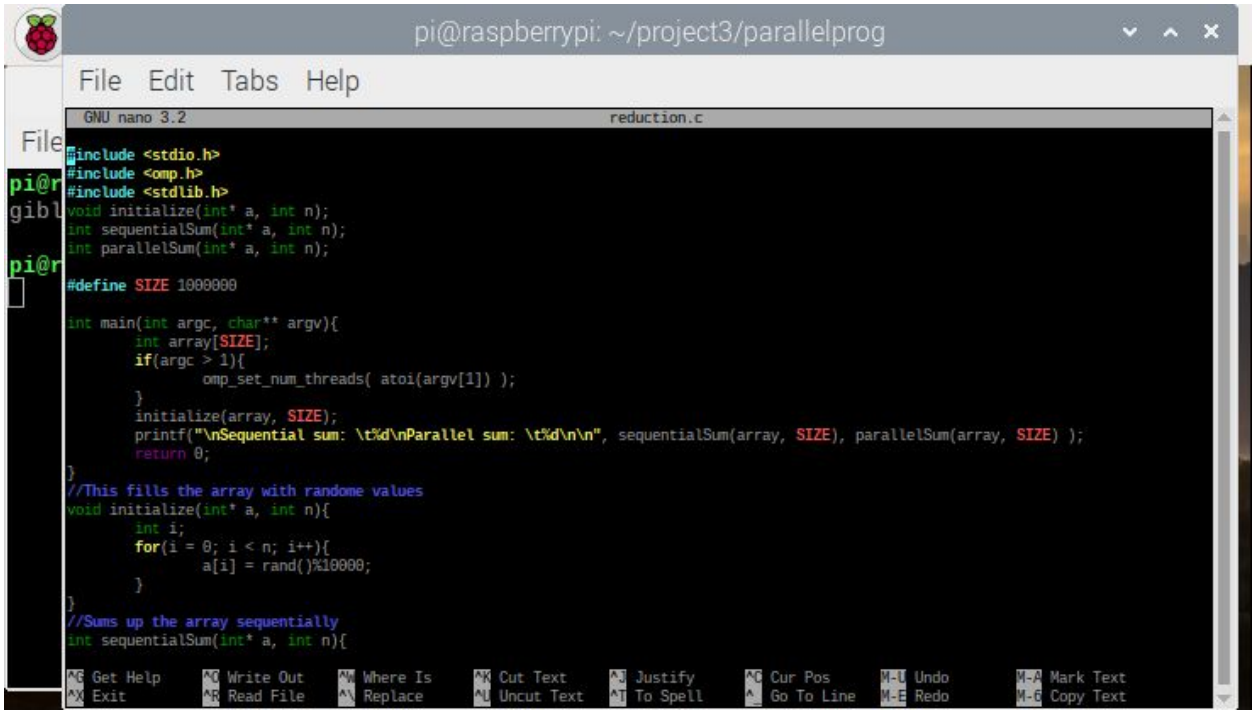
that this program evenly distributes one iteration of the loop and then the next one. To execute this program, there was the commented version which was simple and the uncommented version which was more complex. These versions can be seen below.



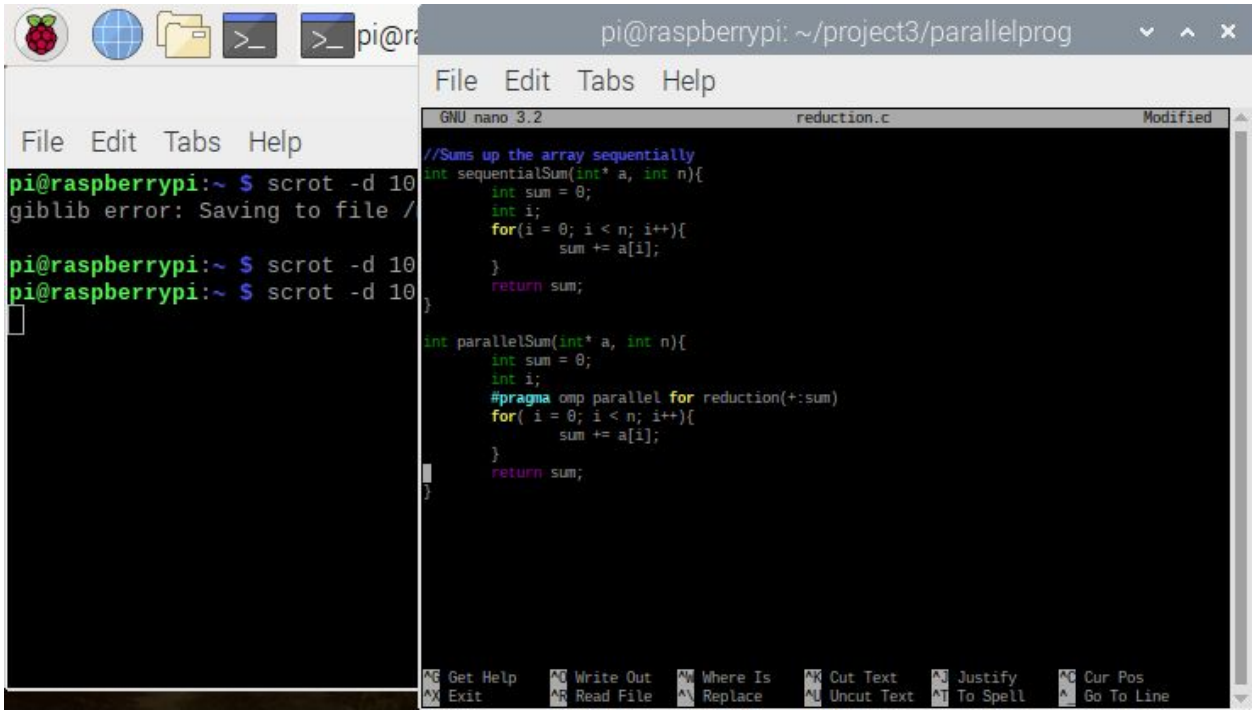
This output is of the commented version of equal chunks. This version was simple as it allowed for the program to execute every task but was restrictive.



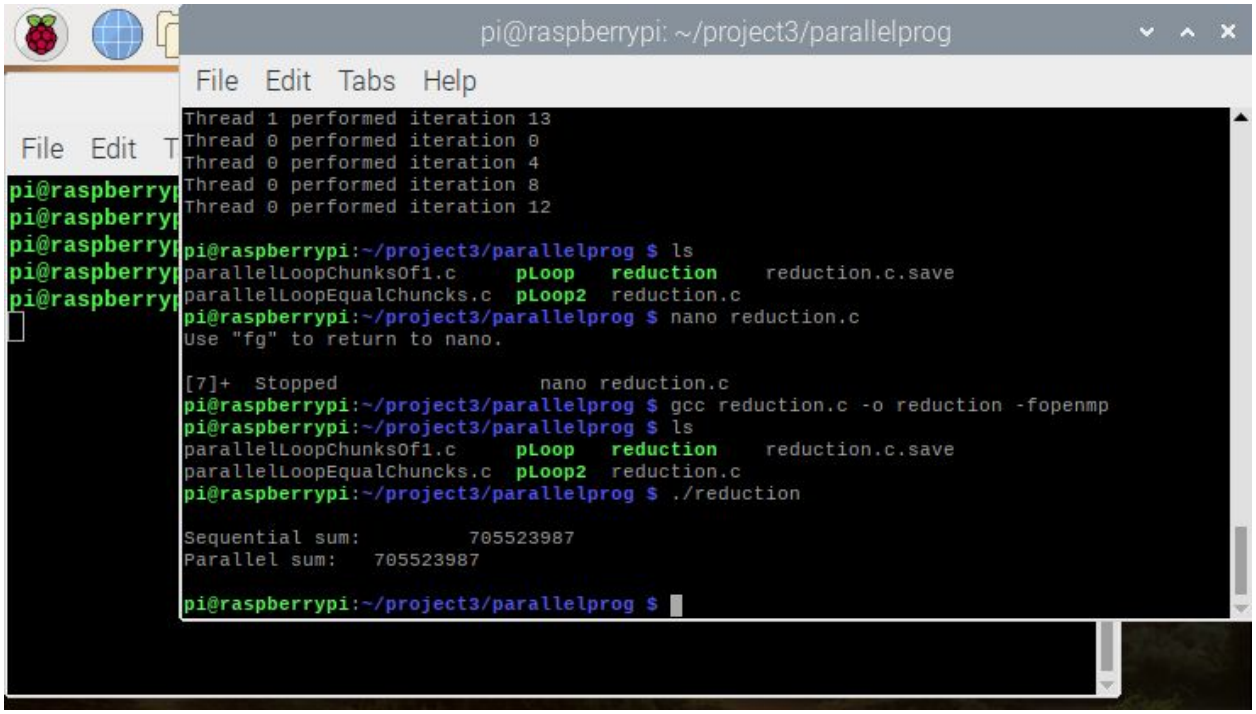
This output is of the uncommented version of equal chunks. This part of the program was more complex as it had the program run the loop several other times for each task. Even though it was a more complex code, it was less restrictive.



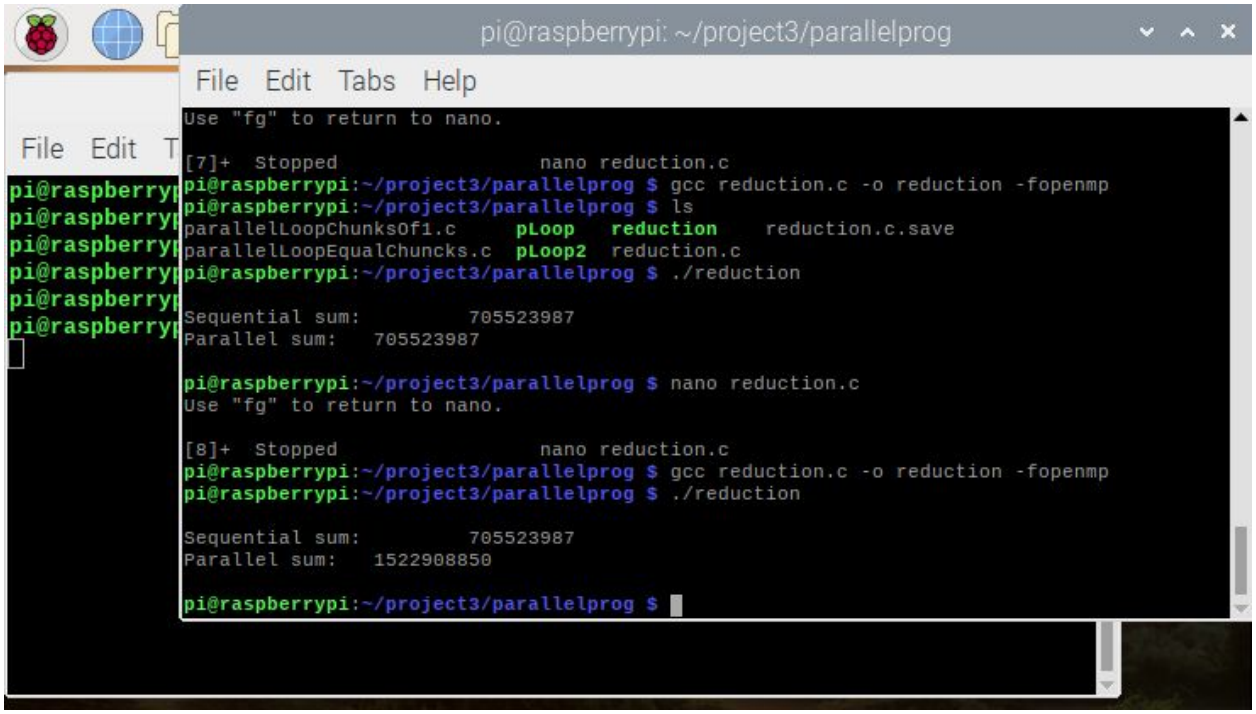
This program is another one of the parallel computations but it has a dependency. It has to update sequential sum while its running in the loop and to do this it has to perform each task dynamically.



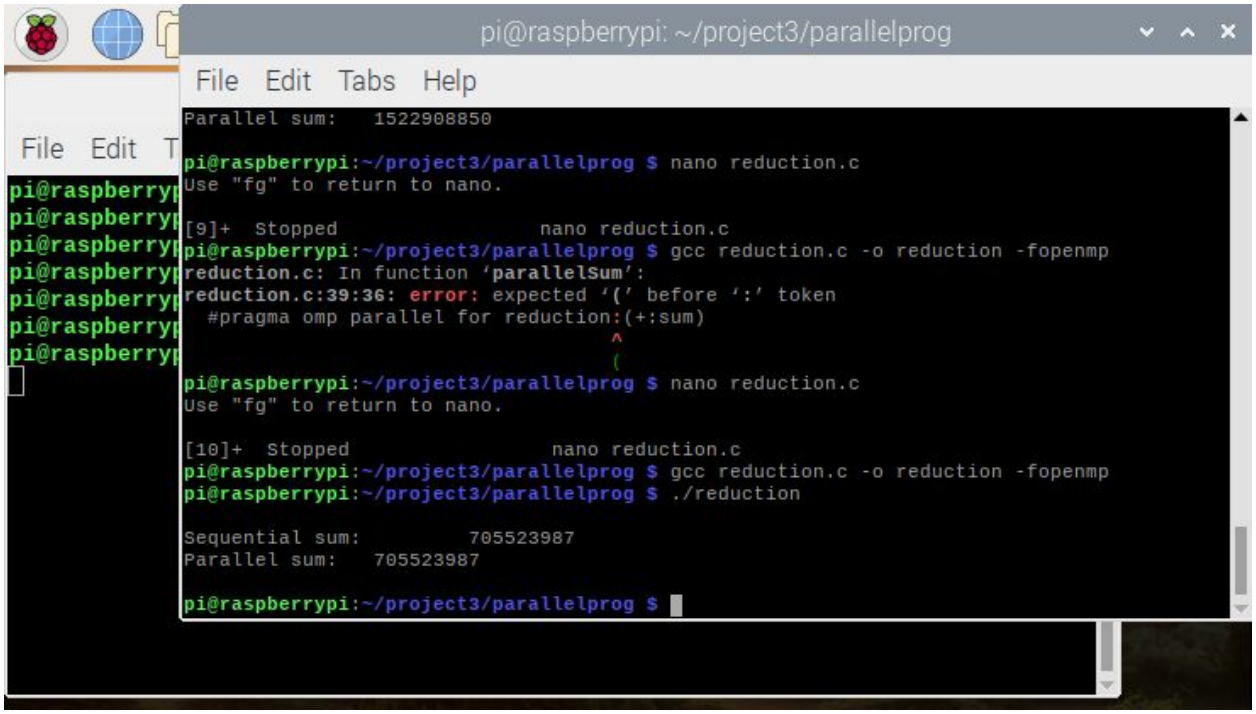
This screenshot is the other half of the reduction program where it shows the loop for parallel sum.



This is the screen shot for the reduction program with “pragma omp parallel for reduction(+:sum)” commented out. The output is printed out at the end if the screenshot where the parallel sum is the same as the sequential sum.

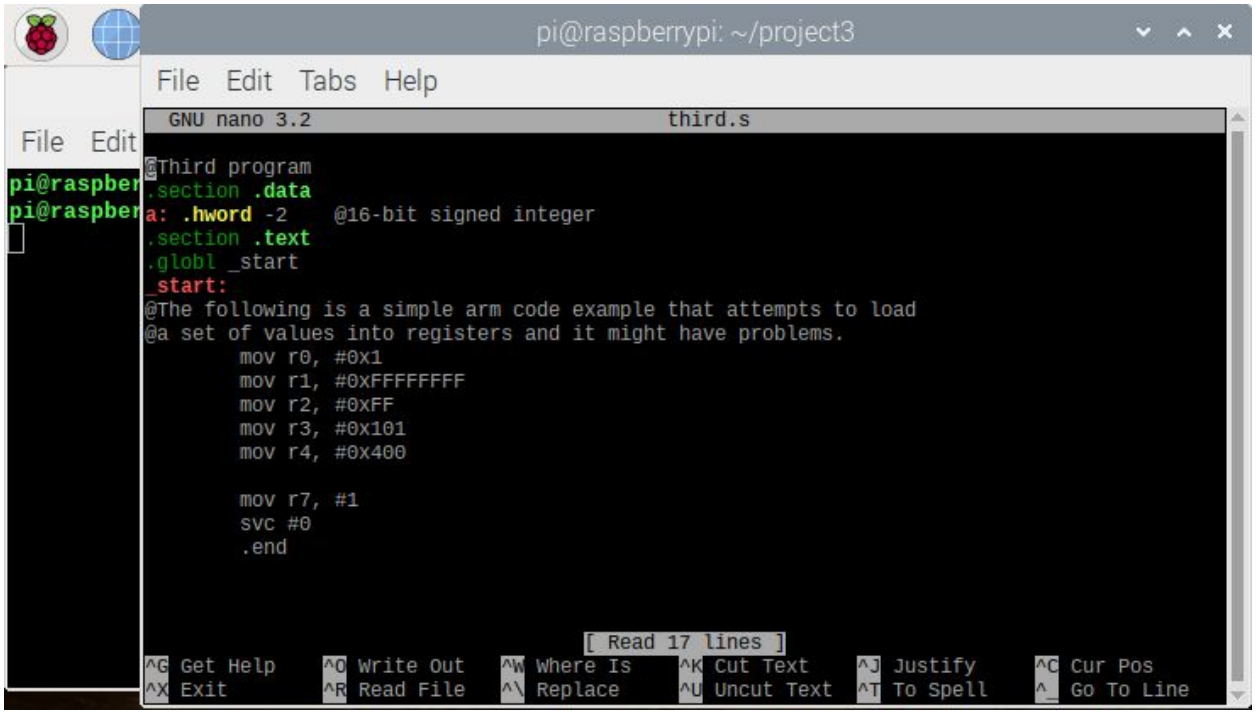


This is the screen shot for the reduction program with “pragma omp parallel for” commented out. This output is shown at the bottom where the parallel sum is different from the sequential sum.

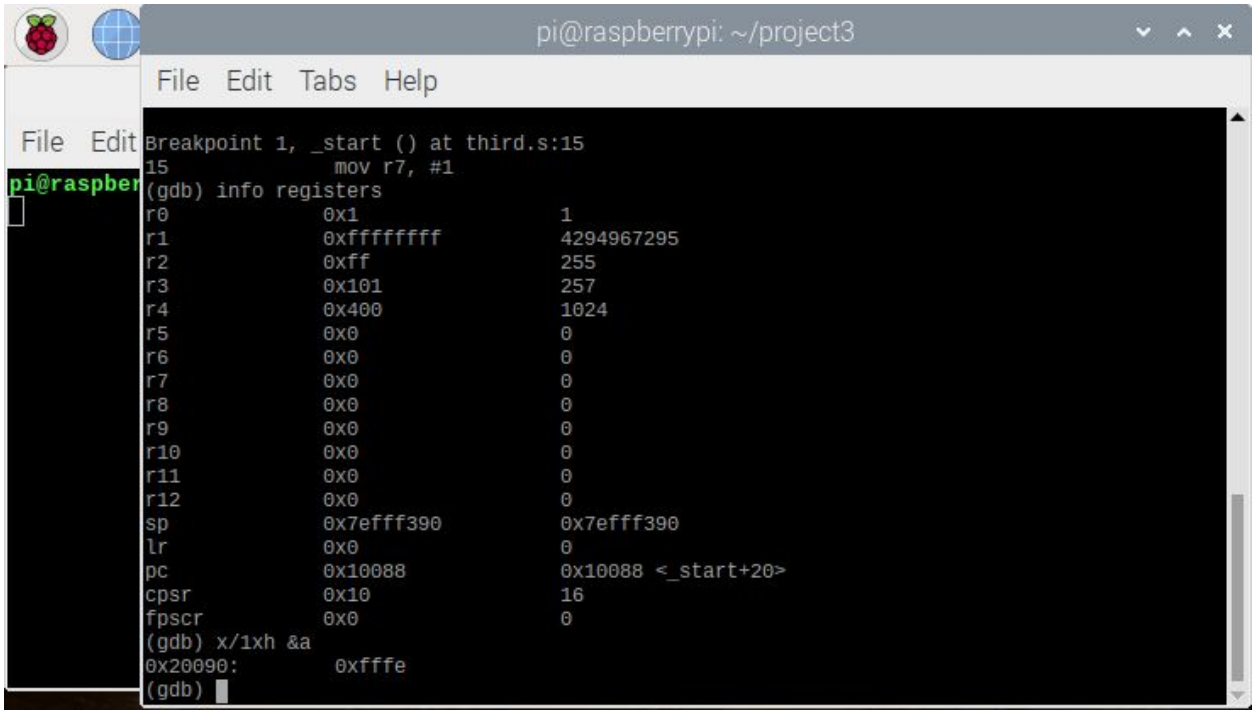


This is the screen shot for the reduction program with “pragma omp parallel for reduction(+:sum)” uncommented. The output is shown at the bottom of the screen.

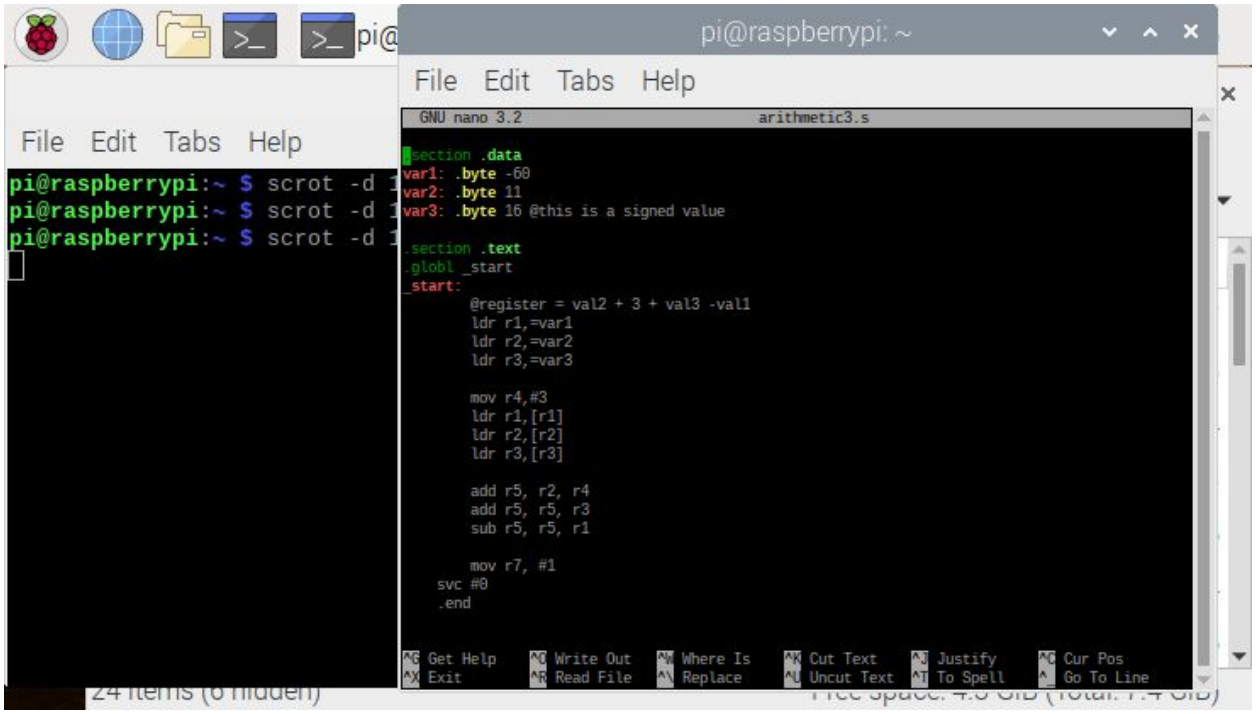
ARM Assembly Programming:



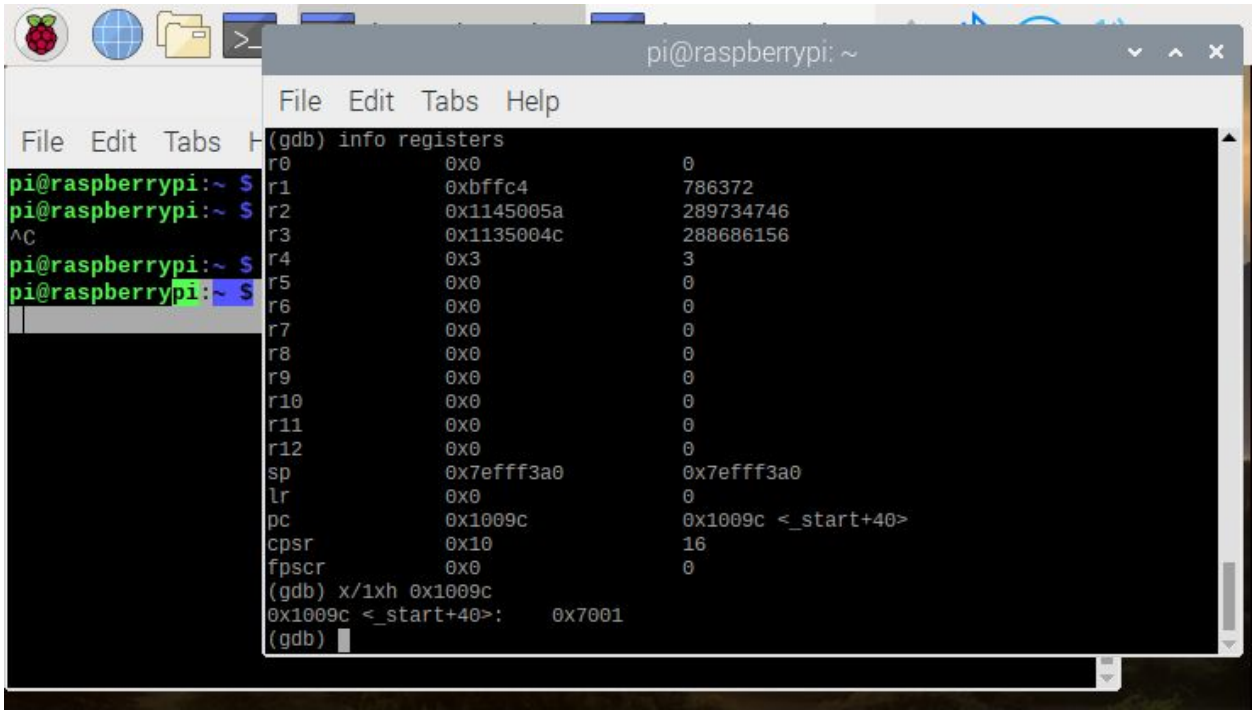
This program is to show how arm assembly loads bytes and signed values into the registers. There was an issue where ".shalfword" did not function properly and gave the error "SIGNSEV' where the variable 'a' was not read by the machine.



This is the output to the third.s program where the values are displayed. For register r1 it shows the value 429467295 and when it's converted using 2's complement it shows its the signed value of -1. Then when finding the memory address for variable 'a' it gave 0x20090: 0xfffe, and when this hexadecimal is converted using 2's complement it produces the signed value of -2.



This screenshot shows the code of the program arithmetic3.s where I add and subtract some values together but instead using a combination of signed and unsigned byte values. The issue with this program is that it would not execute properly when loading the values in the memory to the register. Also there was the issue of byte not being a signed value because when executed it would give the error “SIGNSEV” and numerous corrections did not help solve the problem.



This is the execution of the program arithmetic3 where the value in hexadecimal is 5a, and when it is converted using 2's complement it gives the value 90. The answer is located in register 2 at the end of the hex value.

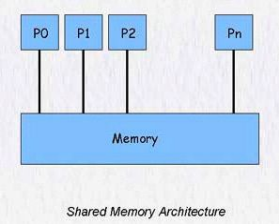
Parallel Programming Skills Foundation: Andre Nguyenphuc

- **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
  - Task - A section of work that has to be completed. An example of this would be a program-like set of instructions that is executed by a processor.
  - Pipelining- Breaking down a task into steps which are then performed by different processor units.
  - Shared Memory- A computer architecture where all processors have direct access to a common physical memory.
  - Communications- This is how the parallel tasks exchange data one way this is accomplished is when there is a shared memory bus, this is not the only way data can be exchanged there are also other ways.
  - Synchronization- The coordination of parallel tasks in real time. There is an establishing of a synchronization point and tasks must wait until other tasks get to the same point.
- **Classify parallel computers based on Flynn's taxonomy.Briefly describe every one of them.**
  - SISD- The CPU executes only one instruction stream during any one clock cycle and only one data stream is being used.
  - SIMD- All the processors are executing the same instruction at any given clock cycle and each processor is working with a different data stream.
  - MISD- All the processors are executing different instruction streams and a single data stream is being used by multiple processors.
  - MIMD- All the processors are executing different instruction streams and every processor is working with different data streams.
- **What are the Parallel Programming Models?**
  - Shared Memory (without threads)
  - Threads
  - Distributed Memory/Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data
- **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
  - Uniform Memory Access (UMA) - A shared memory model when all the processors have equal access and access times to a shared memory.
  - Non-Uniform Memory Access (NUMA) - All SMPs (Symmetric Multiprocessor) have their own memory. All the SMPs are then linked through buses and the SMPs can access memory of another SMP. However not all processors have equal access time to all memories.

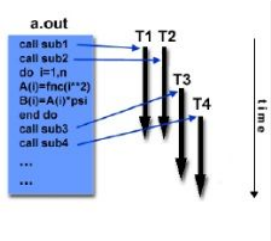
- UMA is used by OpenMP because the memory is being shared and there is equal access and access times.

➢ **Compare Shared Memory Model with Threads Model?**

- Shared Memory Model - All the processors have access to a common memory where they are all able to read and write to. To prevent deadlock and race conditions locks or semaphores are used to control the access to the common memory.



- Threads Model - A single process has multiple smaller threads which use the shared resources of the main process to complete the application. The threads communicate with each other by global memory. This is done through synchronization to make sure that different threads are not updating the global address at the same time.



➢ **What is Parallel Programming?**

- Parallel Programming is when tasks are broken down into smaller parts and the processors execute the solutions simultaneously.

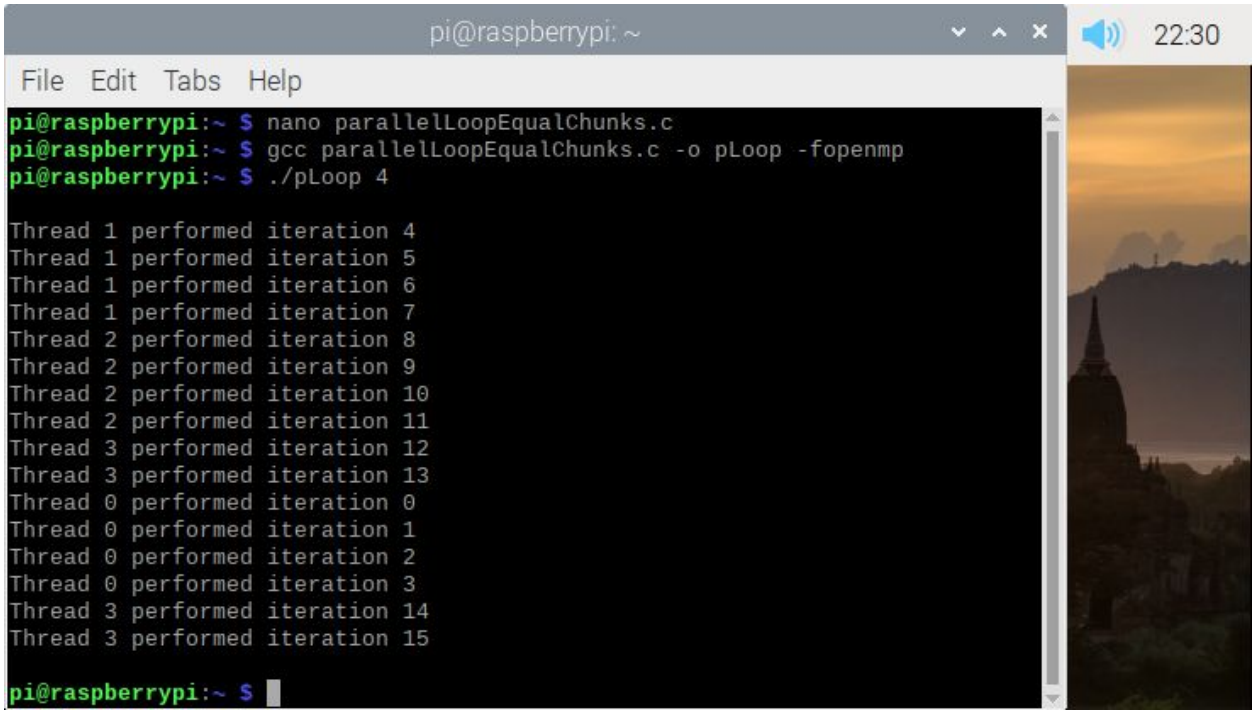
➢ **What is system on chip (SoC)? Does Raspberry PI use system on SoC?**

- SoC stands for system-on-a-chip and is similar to a CPU but instead of having separate parts to make a computer a SoC contains all the components in a single chip. It contains a CPU, GPU, memory, USB controller, power management circuits, and wireless radios. Yes,Raspberry PI uses system on SoC.

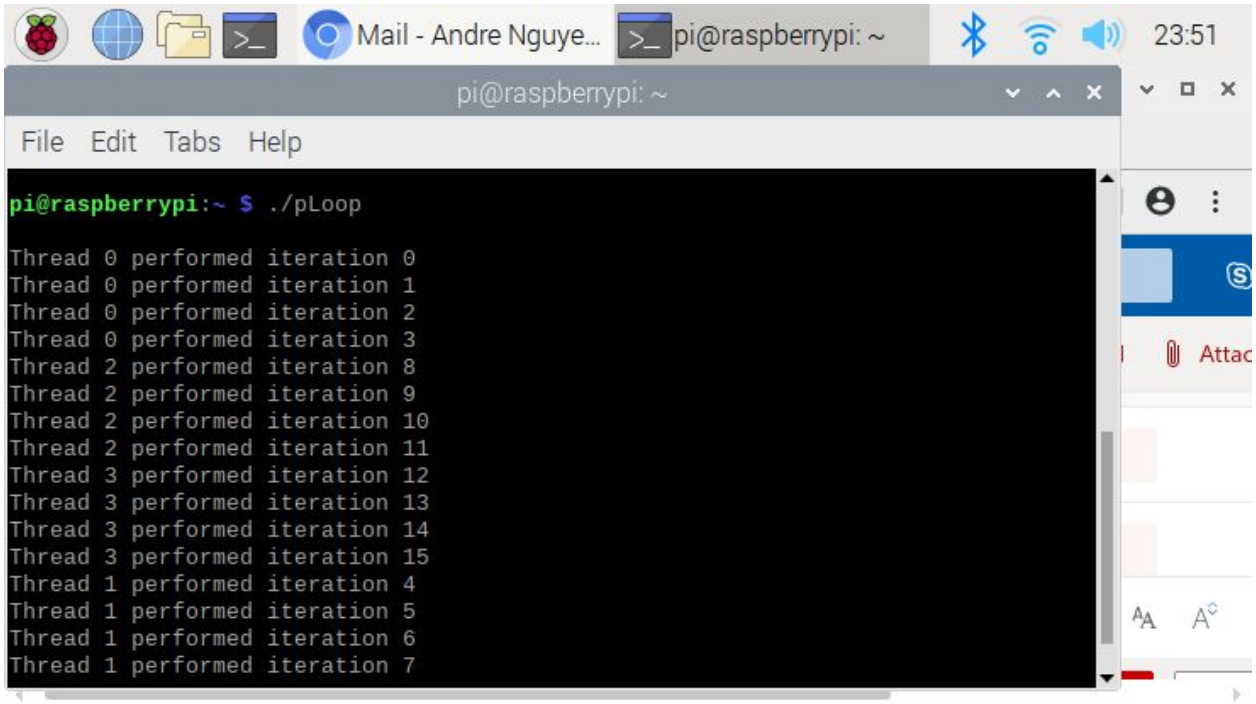
➢ **Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

- The main advantage a SoC has is size because even though it is only a little bit larger than a CPU, it has way more functions than a CPU. With a CPU you need other parts to make a computer, but with a SoC because it has all the parts within you can make a computer in small devices like smartphones. More advantages include it uses less power and it is cheaper due to less number of physical chips.

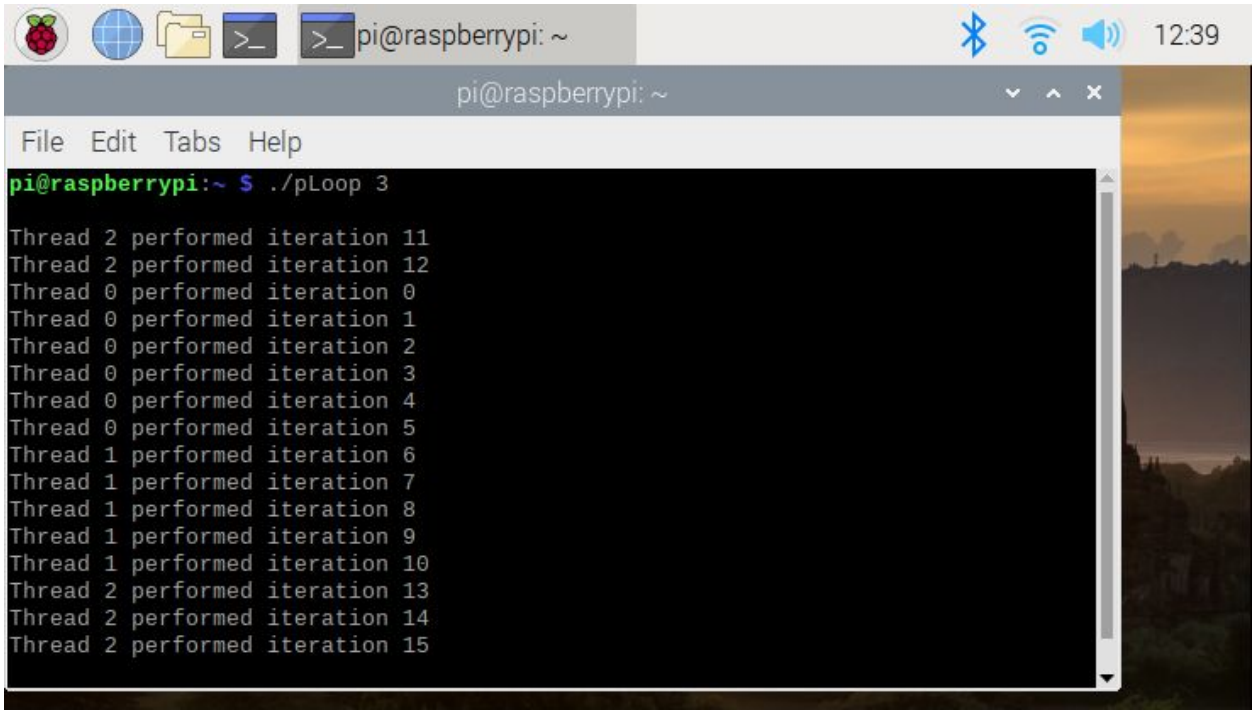
Parallel Programming Basics: Andre Nguyenphuc



In this screenshot, I compiled the program and then ran it with ./pLoop 4 which then performs 16 iterations from 4 different threads. In this screenshot, you can see that the order does not matter because it prints whatever is done first.

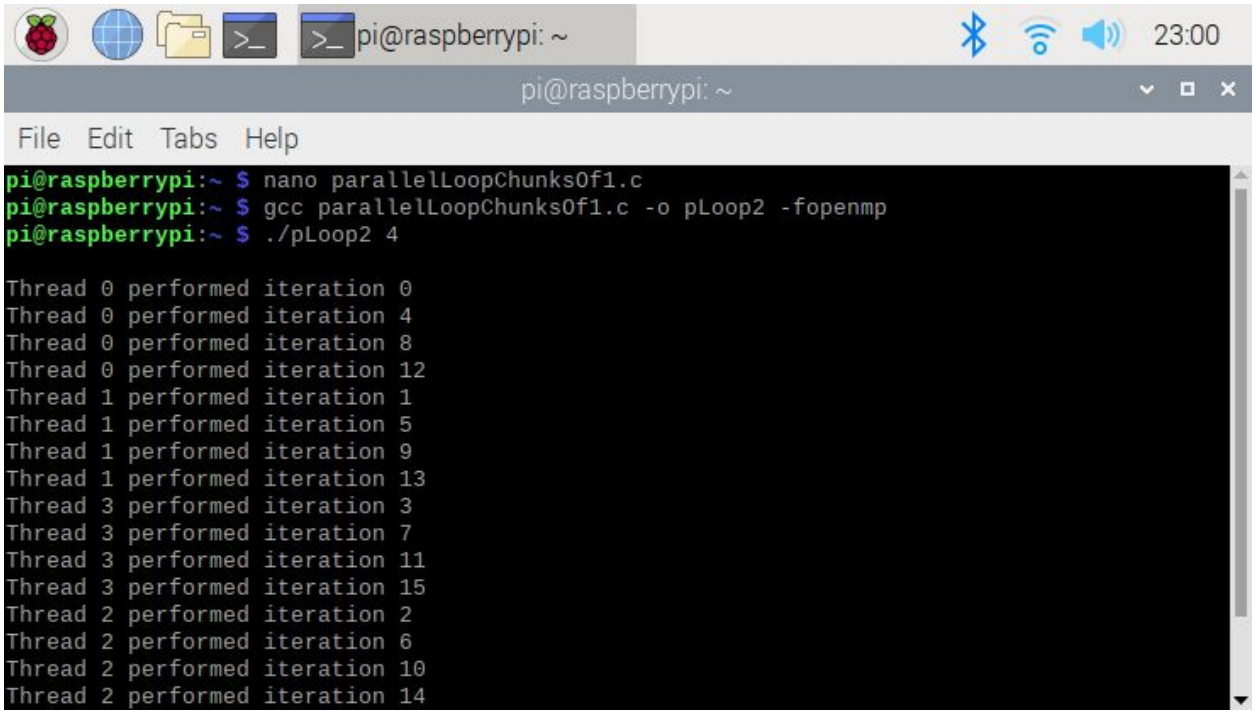


In this screenshot, I tried to not put a number for the ./pLoop, and I got 16 iterations from 4 threads. If I had done ./pLoop 3, I predict it would do 16 iterations,but with 3 threads.

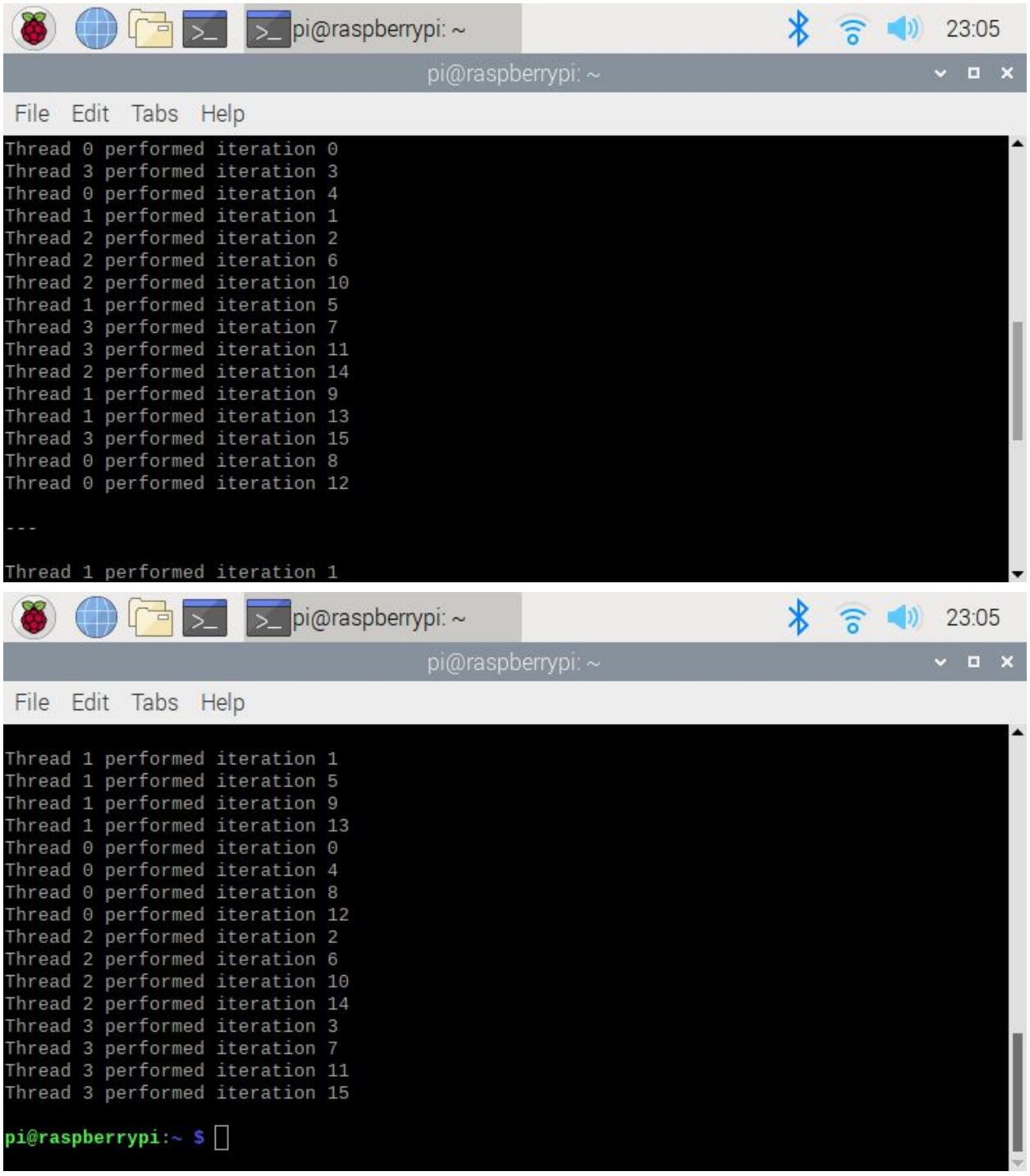


I tested my hypothesis and I was right there are only 3 threads but it still performs 16 iterations. It seems like it tried to evenly divide the iterations, but it can not since there are only 3 threads so thread 0 had to perform the extra iteration.

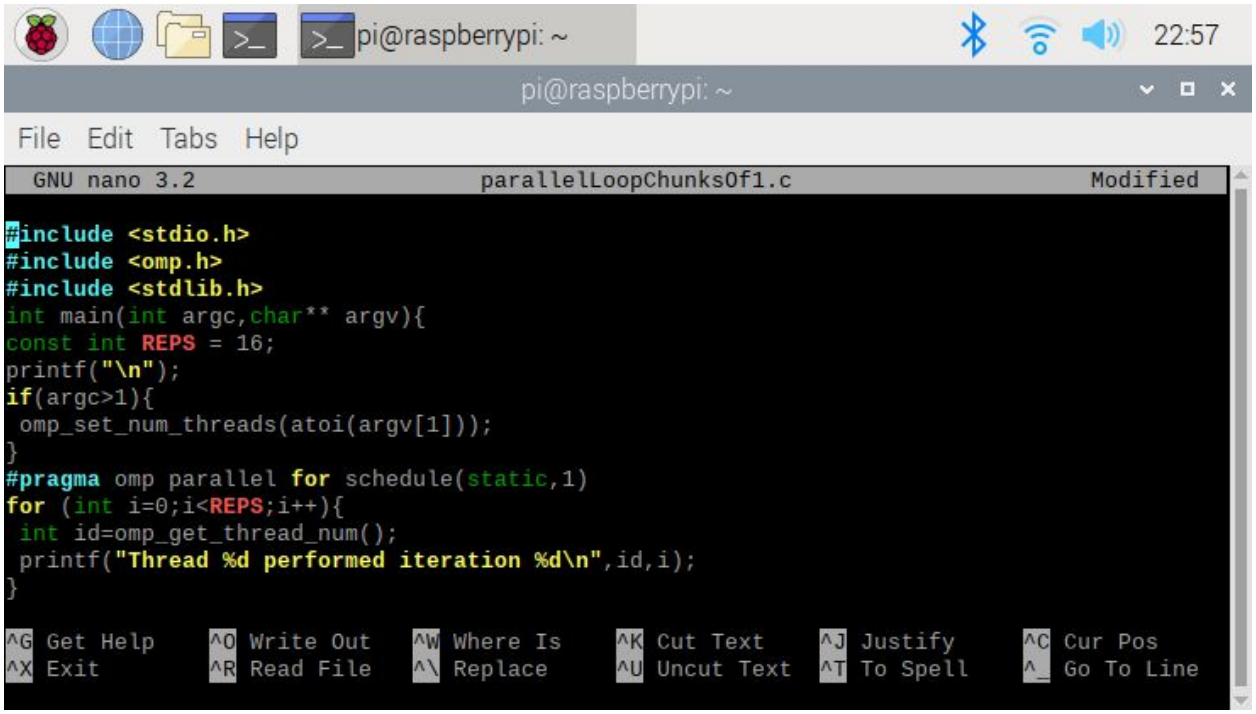
Part 2 of Parallel Programming:



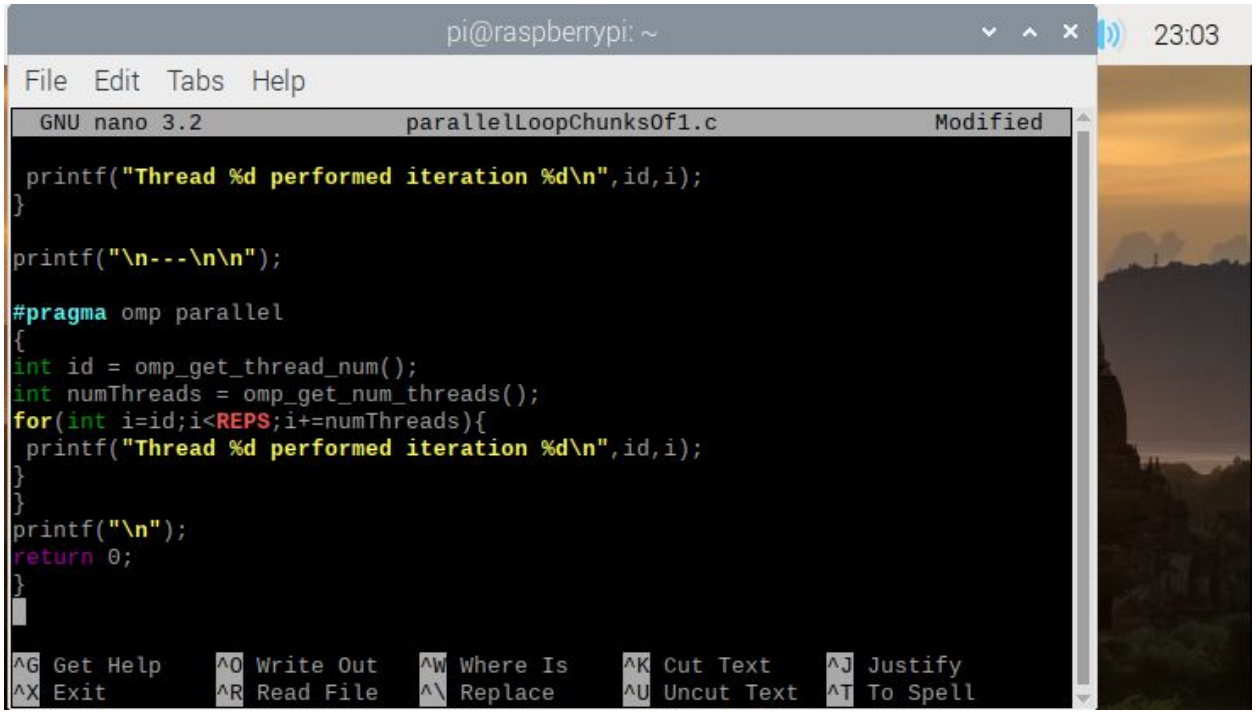
In this screenshot I kept the commented part of the parallelLoopChunksOf1 and I got 4 threads performing 16 iterations.



In the two screenshots above I uncommented the “commented out” part and I got 4 threads performing 16 iterations. The difference between the first loop and second loop is that the second loop although more complex it is less restrictive. This is shown in this first picture which is the second loop the order of threads performing the iterations is more random. While in the second picture although order of threads is still random the threads performing the iterations have more of a pattern.

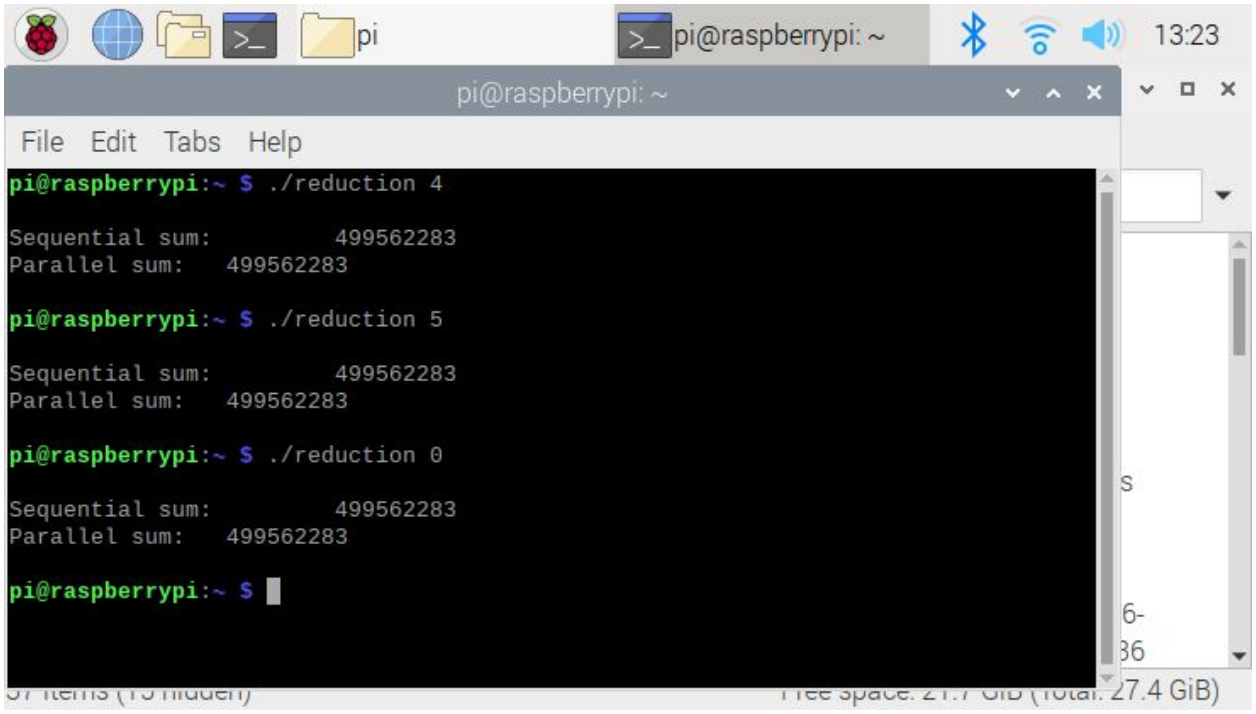


I believe the first loop has a pattern due to there being a static as the keyword which schedules each thread to do one iteration of the loop in a regular pattern.

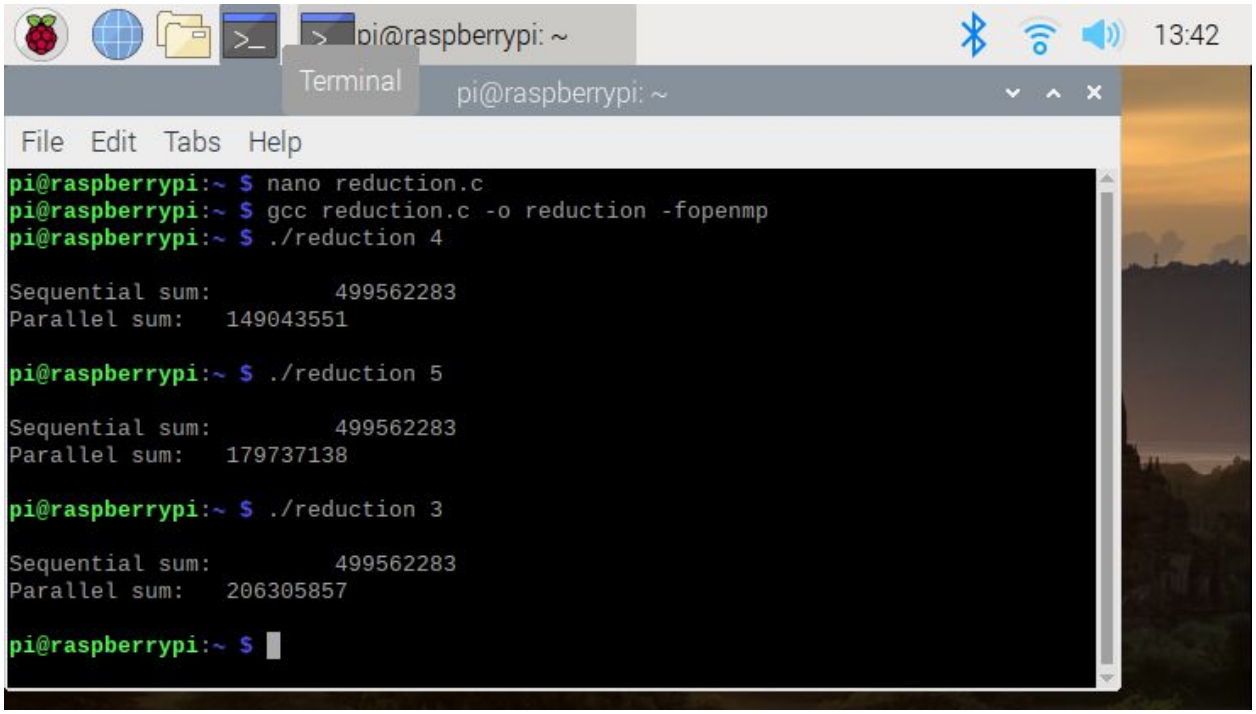


In the second loop there is no static or schedule which allows for the second loop's order of threads performing iteration to appear more random or with no pattern.

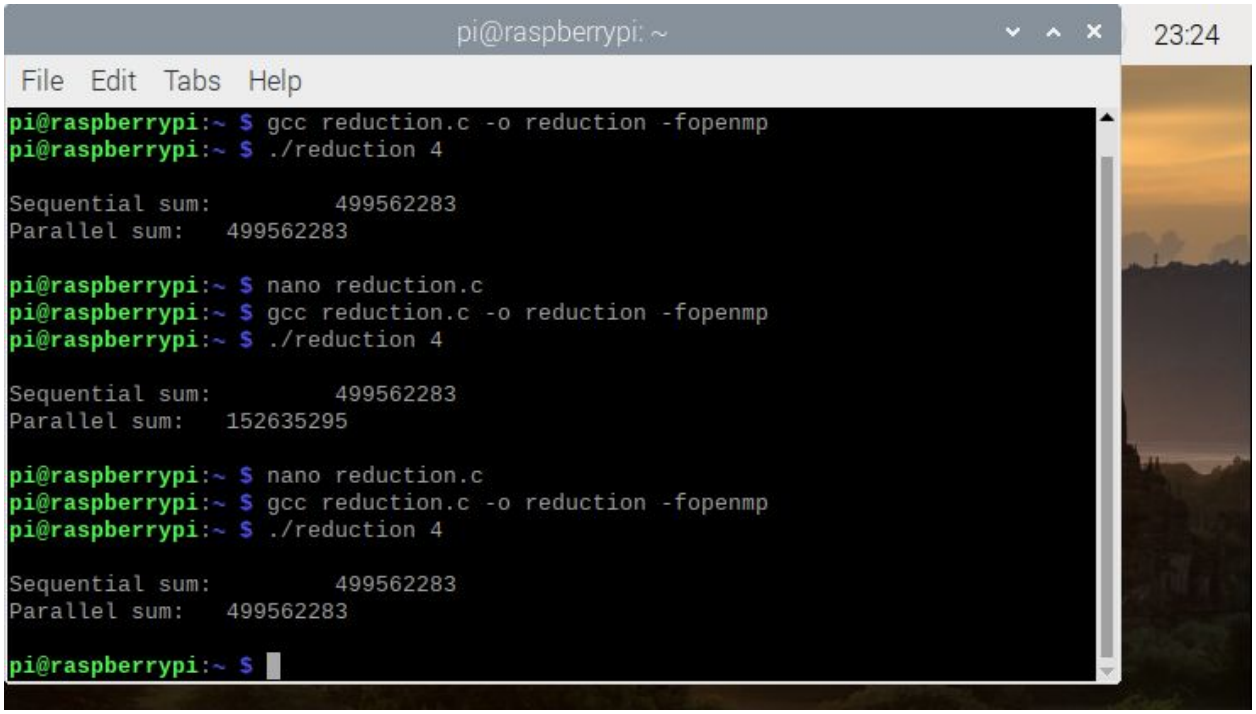
Part 3 of Parallel Programming:



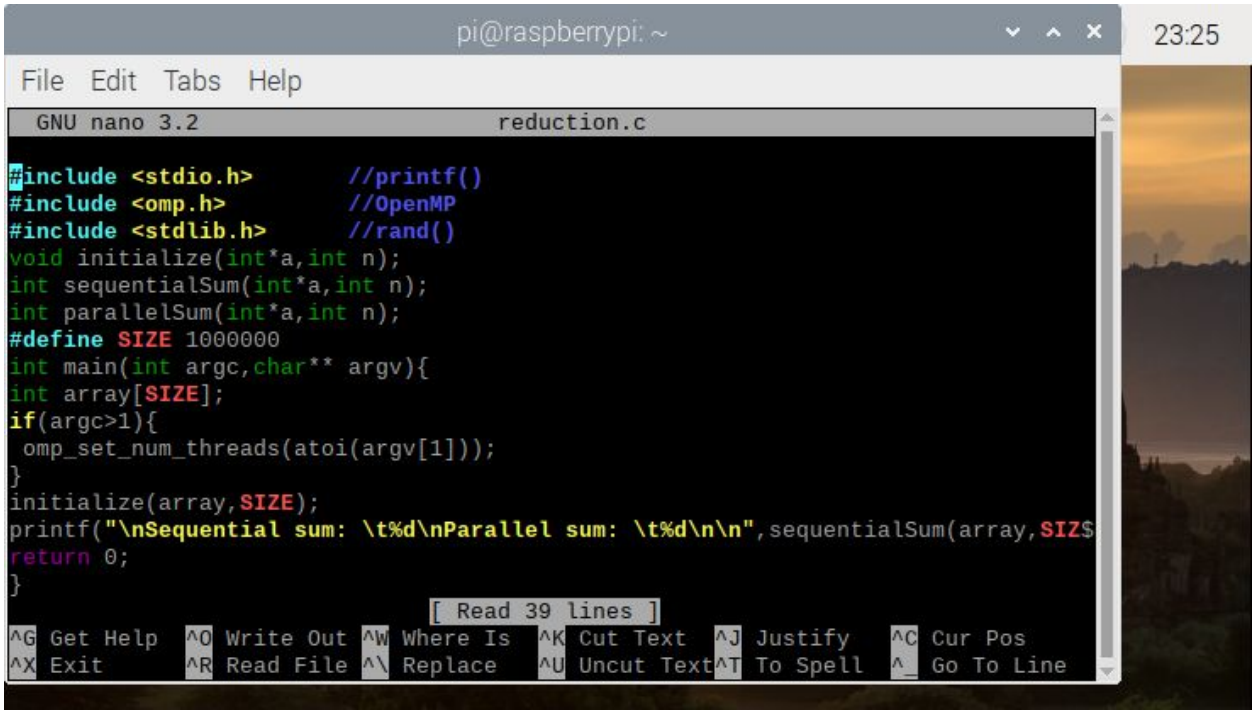
In this screenshot I first tested 4 and got the sequentialSum and parallelSum to match. I then tried different values for the number of threads and I noticed that I still get the same number as 4 (this is done with the comments still in line 39).



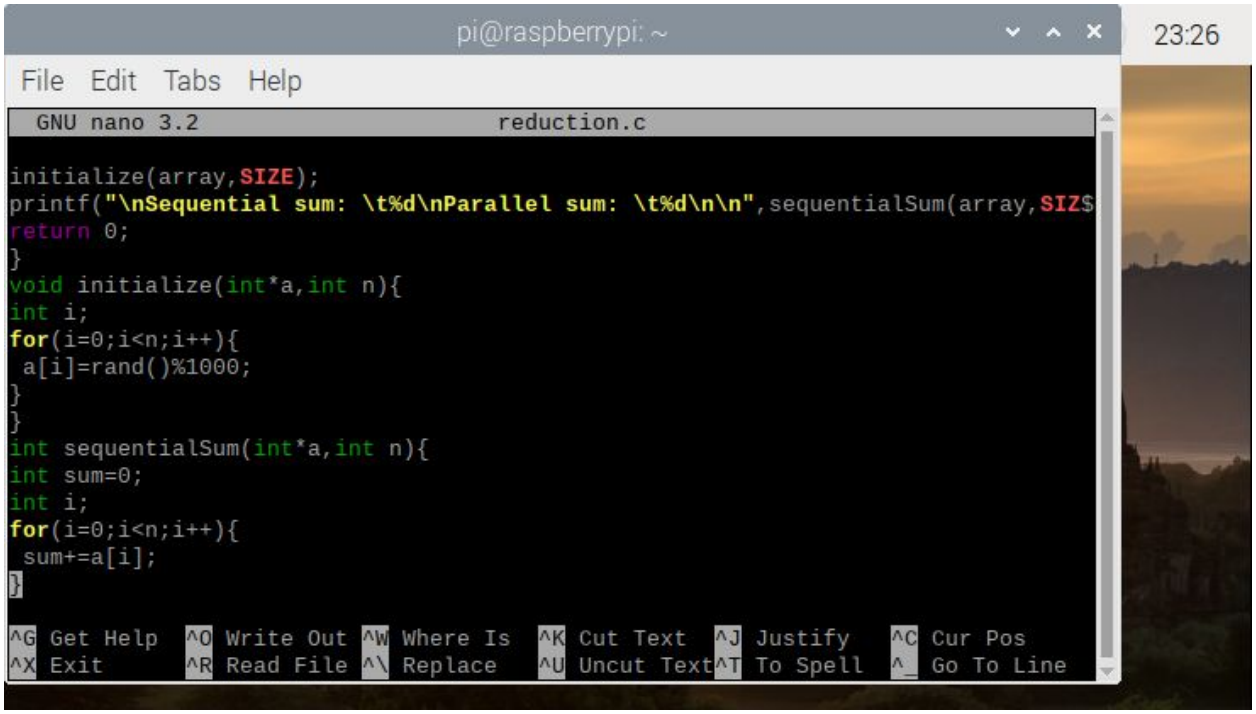
In this screenshot I first test 4 and sequentialSum and parallelSum did not match this is because I still had the reduction(+:sum) still commented. I then tried different values and all though sequential sum was the same for all of them I noticed parallel sum was not the same.



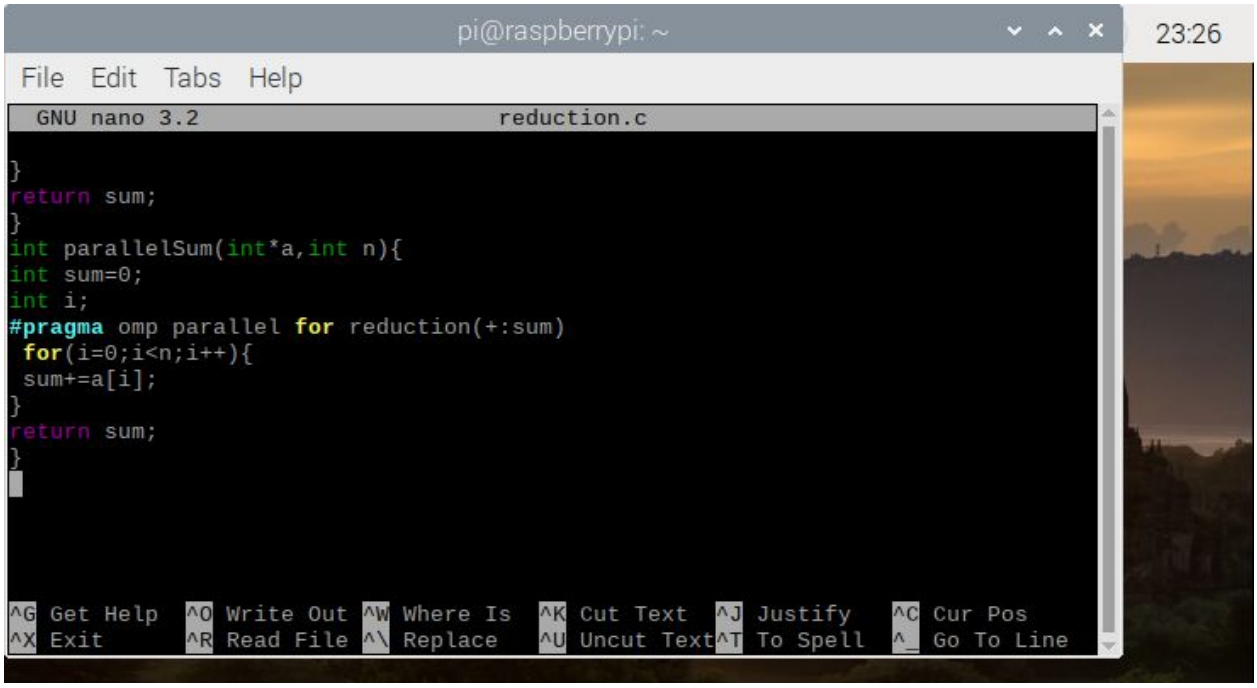
In this screenshot I ran the reduction program 3 different times. The first time in line 39 it had comments in front of #pragma omp parallel // for reduction (+:sum) and I got the same answers for sequential sum and parallel sum. Once I removed the first set of comments in line 39 I noticed the answers for sequential sum and parallel sum did not match. Then I removed the second set of comments and the answers matched up again.



This is the first part of the reduction program and it mainly shows the initialization of sequentialSum and parallelSum.

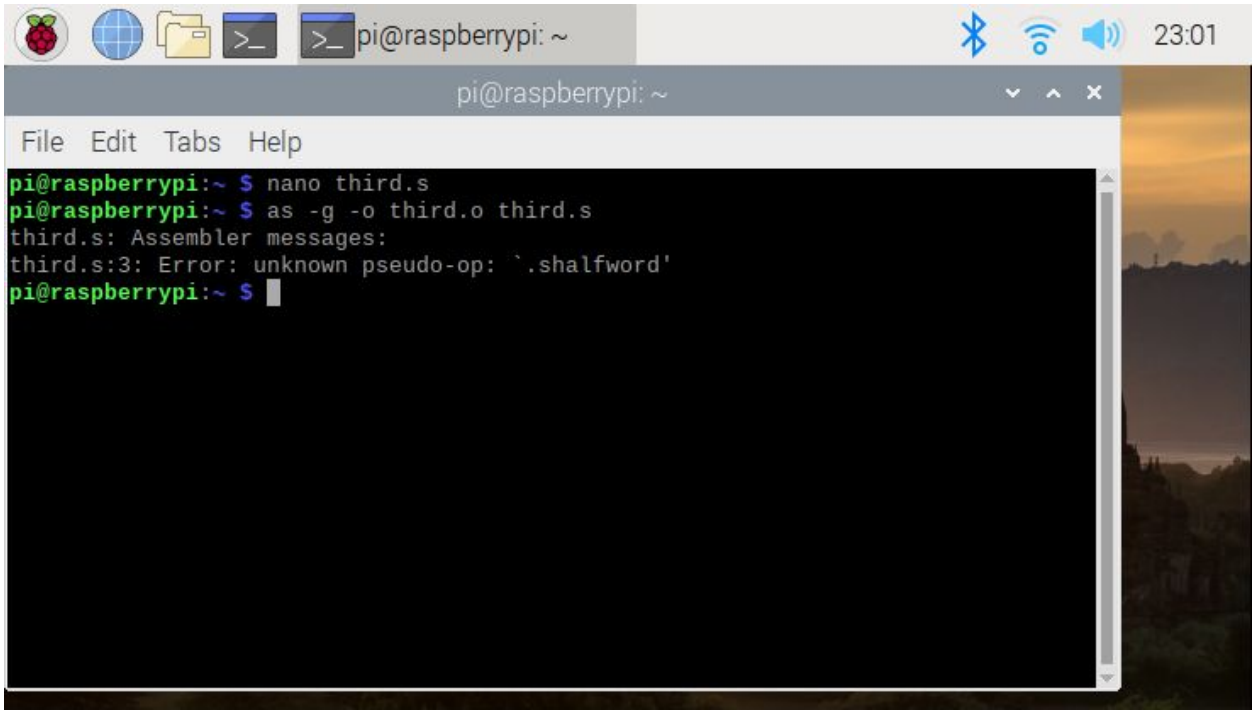


The second part of the program shows the for loops for initialize and sequentialSum.

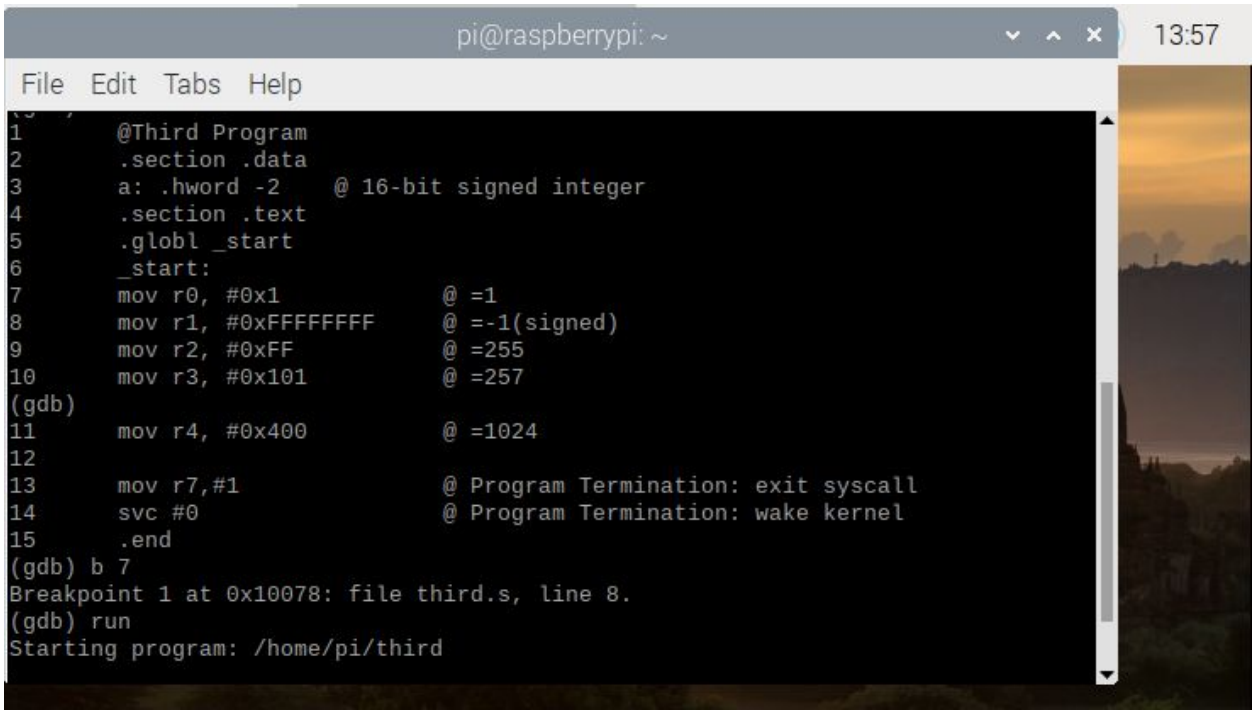


If the first // is removed from the #pragma omp parallel the sequentialSum does not match the parallelSum. When the second // is removed from the #pragma omp parallel for reduction (+:sum) the answers to sequentialSum and parallelSum match up again. This is because sum needs to be private to each thread and the reduction allows for it to be private. When each thread is finished, add all the individual sums to the final sum which is then computed.

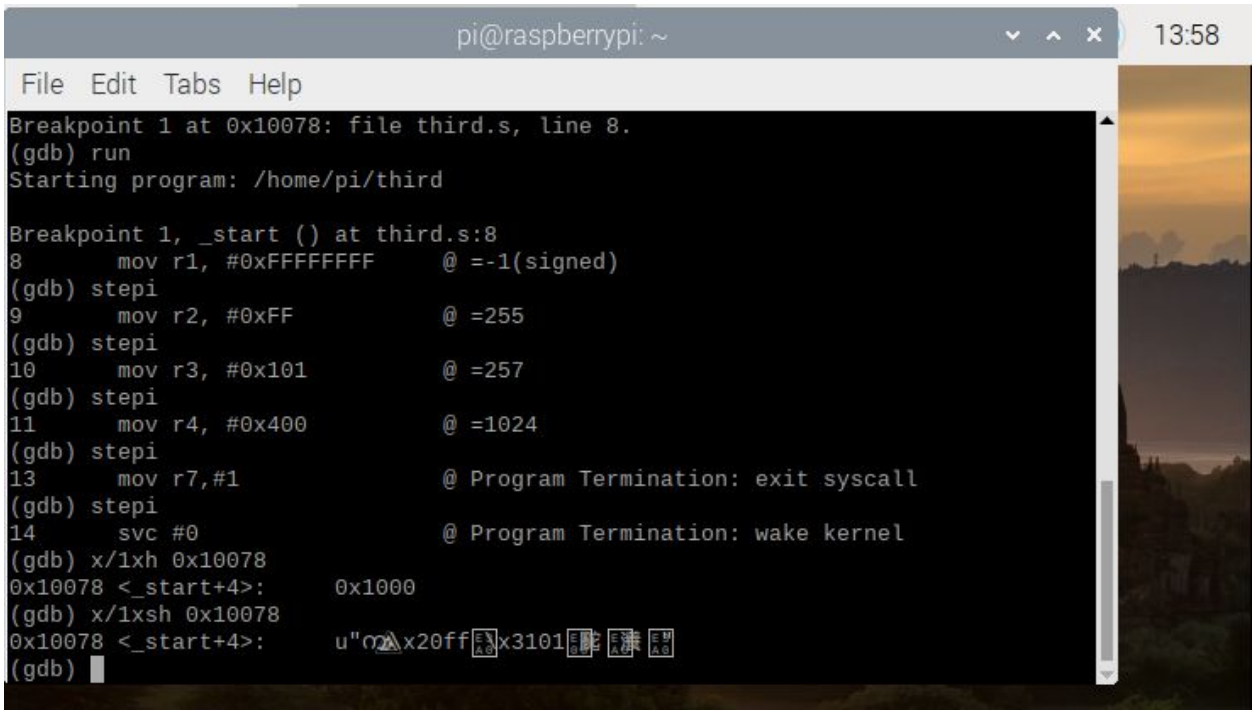
ARM Assembly Programming: Andre Nguyenphuc



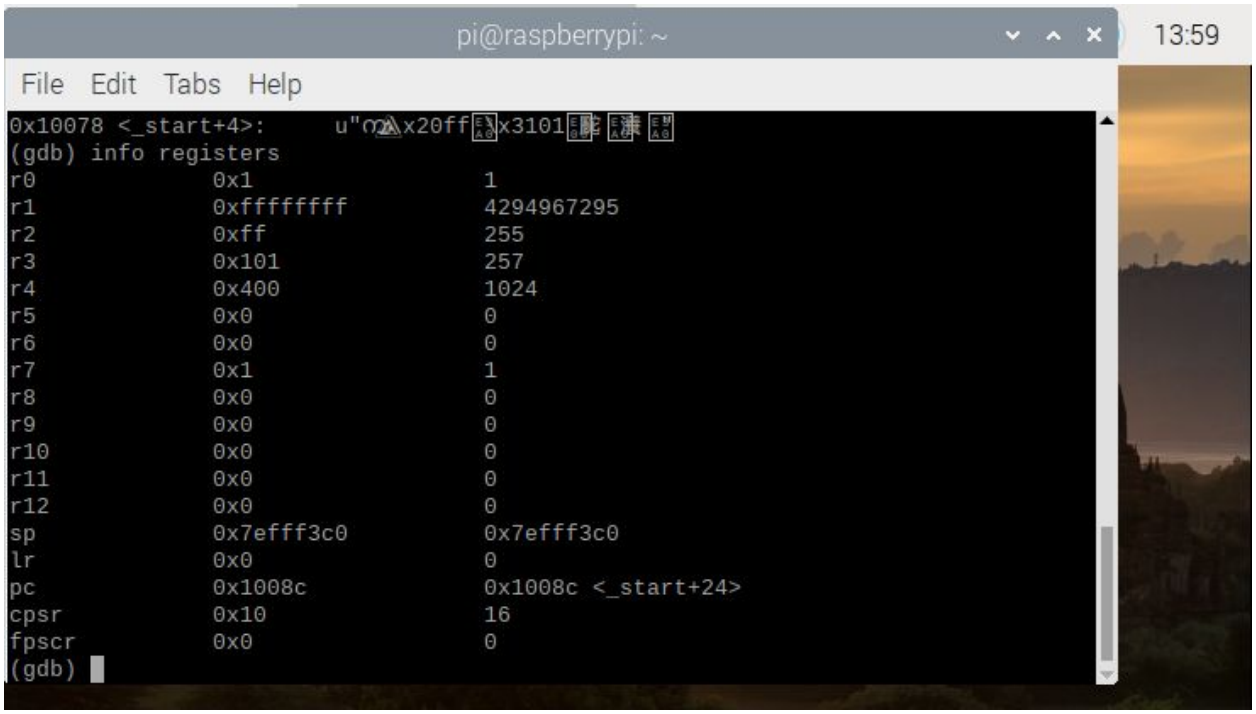
When first compiling the program I encountered the unknown pseudo-op error due to shalfword this is because shalfword is not the way to declare a 16-bit signed integer. To fix this error I changed the shalfword to hword and the program compiled for me.



This is my code for the third program as you can see I changed the shalfword to hword due to that being the way to declare 16-bit integers.

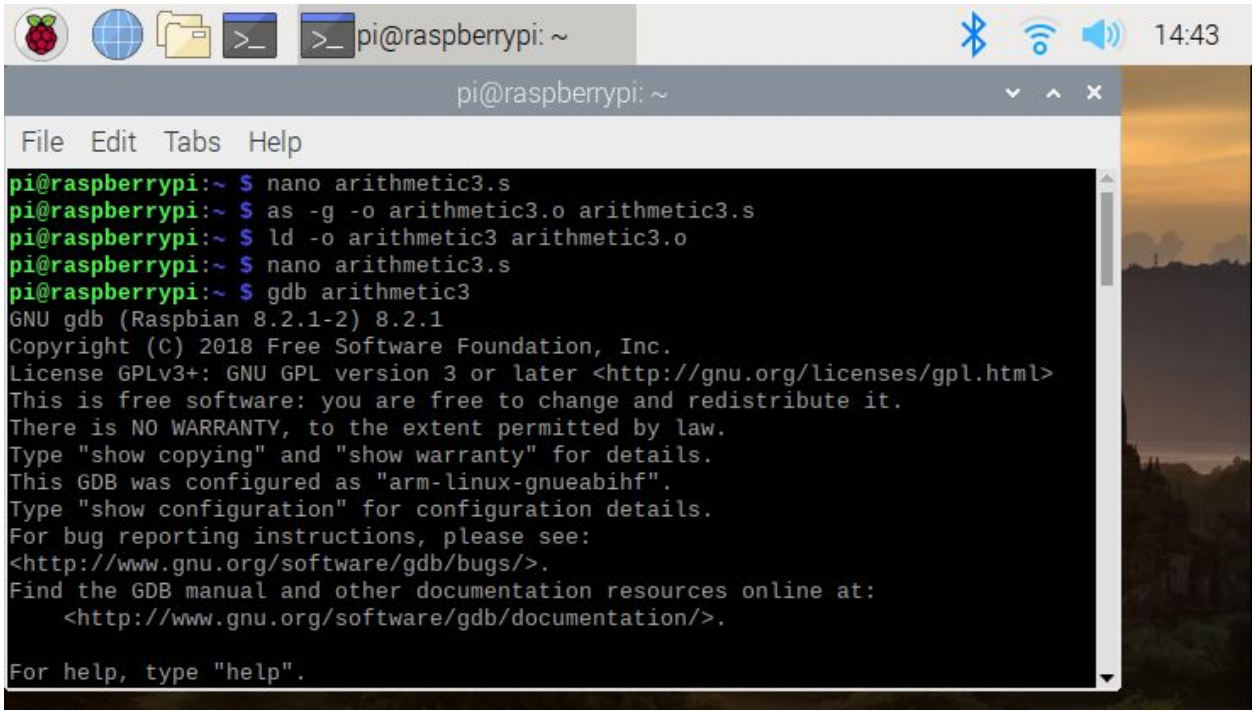


I set my break point and then I ran the program and stepped over and I noticed when trying to display my memory with x/1xsh I got symbols that I do not understand, but with x/1xh i got 0x1000.

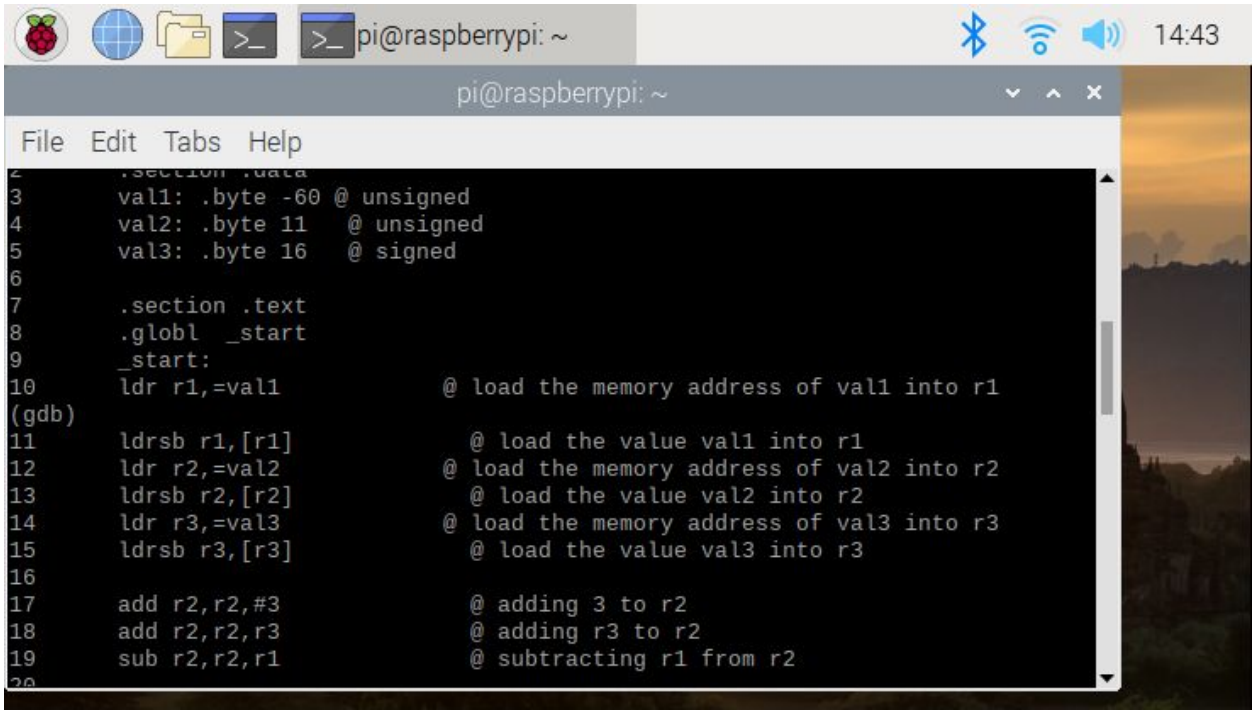


In this screenshot I am displaying my registers which have the values I assigned in my code.

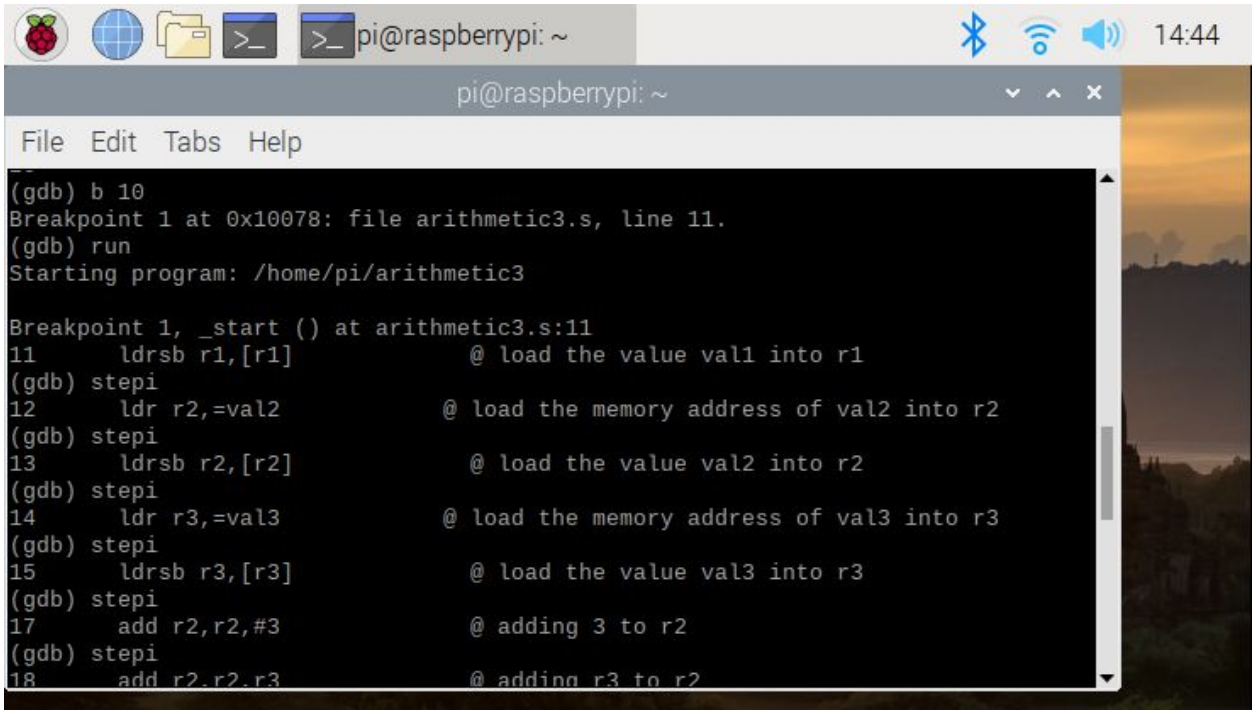
Part 2 of ARM Assembly Programming:



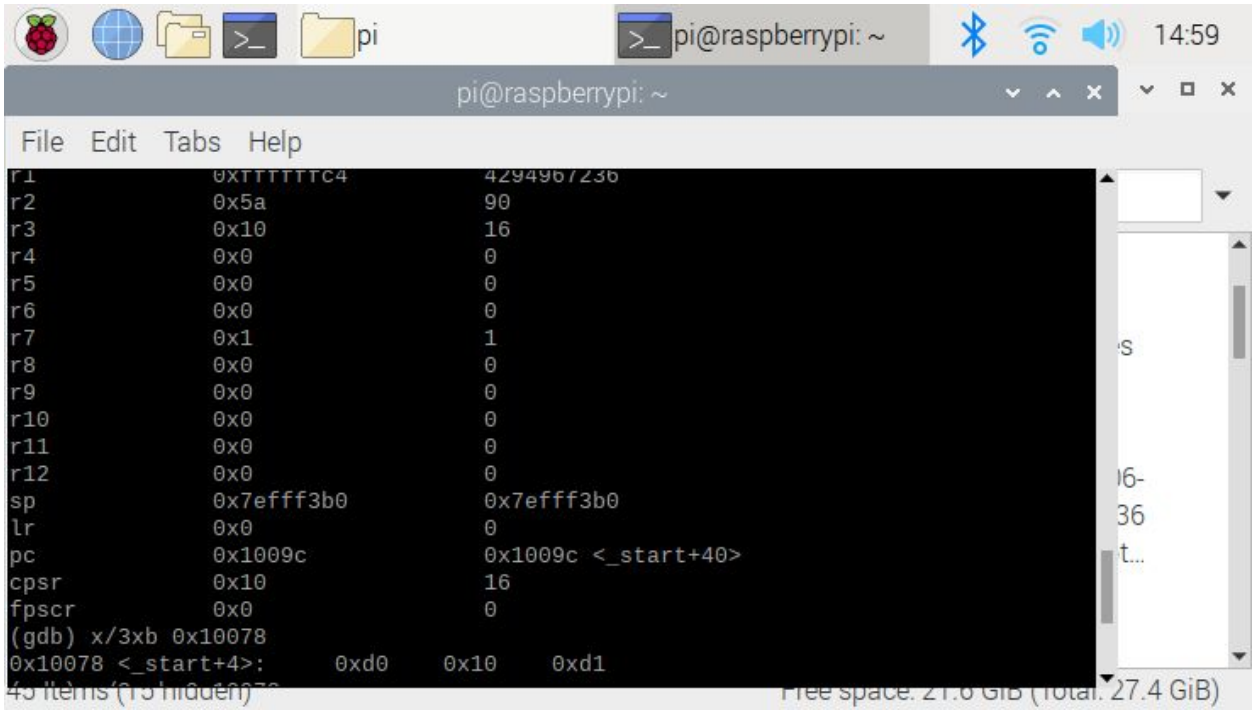
In this screenshot I am assembling and compiling my arithmetic3 program.



This is my code I assigned val1,val2,and val3 as bytes due to the question stating they are 8-bit integer memory values. I then loaded all the values as signed bytes and did arithmetic to get my answer.



In this screenshot I am setting a breakpoint at line 10, running, and then stepping over to get my answer.



In this screenshot I am displaying the registers and memory. In the r2 I get my answer of 90 which is 5a in hex because that is where I store my final result. In r1 is the -60 which is represented by ffffffc4 and 16 is represented by 10 which is the hex for the numbers. In my memory I get 0xd0, 0x10, and 0xd1. For my flags I look at the CPSR and I get 0x10 or 16.

Parallel Programming Skills Foundation: Arteen Ghafourikia

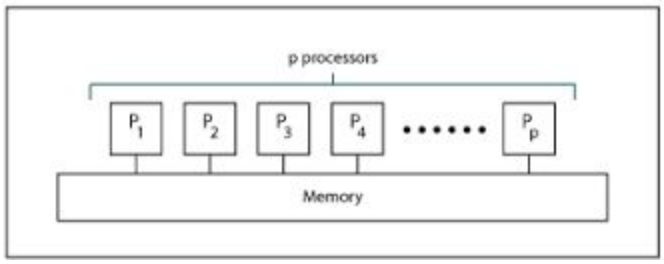
- **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
  - **Task-** It is the set of instructions that are executed by the processor.
  - **Pipelining-** This is when you have a problem and break it down into smaller parts so the processor units can compute it orderly.
  - **Shared Memory-** In hardware, processors have access to a common physical memory. In programming, it is where parallel tasks can access the same memory address regardless if it exists.
  - **Communications-** It is the exchange of data through a shared memory bus or over a network.
  - **Synchronization-** It is when tasks wait for another task to finish or reach a certain point.
- **Classify parallel computers based on Flynn's taxonomy.Briefly describe every one of them.**
  - **Single Instruction, Single Data (SISD)-** Only one instruction executes by the cpu during a clock cycle and only one data stream is used.
  - **Single Instruction, Multiple Data(SIMD)-** All processing units execute the same instruction in any given clock cycle. Each of the different processing units can run on various data elements.
  - **Multiple Instruction, Single Data(MISD)-** Each of the processing units can operate the data independently using different instruction streams. A data stream is placed in many processing units.
  - **Multiple Instruction, Multiple Data(MIMD)-** Every processor may execute a different instruction stream. All the processors may work on different data streams.
- **What are the Parallel Programming Models?**
  - They represent an abstract reference to the hardware and memory architectures.
    - Multiple Program Multiple Data (MPMD)
    - Single Program Multiple Data (SPMD)
    - Hybrid
    - Data Parallel
    - Distributed Memory/Message Passing
    - Threads
    - Shared Memory (without threads)
- **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
  - **Uniform Memory Access (UMA)-** It uses the same processors, and it allows equal access and access times to the memory. If the memory is updated in one location, all the other processors will also know about the update. OpenMP uses

this because you are updating the memory on all the processors after one processor gets access to it.

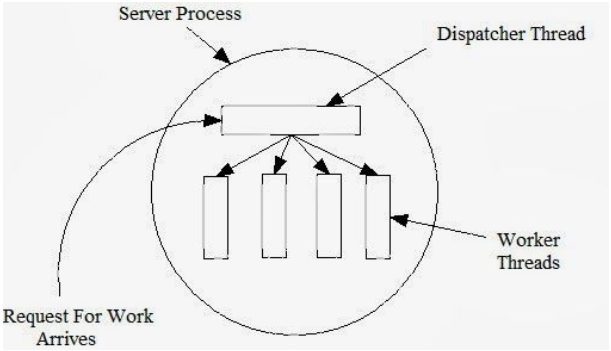
- **Non-Uniform Memory Access(NUMA)-** It is created by physically linking two SMP's. The processors do not have equal access to all the memories, and the entrance in the link is slower.

➢ **Compare Shared Memory Model with Threads Model?**

- **Shared Memory Model-** The processors all share the same memory and are able to read and write the same memory.



- **Threads Model-** Thread models are like mini programs inside of another program which can be more resource draining and they only share the global memory.



➢ **What is Parallel Programming?**

- Parallel programming is being able to carry out many tasks at the same time rather than one by one.

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

- It is a CPU that can integrate all the components of a CPU on a single chip, and it can function. The Raspberry Pi does use an SoC, which gives it the small compact size and lets it have the components of a CPU.

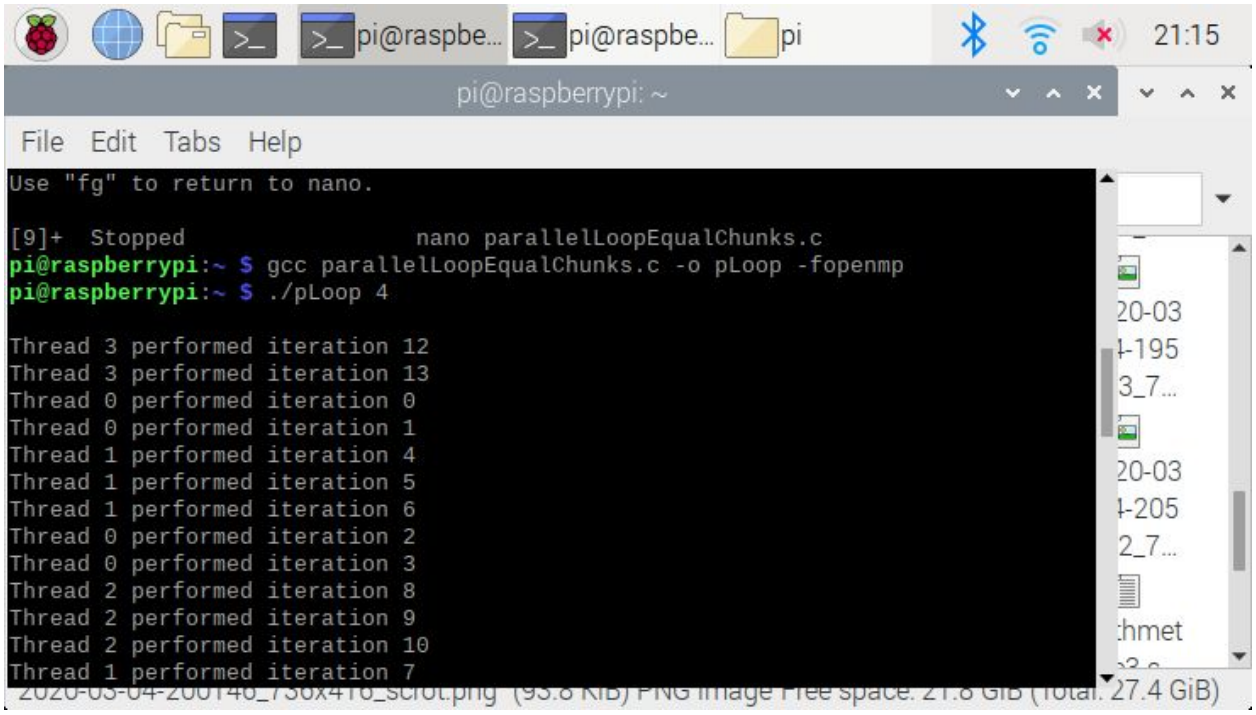
➢ **Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

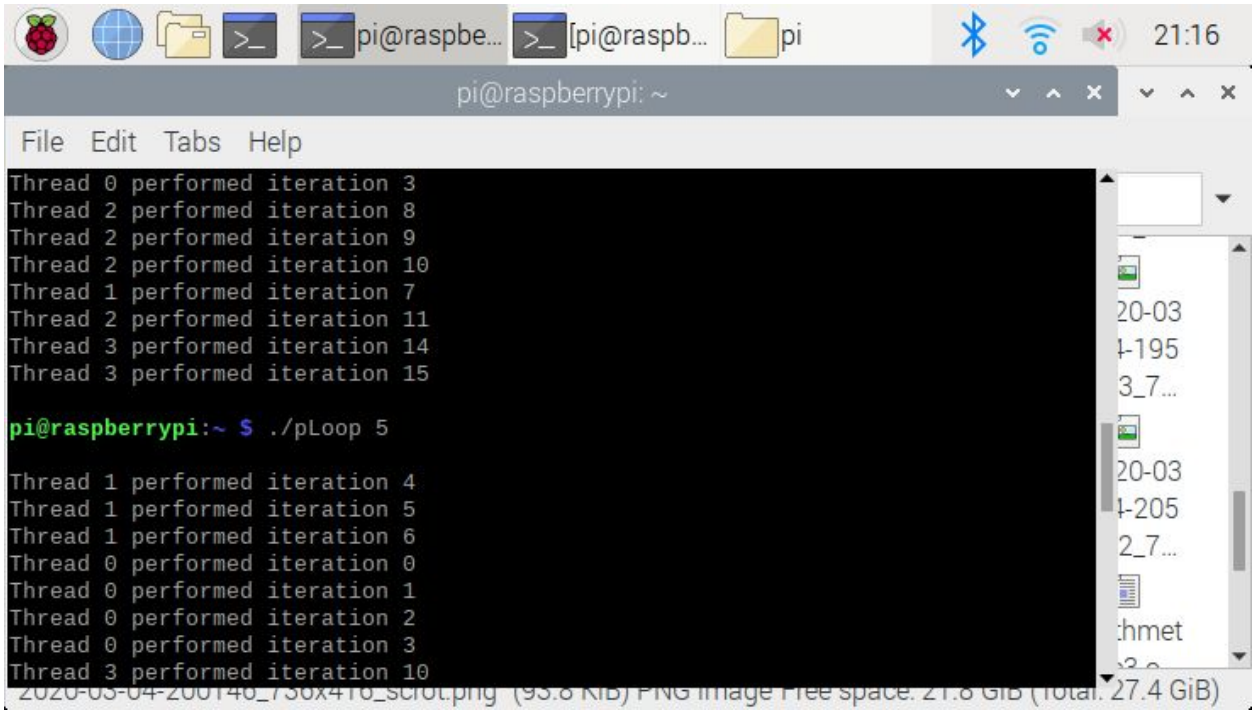
- The SoC is much smaller than CPUs, letting you implement it in things like smartphones in tablets. It also carries a lot more functionality, it uses less power,

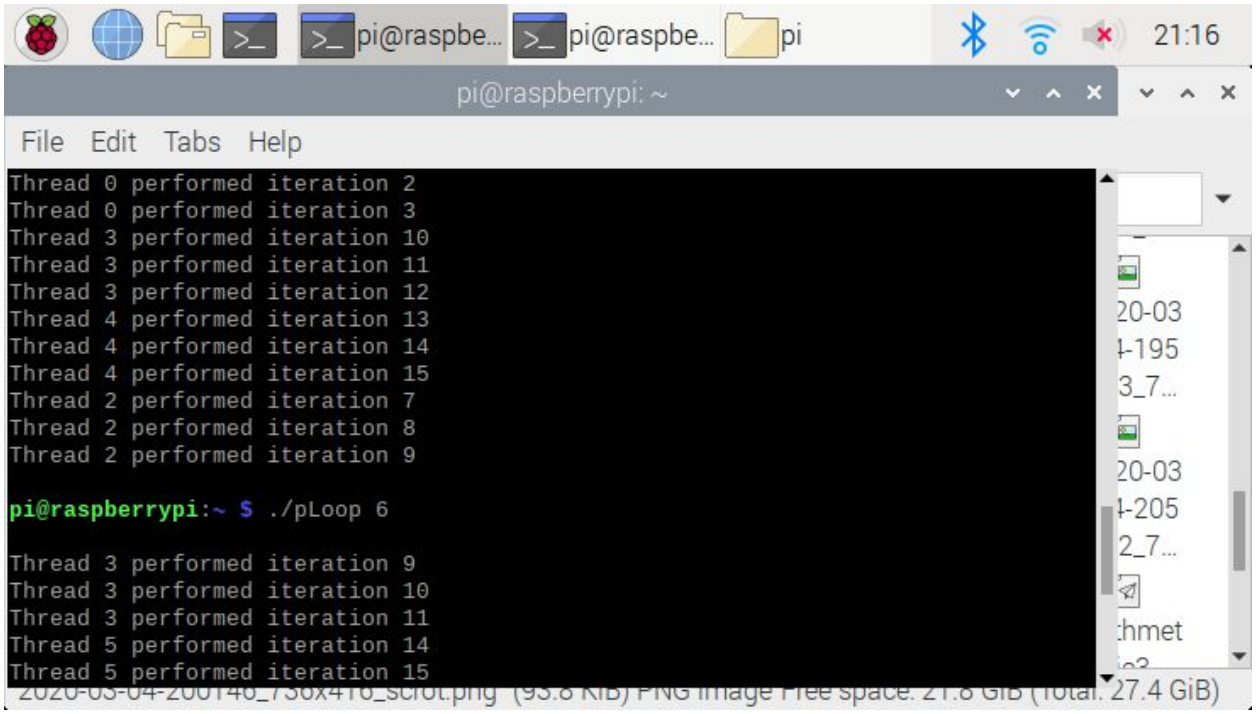
and it is cheaper. The only disadvantage is its flexibility. You are unable to change things such as the CPU, CPU, or RAM.

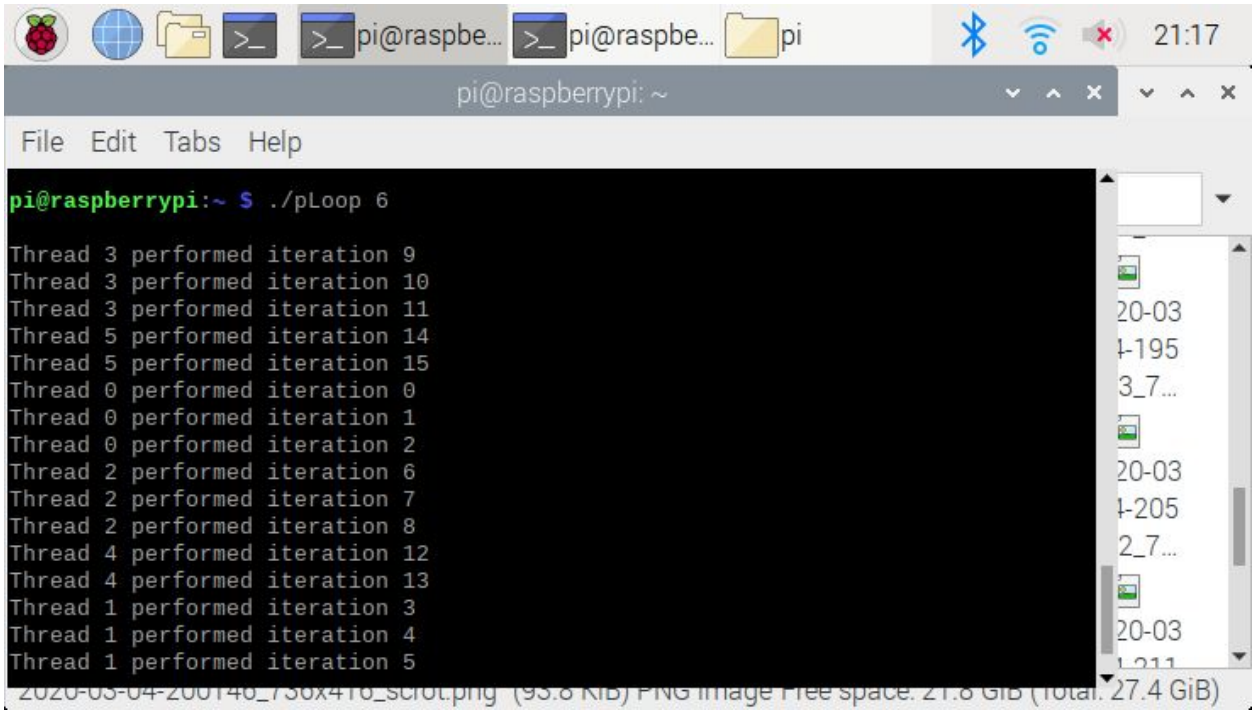
Parallel Programming Basics:Arteen Ghafourikia

Part 1

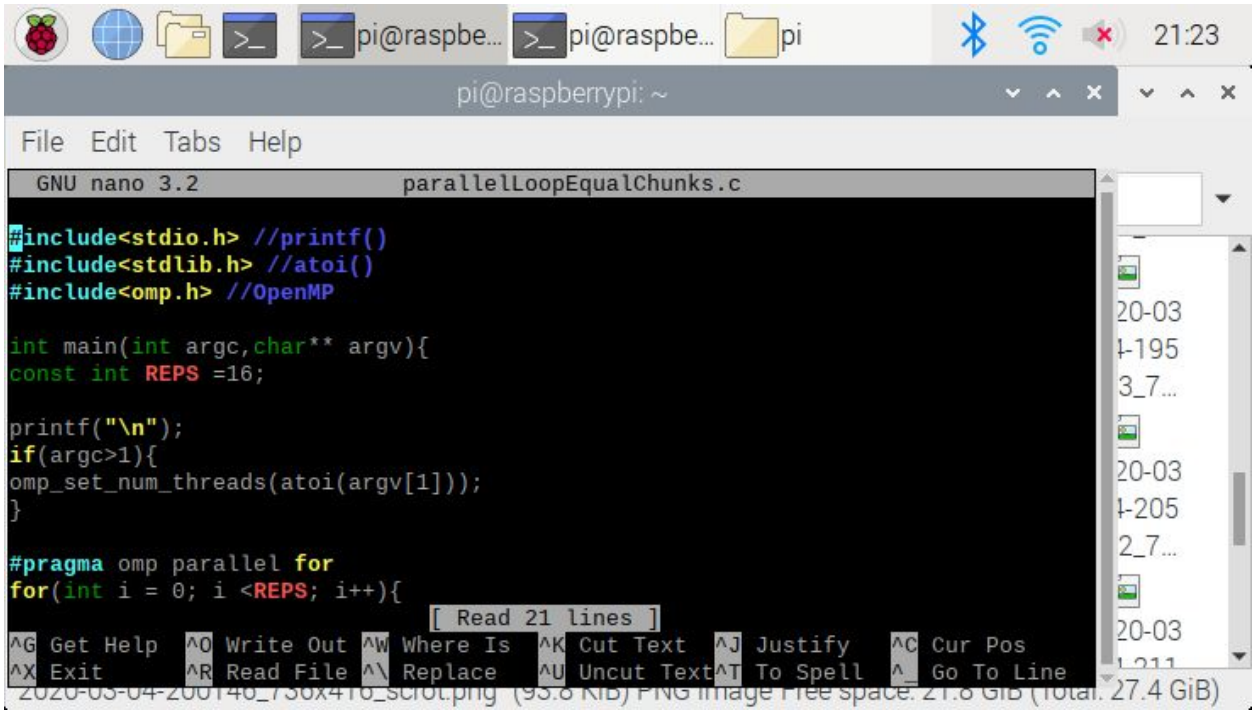


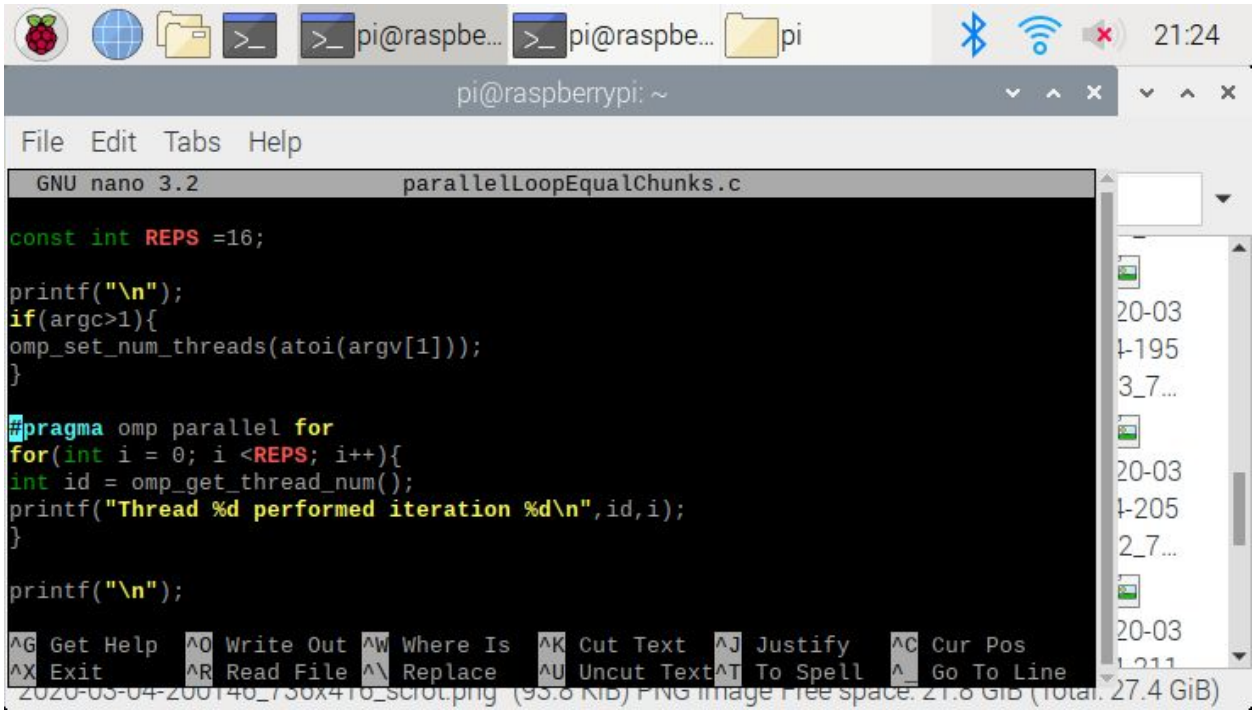

pi@raspbe...
[pi@raspb...
pi
21:16
pi@raspberrypi: ~
File Edit Tabs Help
Thread 0 performed iteration 3
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 1 performed iteration 7
Thread 2 performed iteration 11
Thread 3 performed iteration 14
Thread 3 performed iteration 15
pi@raspberrypi:~ $ ./pLoop 5
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 3 performed iteration 10

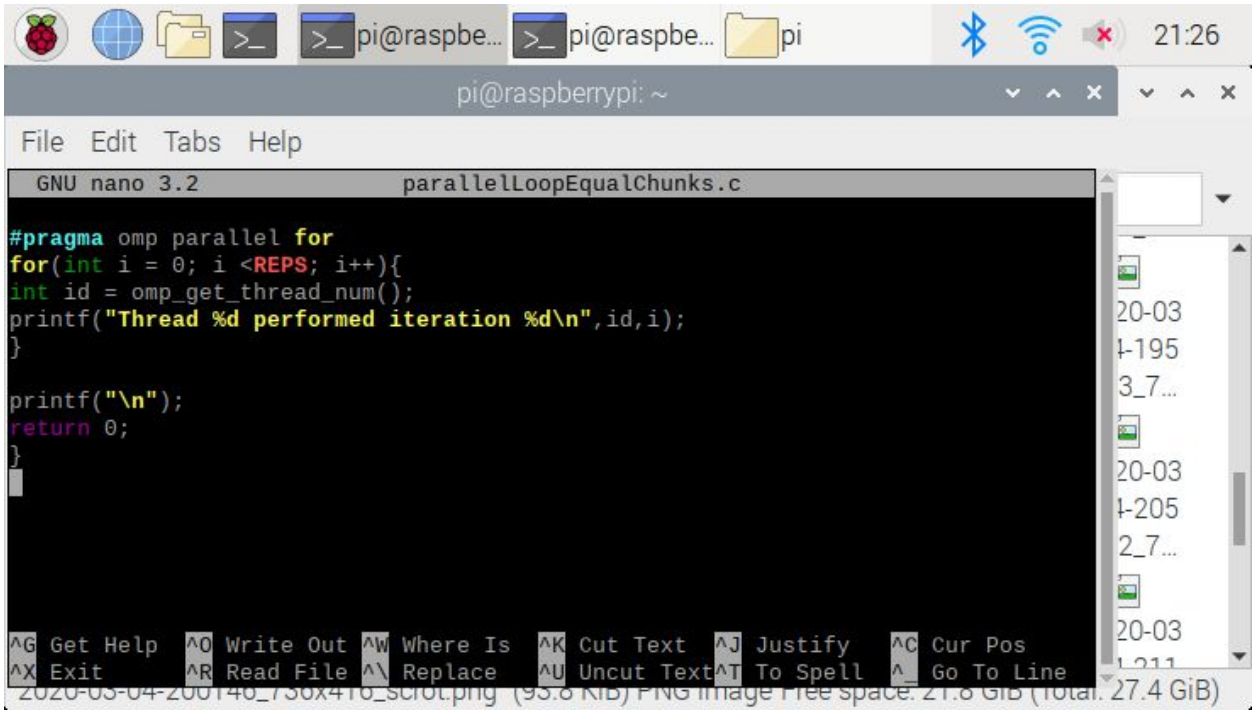

pi@raspbe...
pi@raspbe...
pi
21:16
pi@raspberrypi: ~
File Edit Tabs Help
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 3 performed iteration 10
Thread 3 performed iteration 11
Thread 3 performed iteration 12
Thread 4 performed iteration 13
Thread 4 performed iteration 14
Thread 4 performed iteration 15
Thread 2 performed iteration 7
Thread 2 performed iteration 8
Thread 2 performed iteration 9
pi@raspberrypi:~ $ ./pLoop 6
Thread 3 performed iteration 9
Thread 3 performed iteration 10
Thread 3 performed iteration 11
Thread 5 performed iteration 14
Thread 5 performed iteration 15



In these four screenshots, I am running the example program that is supposed to represent data decomposition patterns. As you can see in the screenshots when I do ./pLoop 4 it is performing the iteration 16 times with 4 different thread ids. Since there is an even distribution, each thread performs four iterations. You can also notice that there is no distinct order to the output of the program. When I perform /pLoop 5, you can see that thread 0 performed 4 iterations while the others only performed 3 iterations. That is because there is not an even distribution. In the program, you also notice that each thread performs iterations consecutively as it is working on chunks of the code at a time. You can also see that the number of repetitions that happens doesn't change since we set the value to 16. When I executed ./pLoop 6 my hypothesis seemed to be true.

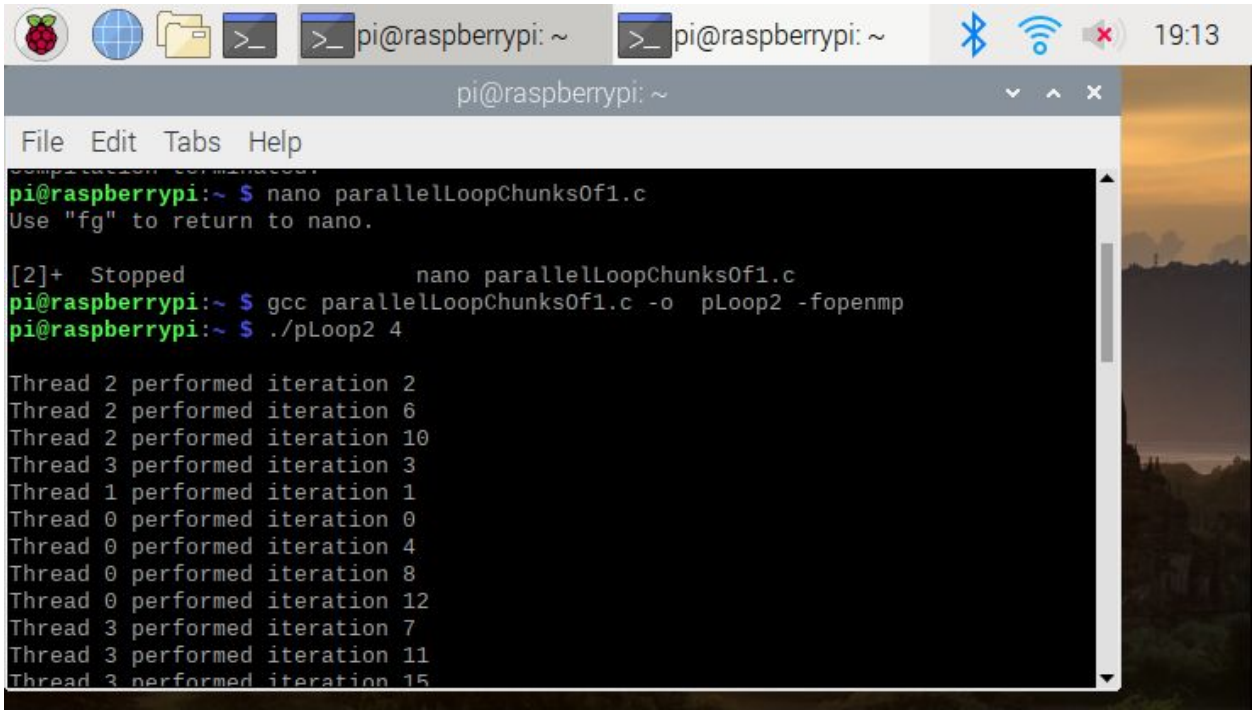

pi@raspbe...
pi@raspbe...
pi
21:23
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopEqualChunks.c
#include<stdio.h> //printf()
#include<stdlib.h> //atoi()
#include<omp.h> //OpenMP
int main(int argc,char** argv){
const int REPS =16;
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel for
for(int i = 0; i <REPS; i++){
[Read 21 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
27.4 GiB)

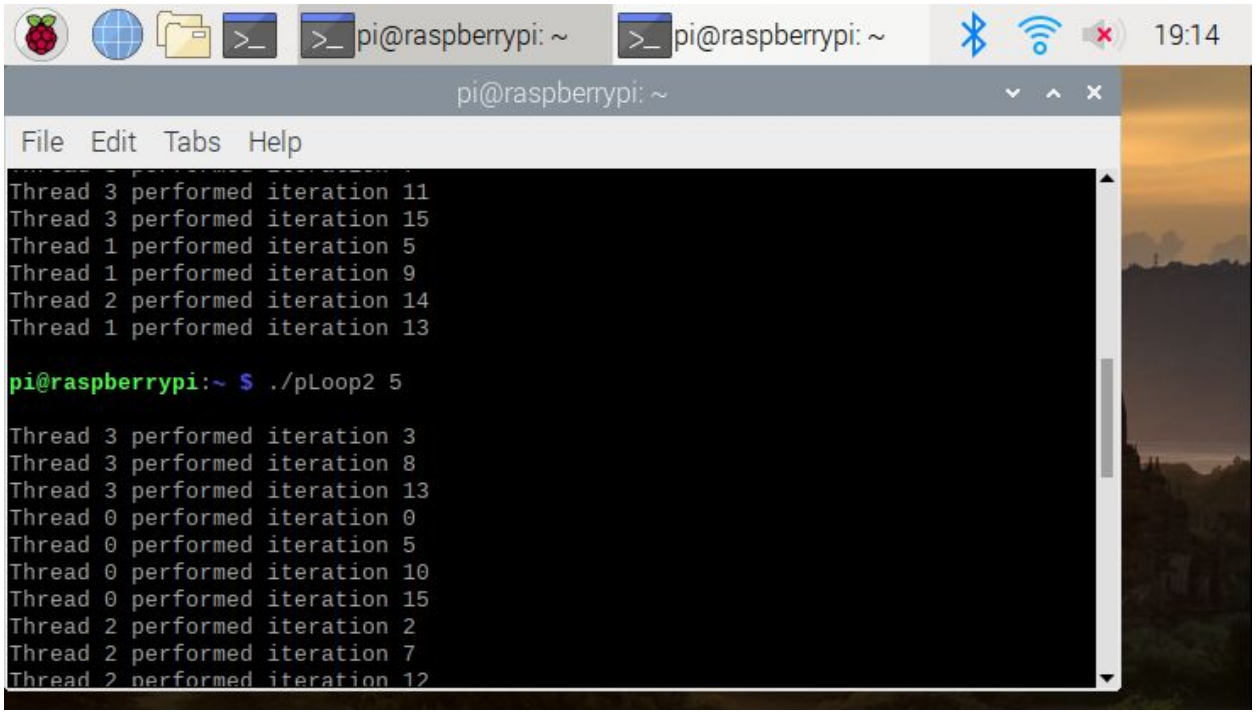

pi@raspbe...
pi@raspbe...
pi
21:24
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopEqualChunks.c
const int REPS =16;
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel for
for(int i = 0; i <REPS; i++){
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n",id,i);
}
printf("\n");
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
27.4 GiB)

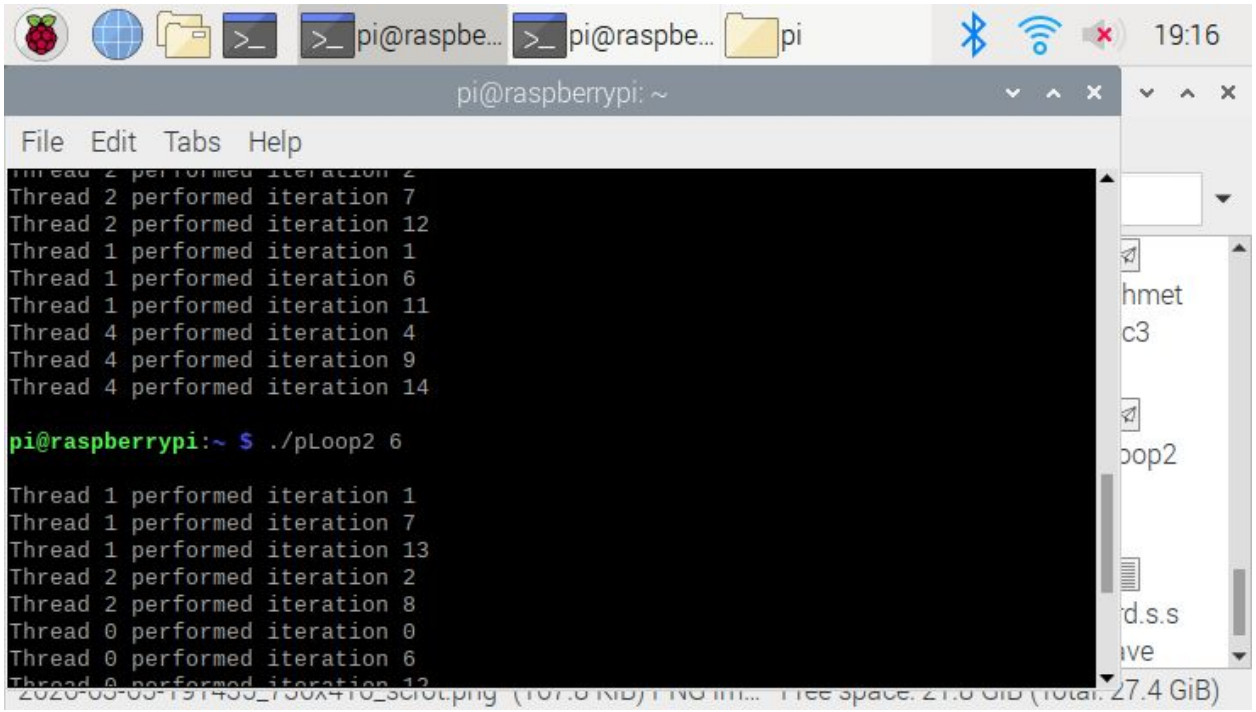


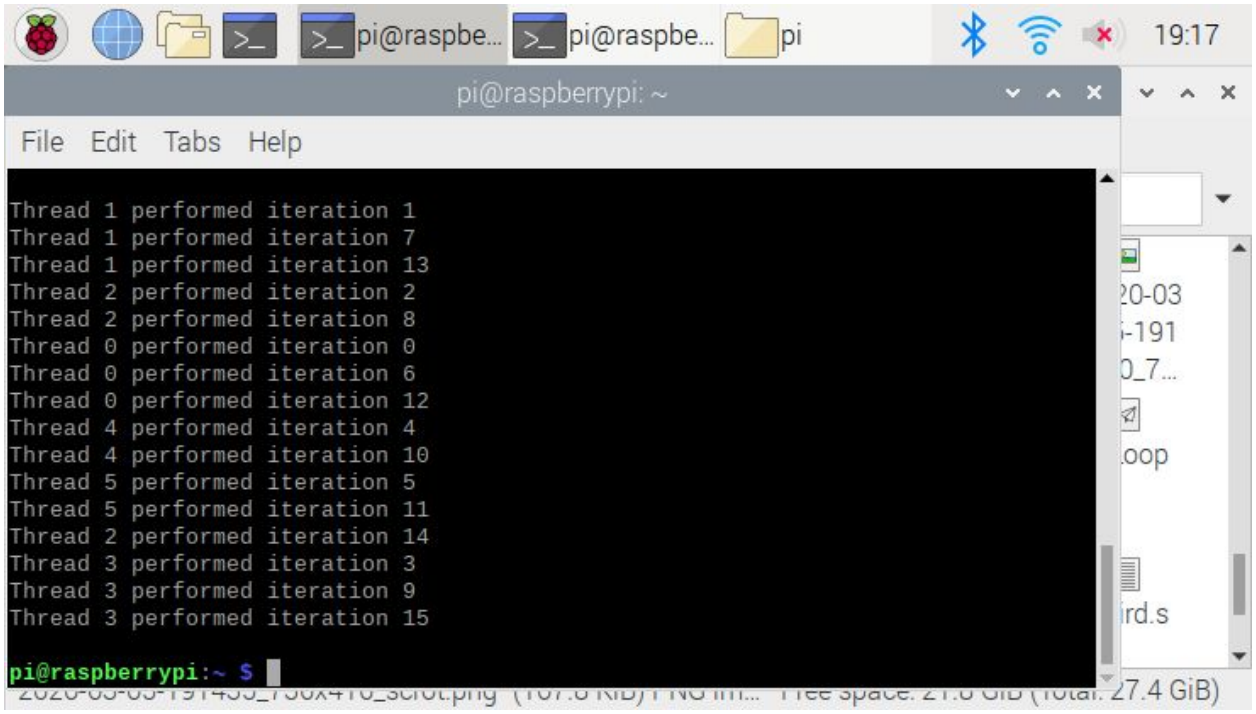
The three screenshots from above are the code for part 1 of Parallel Programming, where we are decomposing the amount of work done.

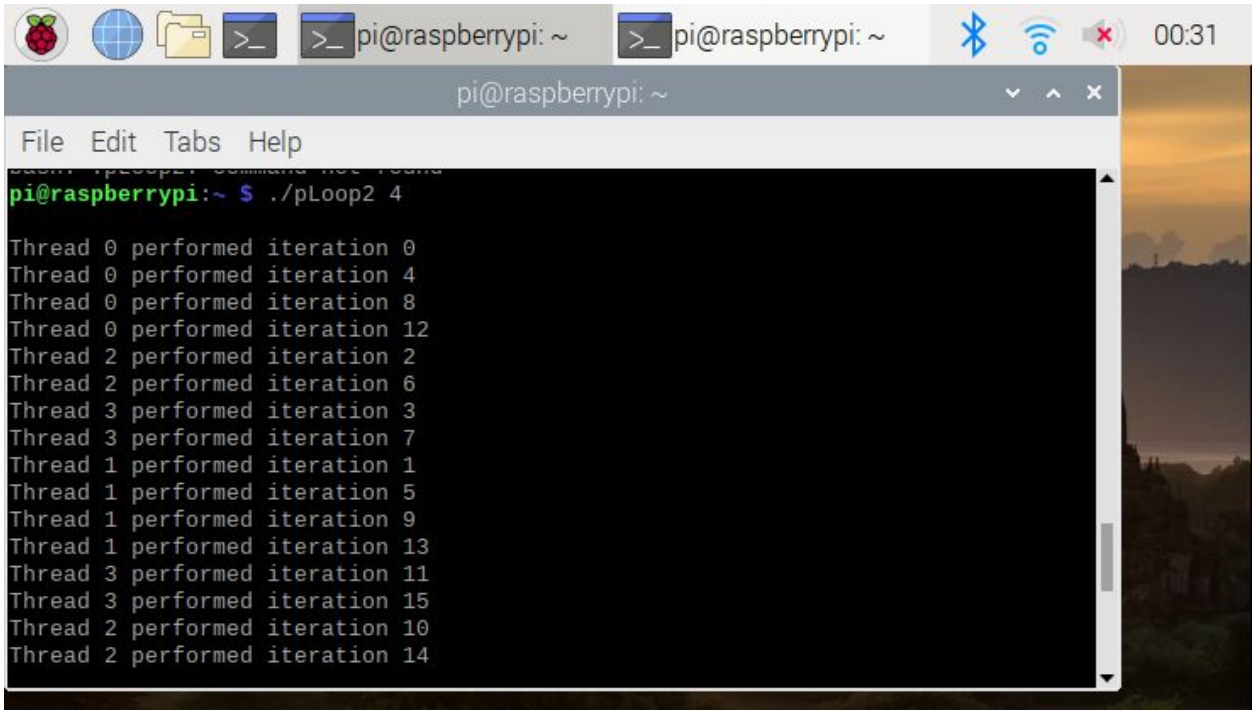
Part 2

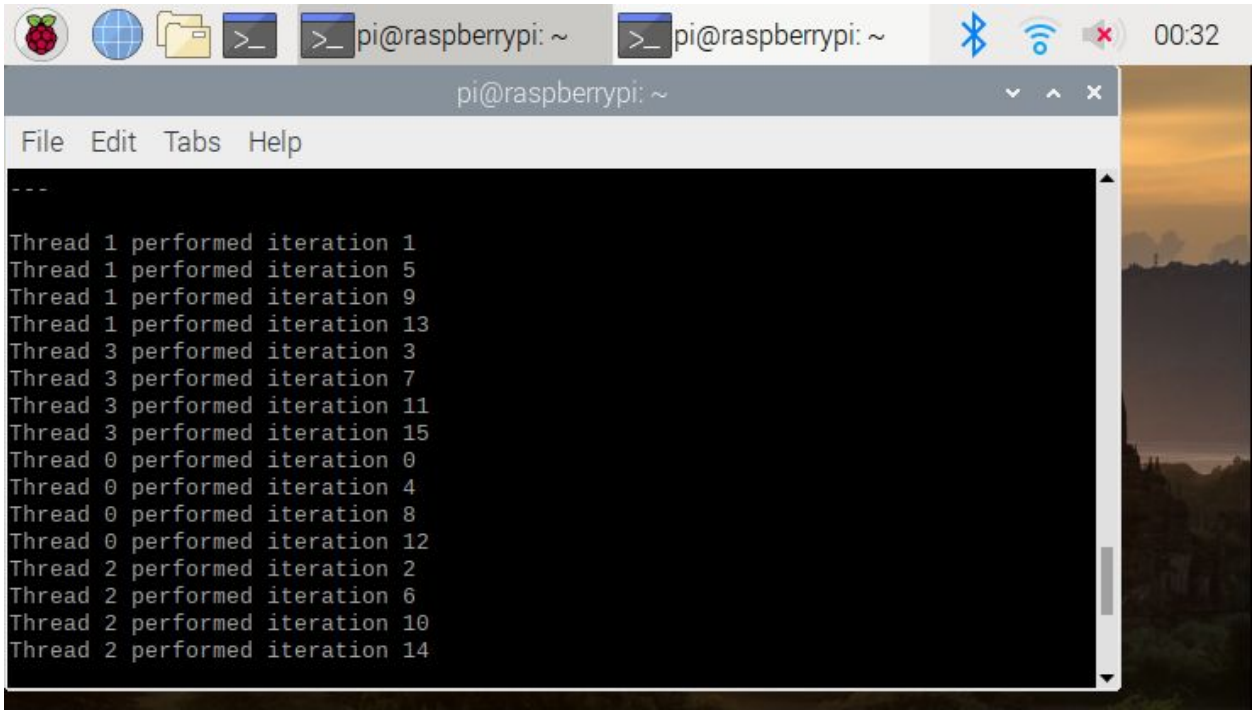



pi@raspberrypi: ~
pi@raspberrypi: ~
19:14
pi@raspberrypi: ~
File Edit Tabs Help
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 2 performed iteration 14
Thread 1 performed iteration 13
pi@raspberrypi:~ $ ./pLoop2 5
Thread 3 performed iteration 3
Thread 3 performed iteration 8
Thread 3 performed iteration 13
Thread 0 performed iteration 0
Thread 0 performed iteration 5
Thread 0 performed iteration 10
Thread 0 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 7

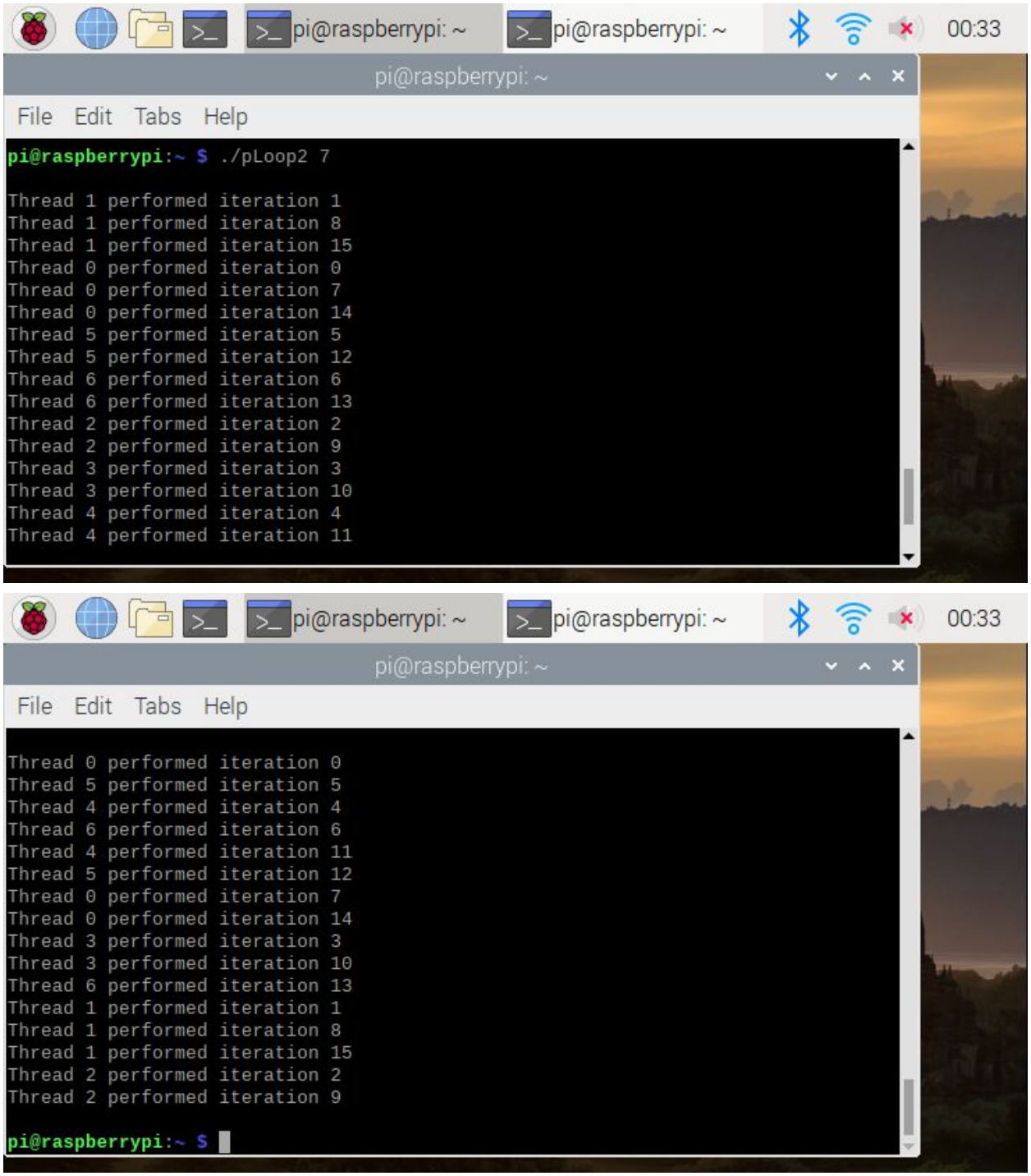

pi@raspbe...
pi@raspbe...
pi
19:16
pi@raspberrypi: ~
File Edit Tabs Help
Thread 2 performed iteration 7
Thread 2 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 6
Thread 1 performed iteration 11
Thread 4 performed iteration 4
Thread 4 performed iteration 9
Thread 4 performed iteration 14
pi@raspberrypi:~ $ ./pLoop2 6
Thread 1 performed iteration 1
Thread 1 performed iteration 7
Thread 1 performed iteration 13
Thread 2 performed iteration 2
Thread 2 performed iteration 8
Thread 0 performed iteration 0
Thread 0 performed iteration 6
hmet
c3
oop2
d.s.s
ve
27.4 GiB)


pi@raspbe...
pi@raspbe...
pi
19:17
pi@raspberrypi: ~
File Edit Tabs Help
Thread 1 performed iteration 1
Thread 1 performed iteration 7
Thread 1 performed iteration 13
Thread 2 performed iteration 2
Thread 2 performed iteration 8
Thread 0 performed iteration 0
Thread 0 performed iteration 6
Thread 0 performed iteration 12
Thread 4 performed iteration 4
Thread 4 performed iteration 10
Thread 5 performed iteration 5
Thread 5 performed iteration 11
Thread 2 performed iteration 14
Thread 3 performed iteration 3
Thread 3 performed iteration 9
Thread 3 performed iteration 15
pi@raspberrypi:~ $

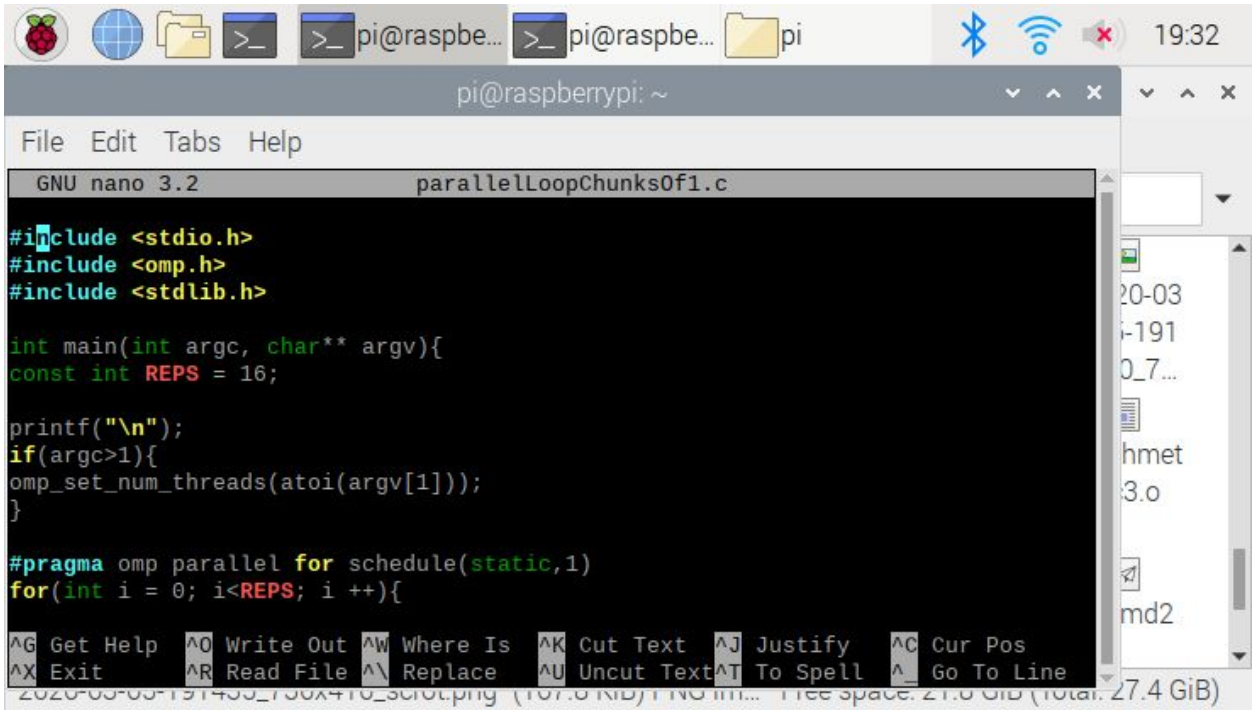

pi@raspberrypi: ~
pi@raspberrypi: ~
00:31
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ ./pLoop2 4
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 2 performed iteration 10
Thread 2 performed iteration 14

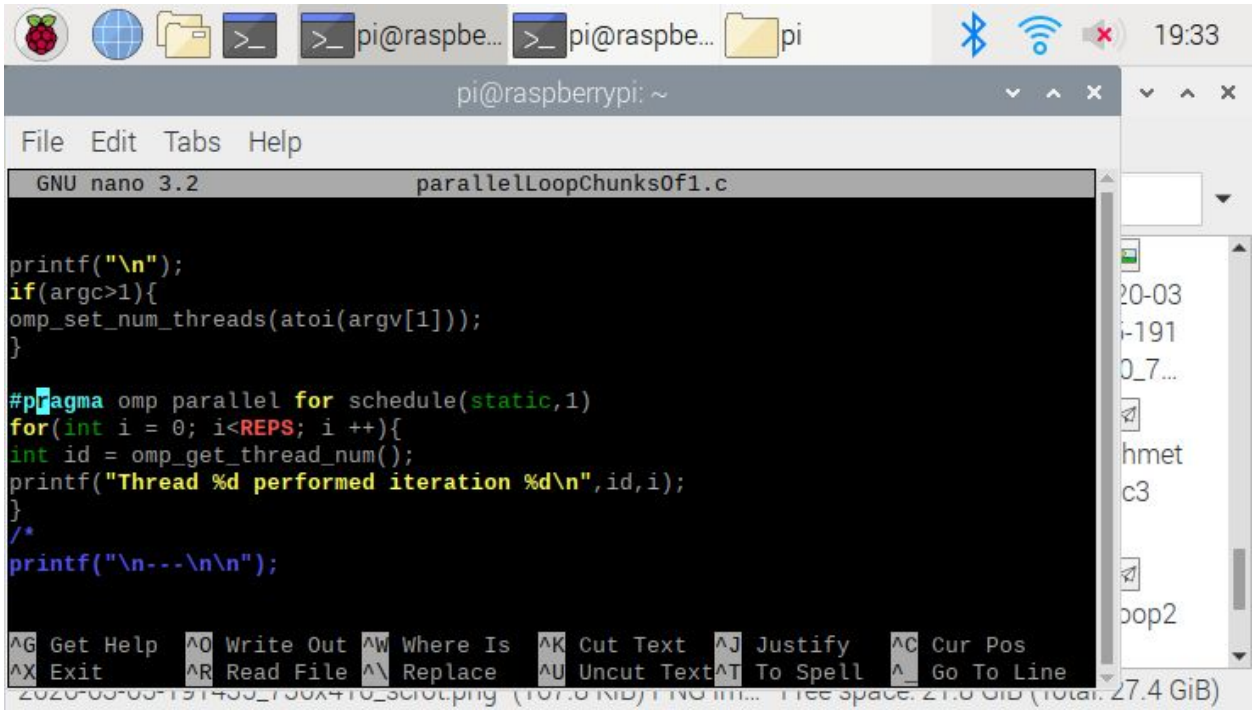

pi@raspberrypi: ~
pi@raspberrypi: ~
00:32
pi@raspberrypi: ~
File Edit Tabs Help

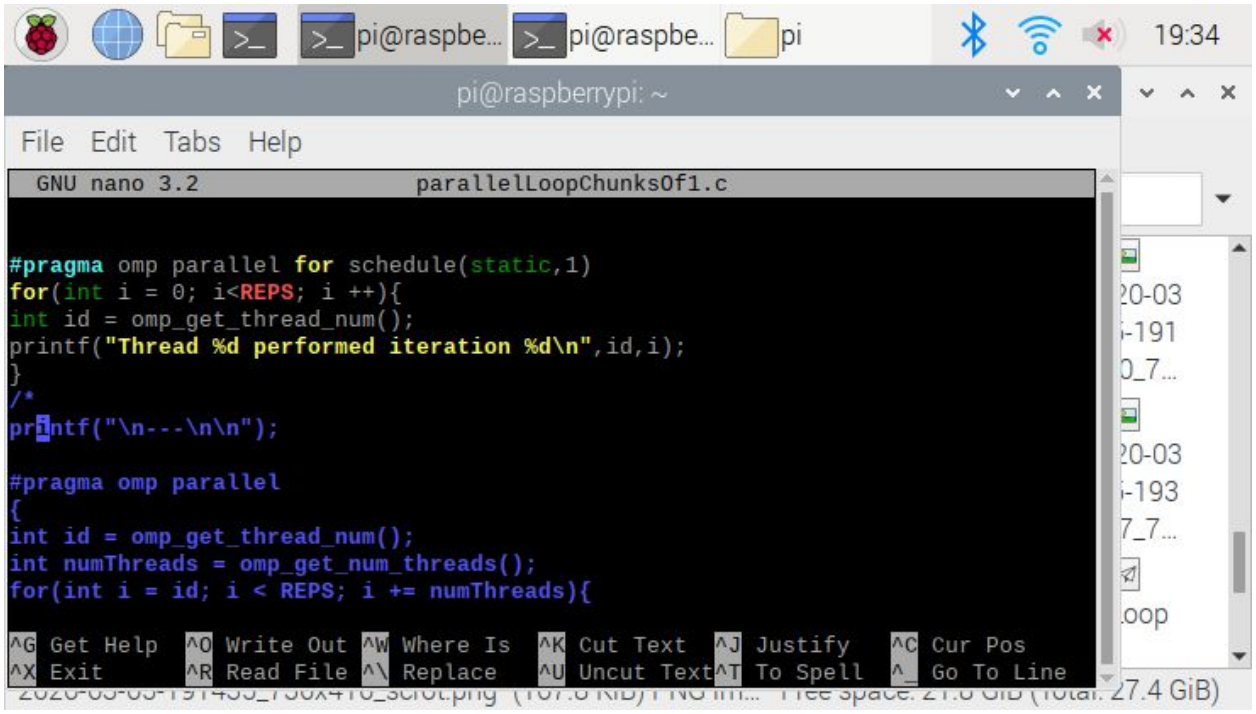
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 2 performed iteration 14

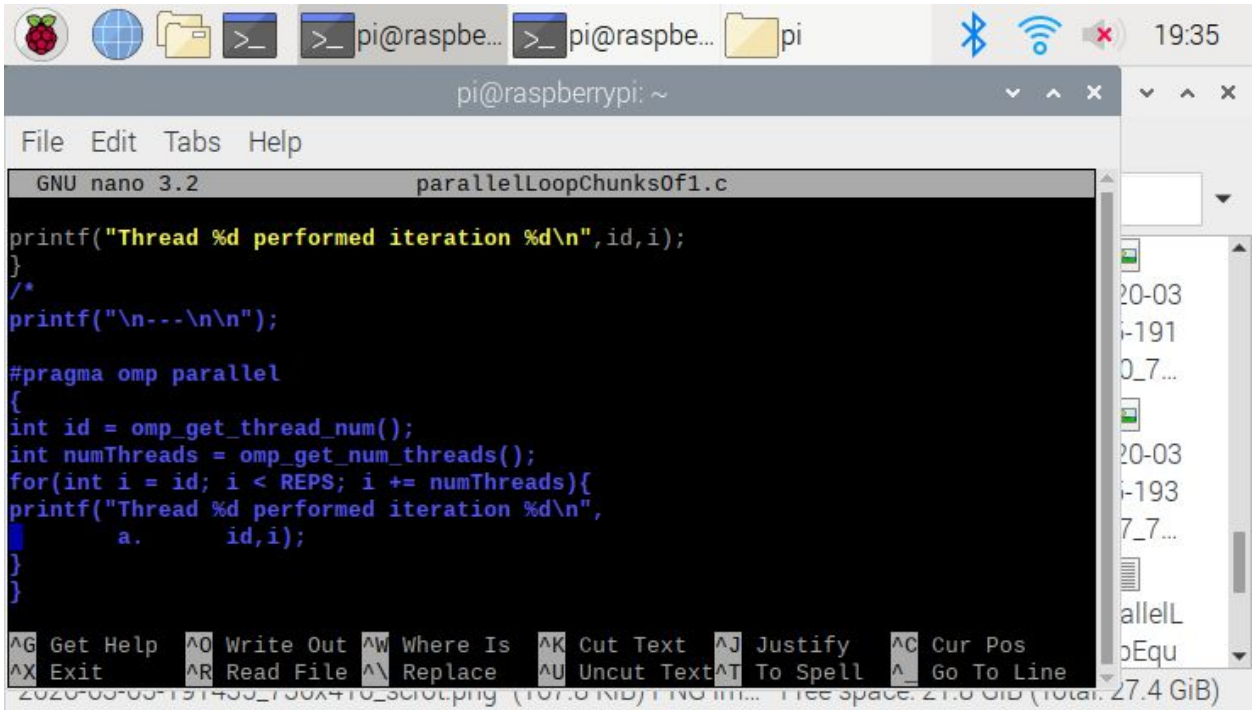


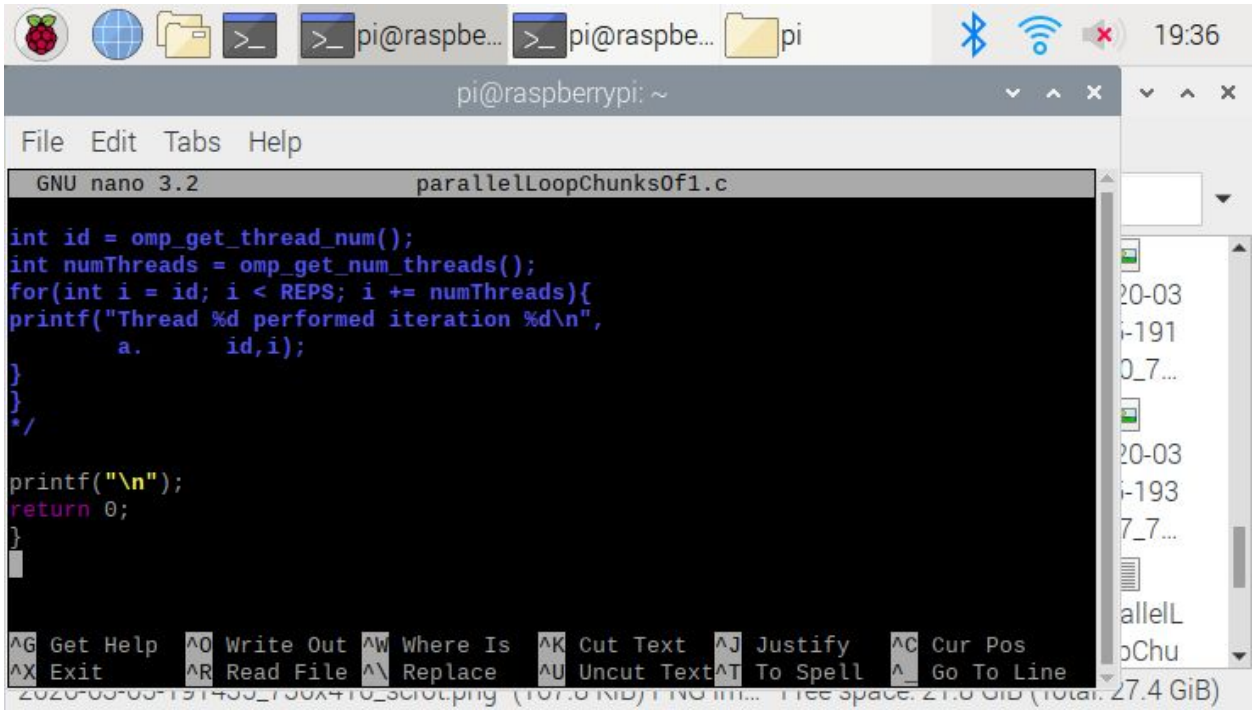
In part 2, we have another example of data decomposition where each thread does an equal amount of work, but the iterations are not consecutive. In this code, we have an commented version where each thread is performing an iteration; however, the threads are not performing repetitions consecutively. In the uncommented version of the code, you see the same pattern occurring.


pi@raspbe...
pi@raspbe...
pi
19:32
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopChunksOf1.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv){
const int REPS = 16;
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel for schedule(static,1)
for(int i = 0; i<REPS; i ++){
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspbe...
pi@raspbe...
pi
19:33
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopChunksOf1.c
printf("\n");
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel for schedule(static,1)
for(int i = 0; i<REPS; i ++){
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n",id,i);
}
/*
printf("\n---\n\n");
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

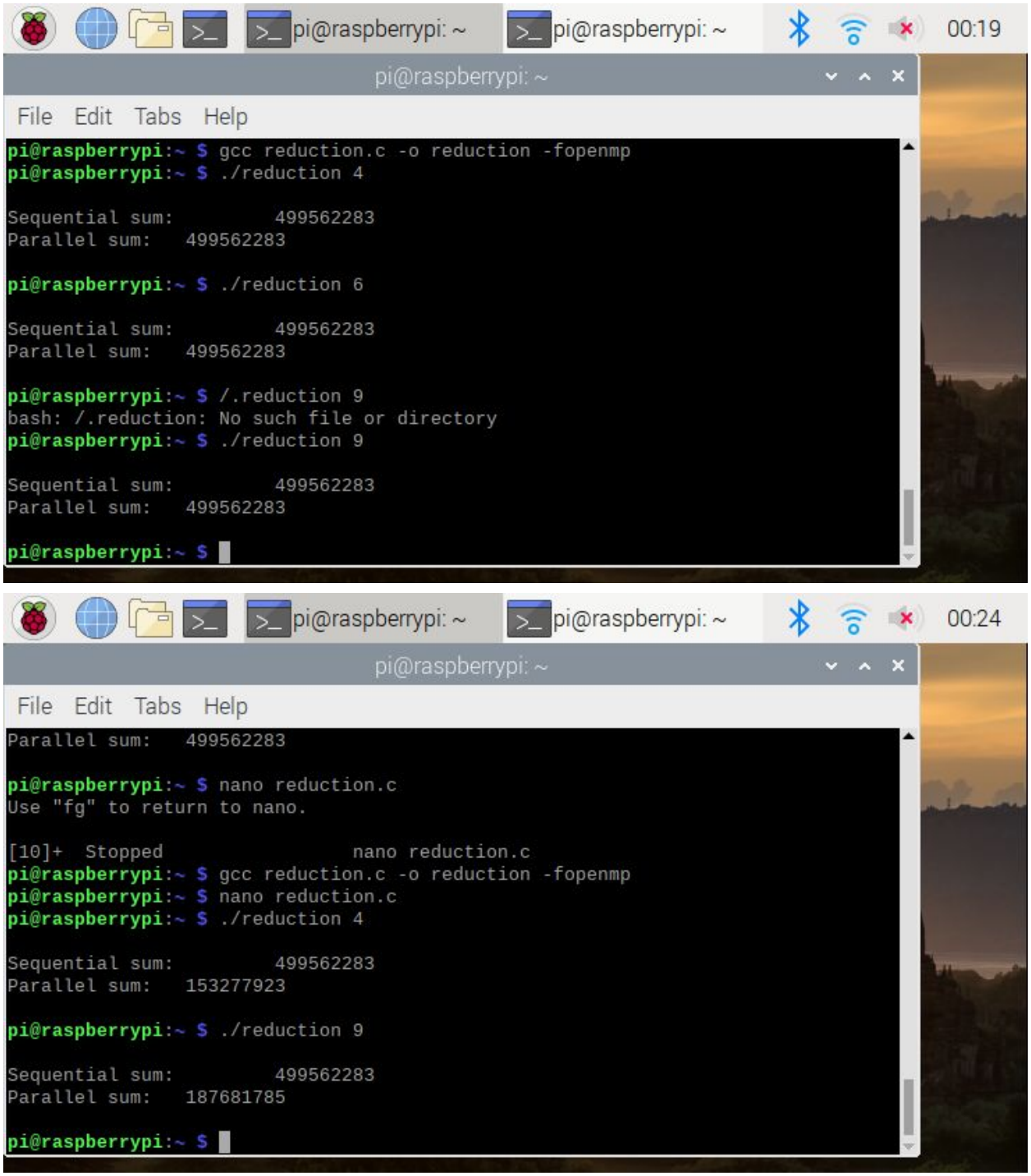

pi@raspbe...
pi@raspbe...
pi
19:34
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopChunksOf1.c
#pragma omp parallel for schedule(static,1)
for(int i = 0; i<REPS; i ++){
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n",id,i);
}
/*
printf("\n---\n\n");
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
for(int i = id; i < REPS; i += numThreads){
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
27.4 GiB)



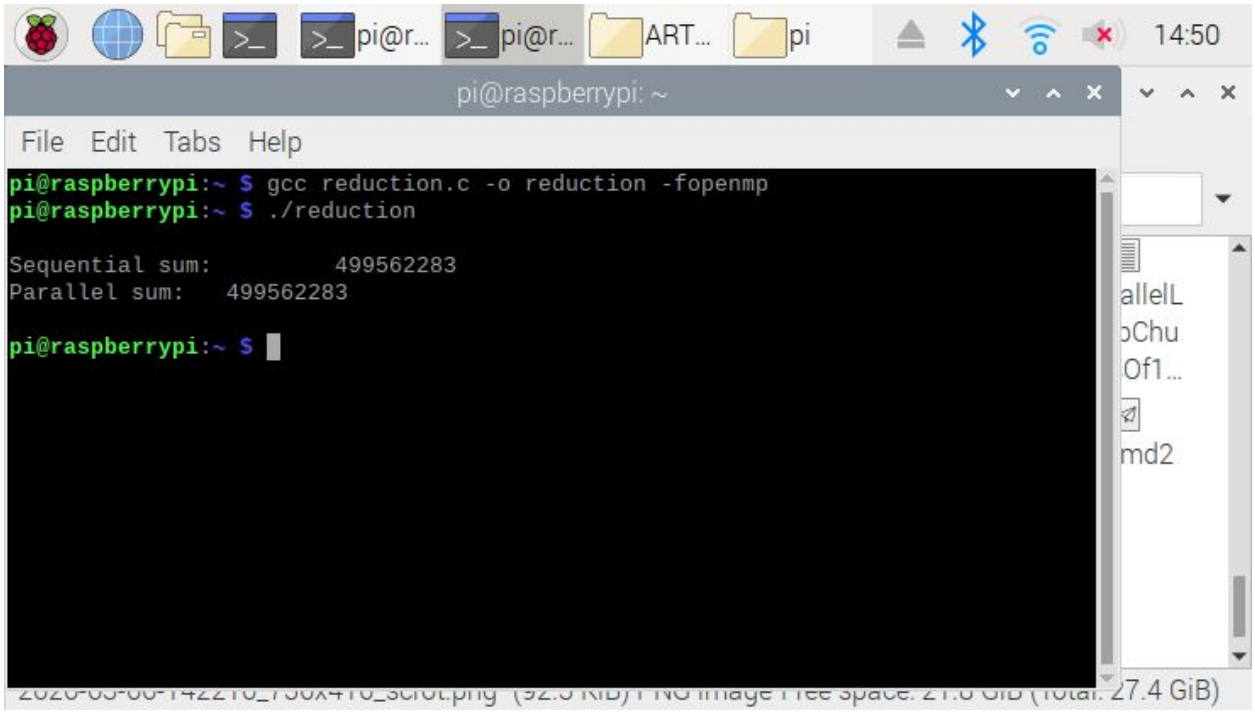


The five screenshots above are the code for part 2 of parallel programming.

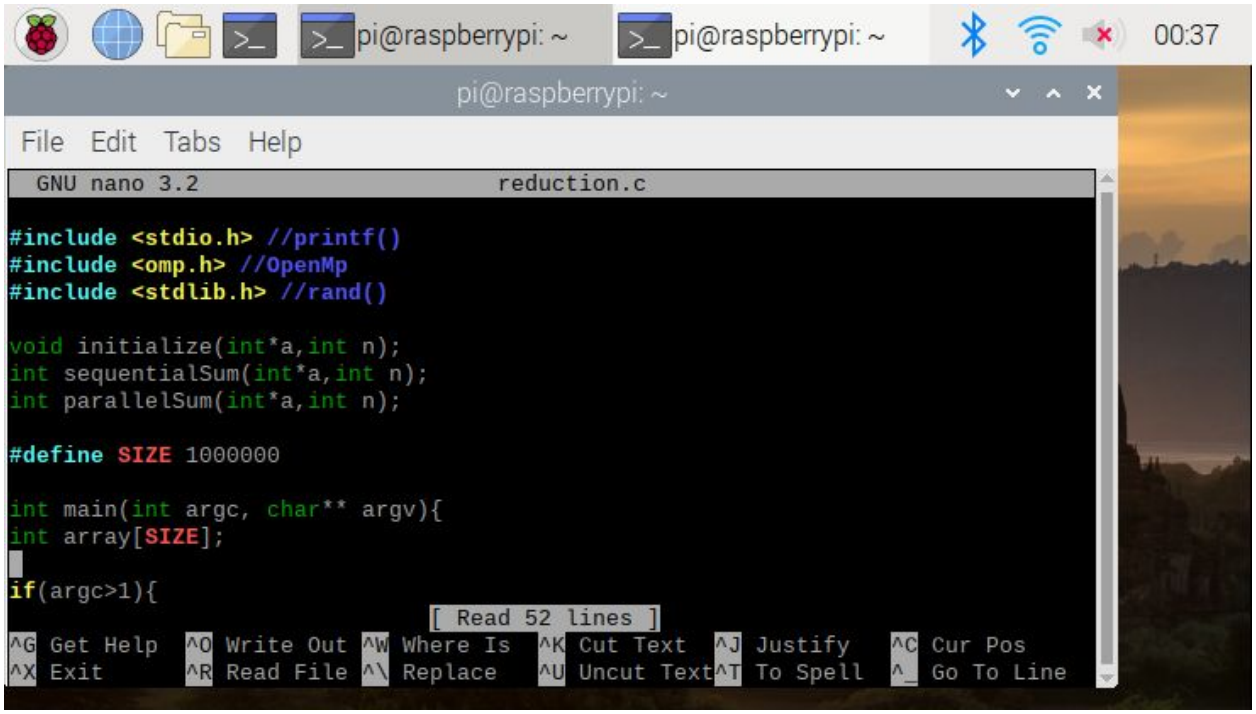
Part 3

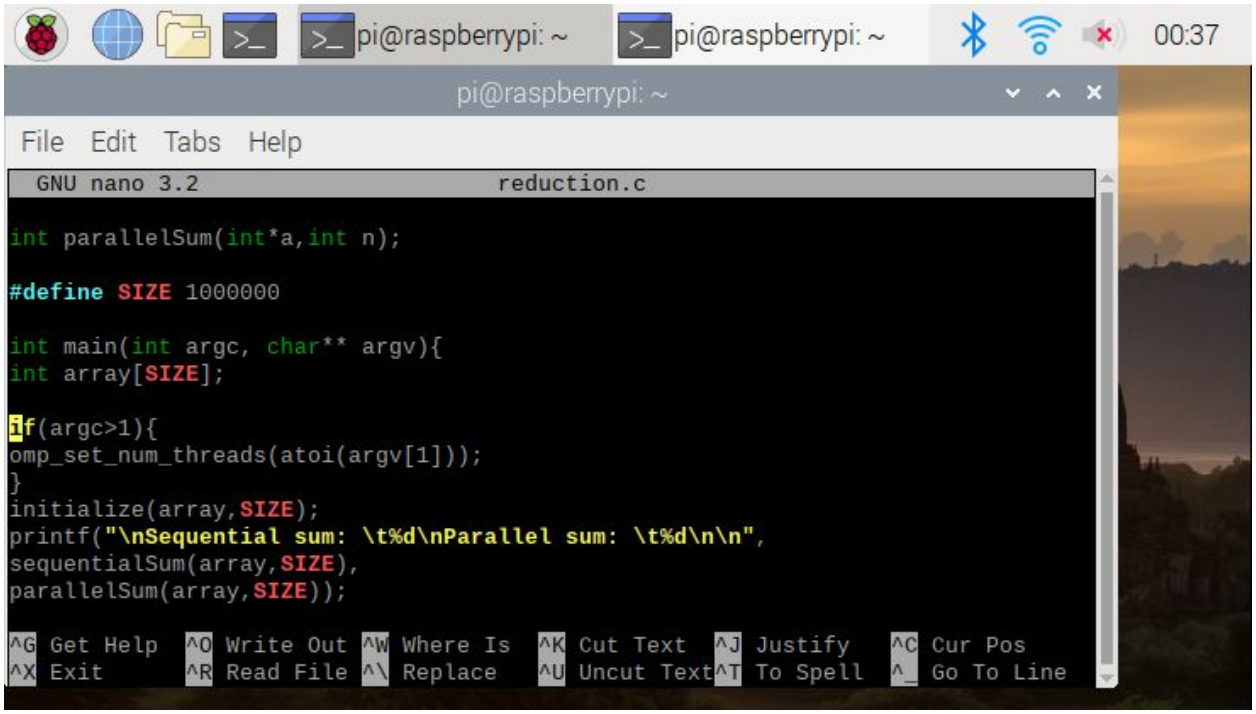


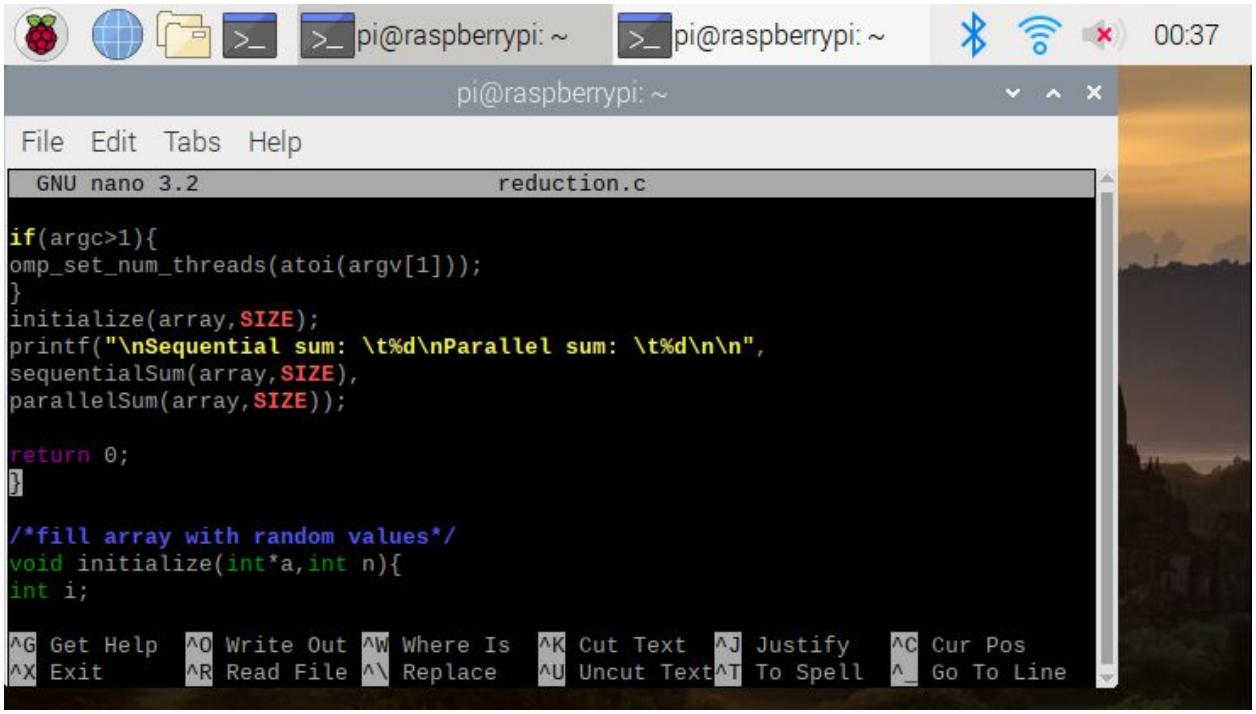
In the commented version of the code, you can see that both the Sequential sum and the Parallel sum has the same value. However, when I uncommented the first set, you see that the Parallel amount has a different and smaller value than the Sequential sum. That means something isn't working since it is not displaying the correct value.

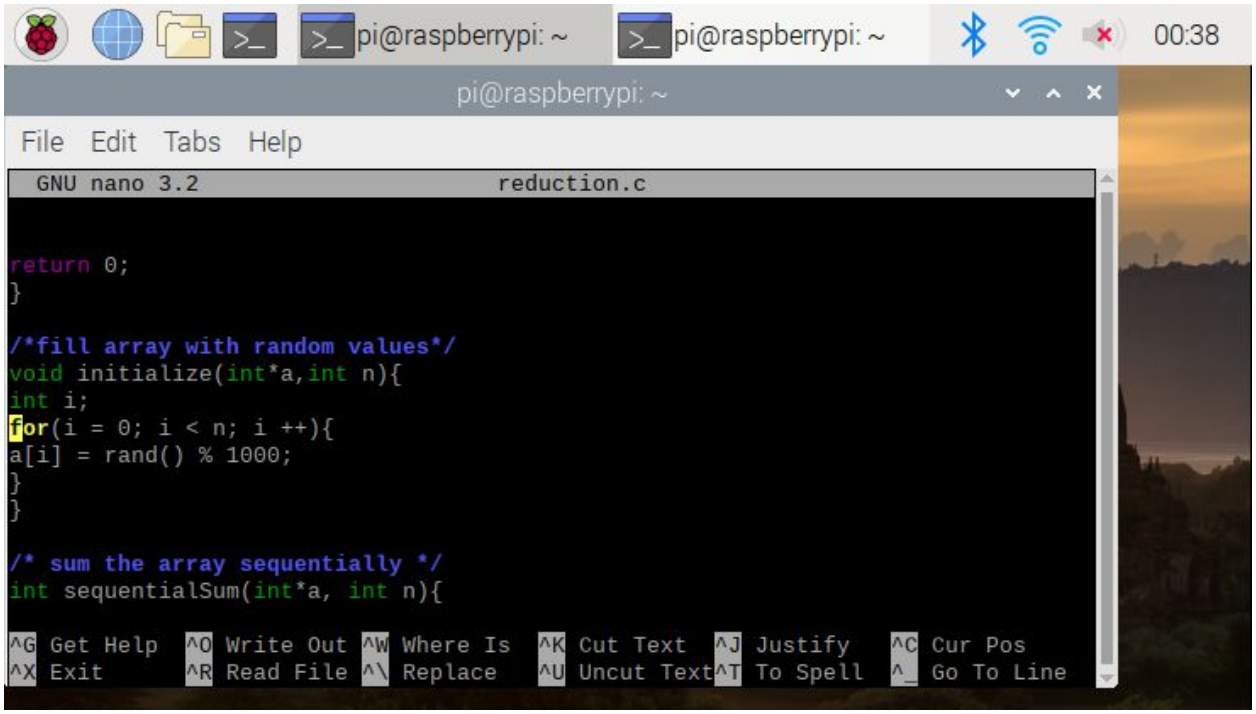


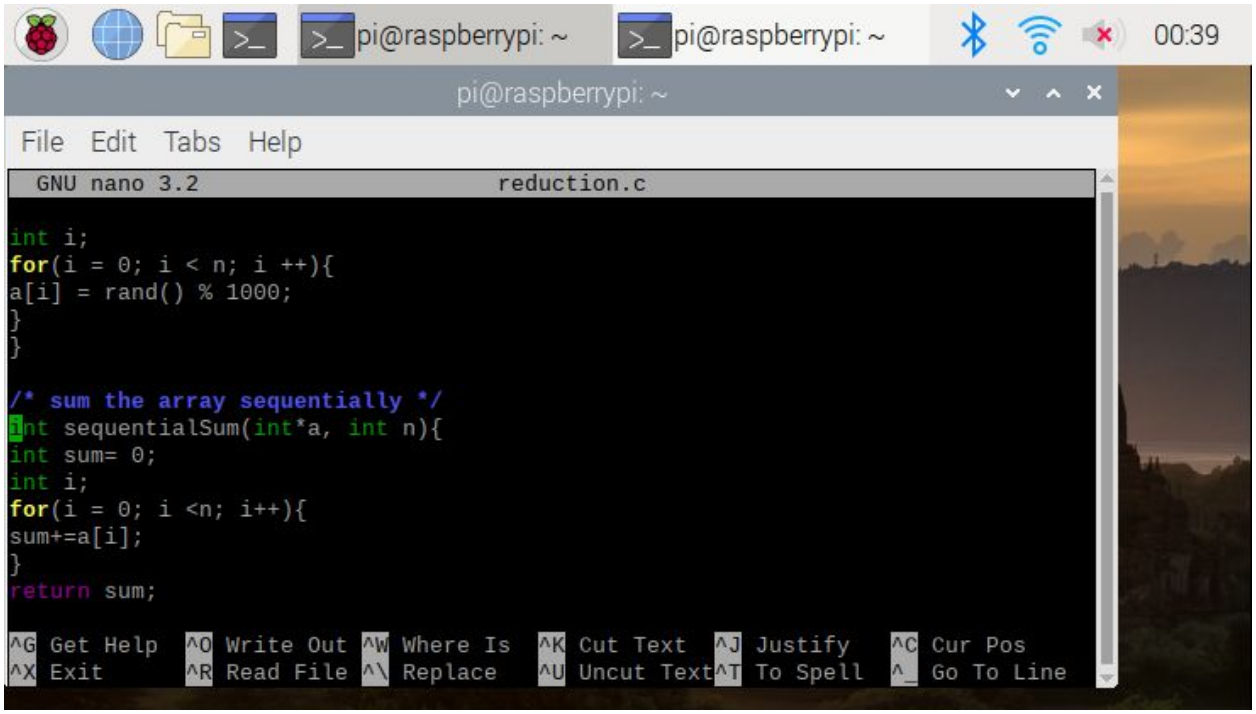
In this screenshot I removed the second set of comments and noticed that the sums are equal again. This tells me that the “reduction (+:sum)” is necessary to have the program run correctly. The reason why we need that statement is because we want to encapsulate it.

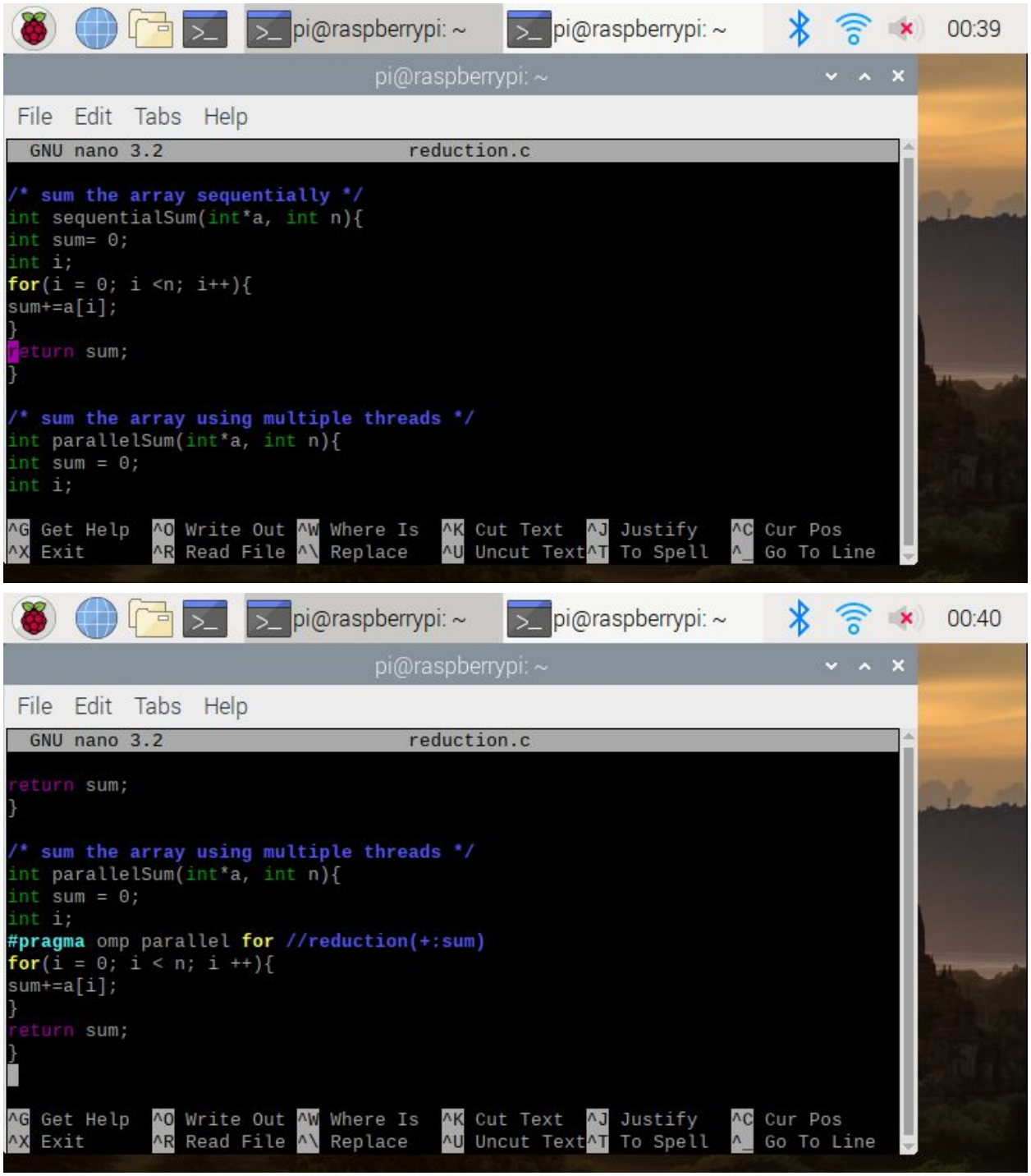

pi@raspberrypi: ~
pi@raspberrypi: ~
00:37
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 reduction.c
#include <stdio.h> //printf()
#include <omp.h> //OpenMp
#include <stdlib.h> //rand()
void initialize(int*a,int n);
int sequentialSum(int*a,int n);
int parallelSum(int*a,int n);
#define SIZE 1000000
int main(int argc, char** argv){
int array[SIZE];
if(argc>1){
[Read 52 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
00:37
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 reduction.c
int parallelSum(int*a,int n);
#define SIZE 1000000
int main(int argc, char** argv){
int array[SIZE];
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
initialize(array,SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array,SIZE),
parallelSum(array,SIZE));
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
00:37
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 reduction.c
if(argc>1){
omp_set_num_threads(atoi(argv[1]));
}
initialize(array,SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array,SIZE),
parallelSum(array,SIZE));
return 0;
}
/*fill array with random values*/
void initialize(int*a,int n){
int i;
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line


pi@raspberrypi: ~
pi@raspberrypi: ~
00:38
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 reduction.c
return 0;
}
/*fill array with random values*/
void initialize(int*a,int n){
int i;
for(i = 0; i < n; i ++){
a[i] = rand() % 1000;
}
}
/* sum the array sequentially */
int sequentialSum(int*a, int n){
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

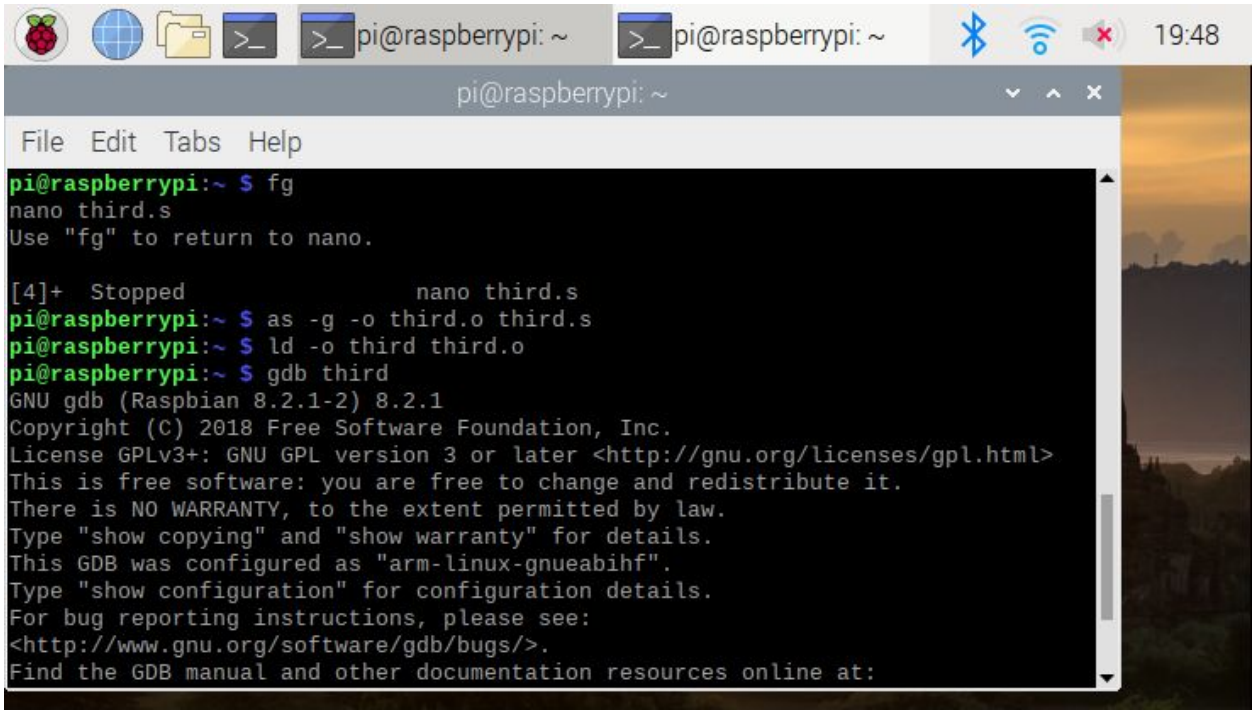

pi@raspberrypi: ~
pi@raspberrypi: ~
00:39
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 reduction.c
int i;
for(i = 0; i < n; i ++){
a[i] = rand() % 1000;
}
}
/* sum the array sequentially */
int sequentialSum(int*a, int n){
int sum= 0;
int i;
for(i = 0; i <n; i++){
sum+=a[i];
}
return sum;
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line



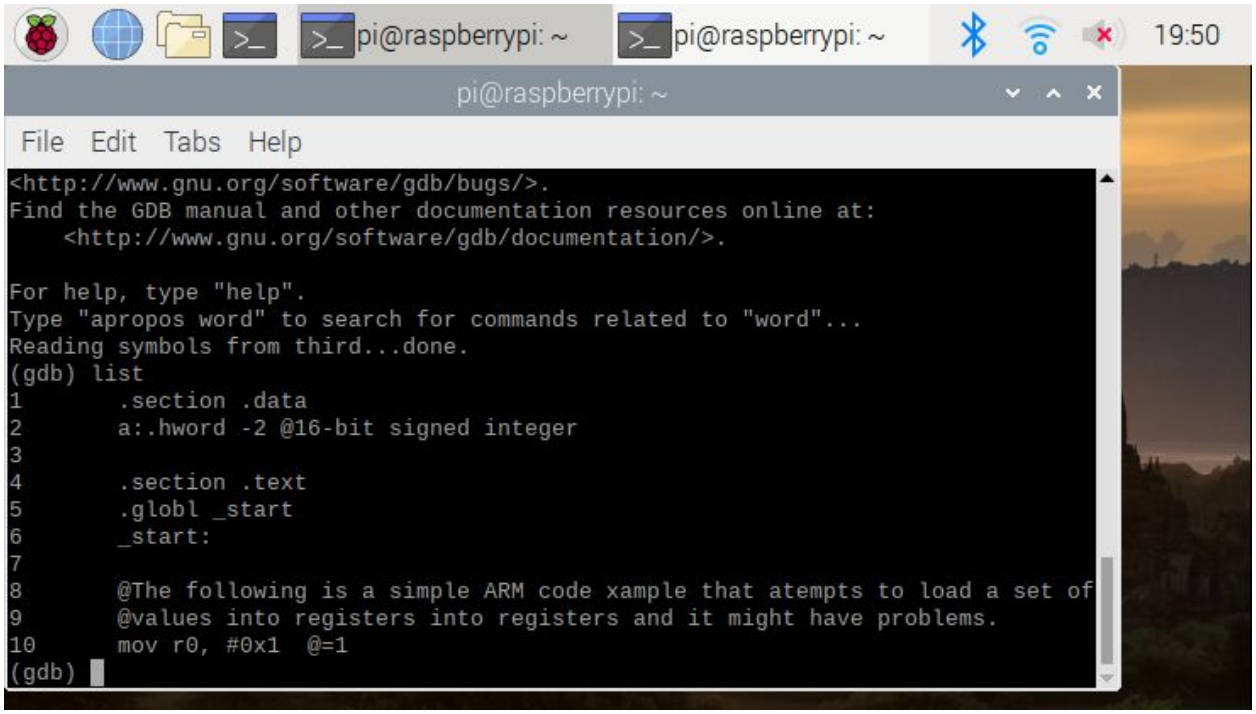
In the above seven screenshots you can see the code for part 3.

ARM Assembly Programming: Arteen Ghafourikia

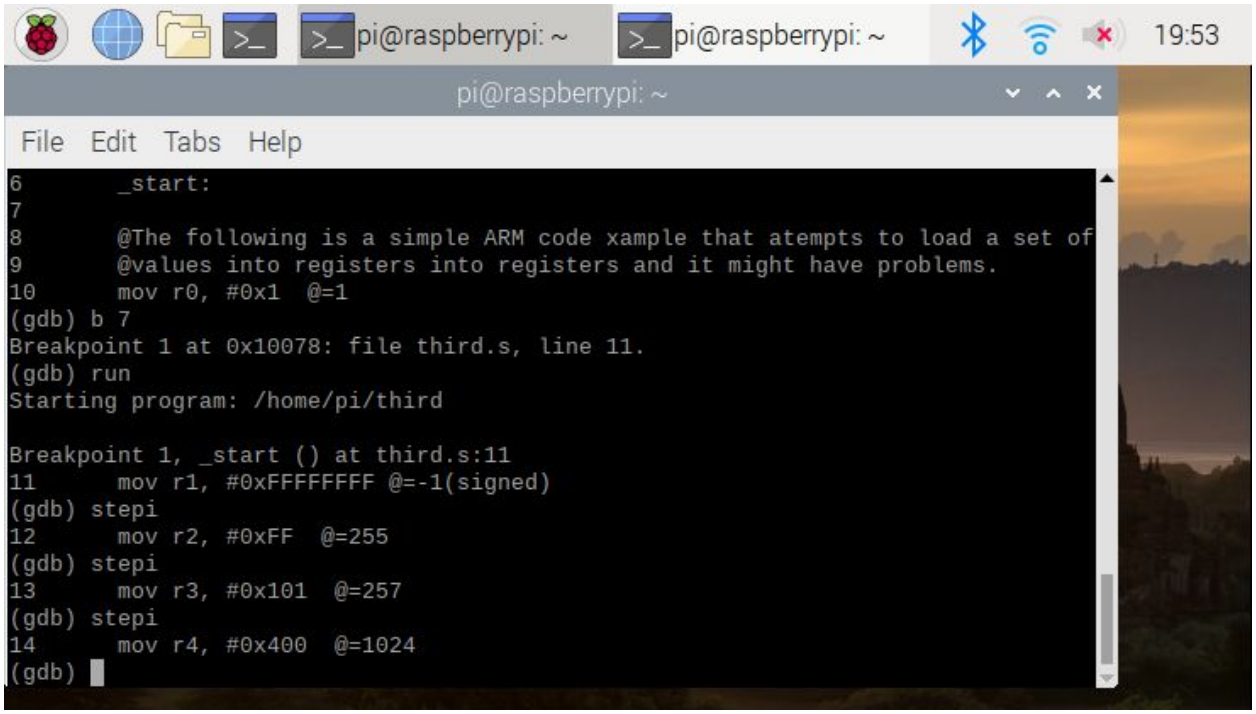
Part A



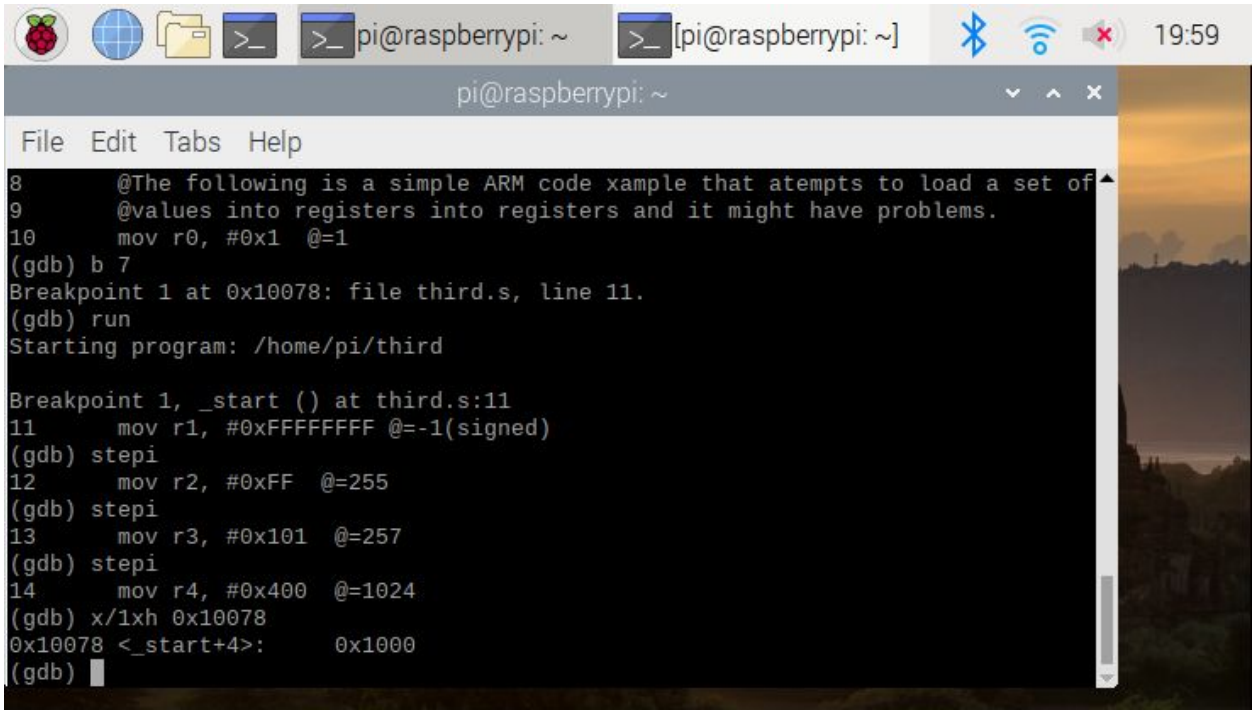
In this screenshot I am assembling and linking the program. I am then getting the debugger started.



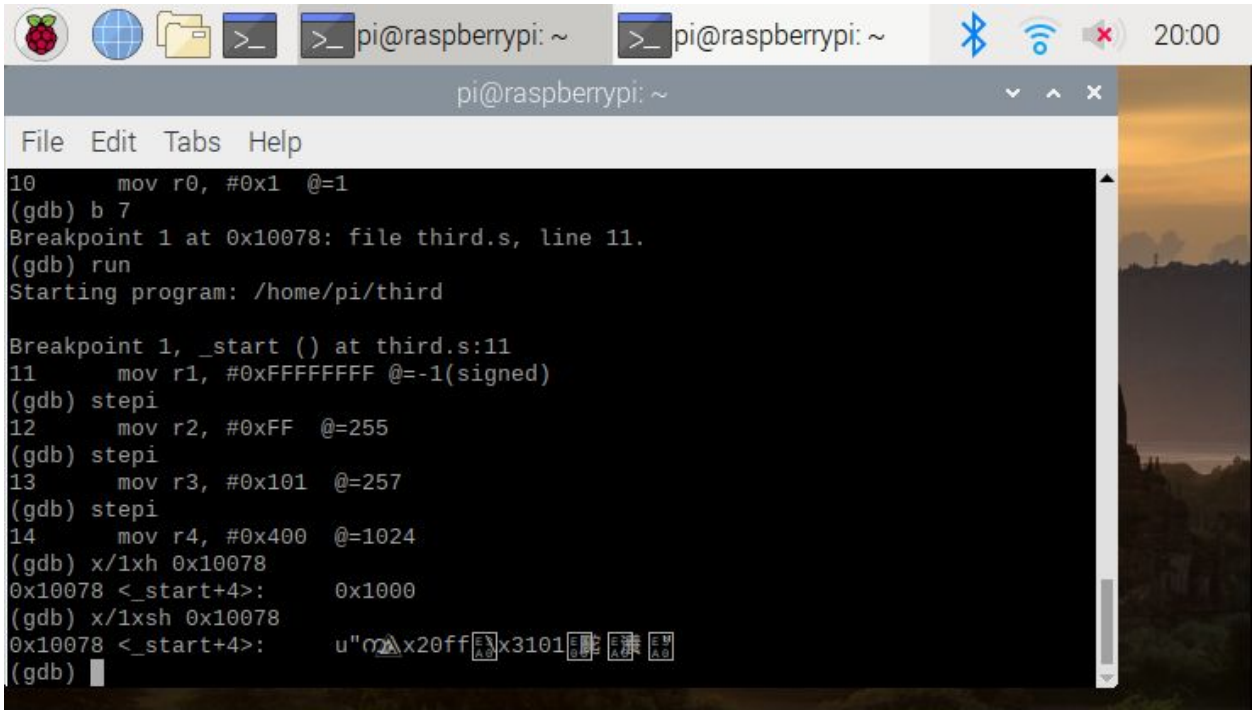
I am listing the first 10 lines of code.



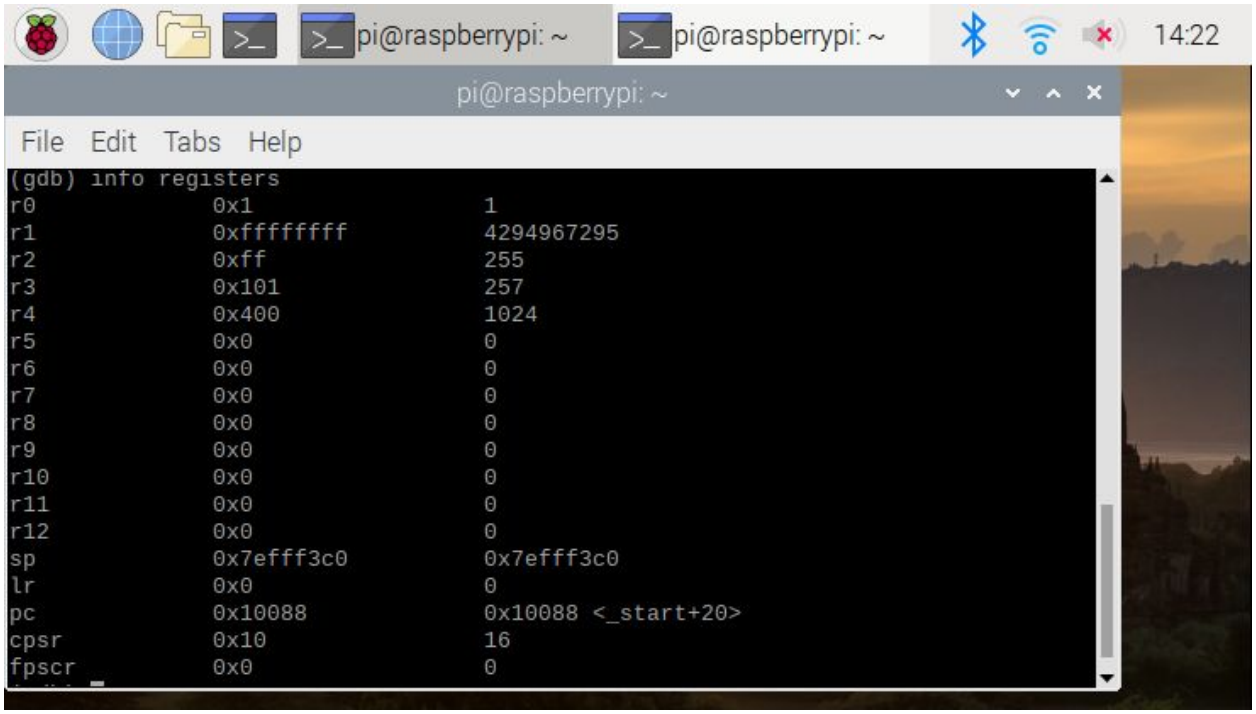
I ran the program and am stepping through each line of code.



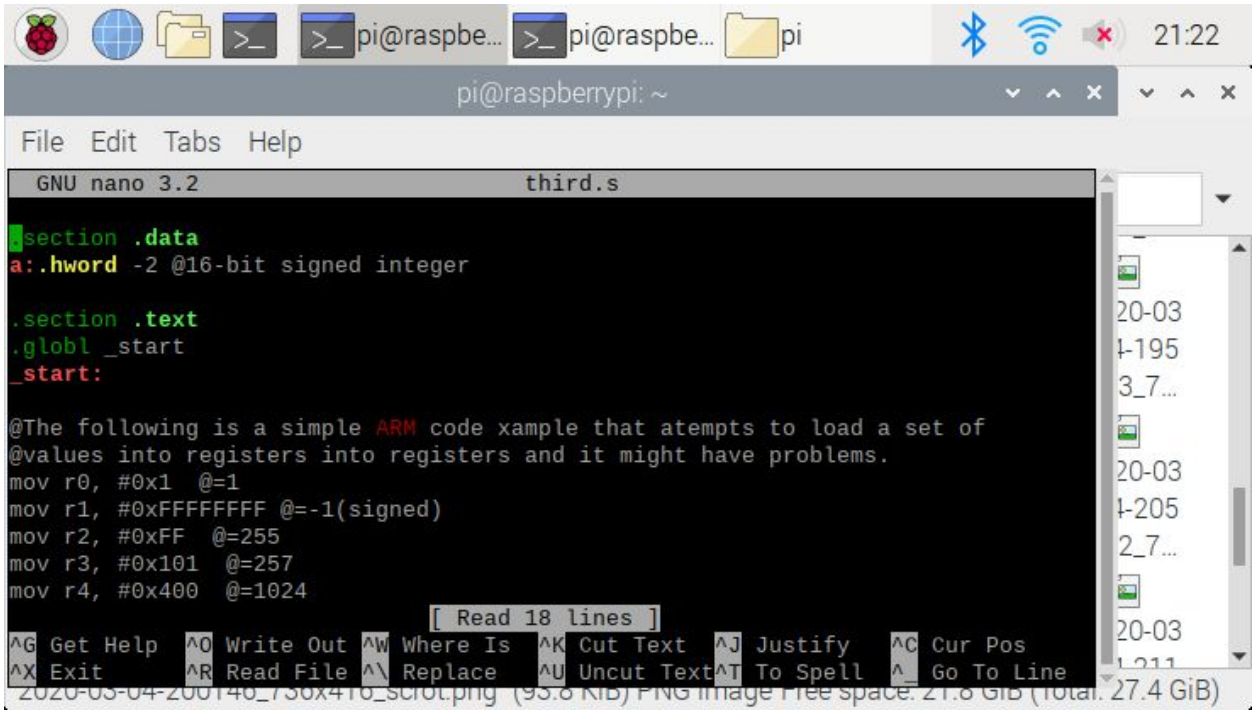
In this screenshot, I displayed the memory address.

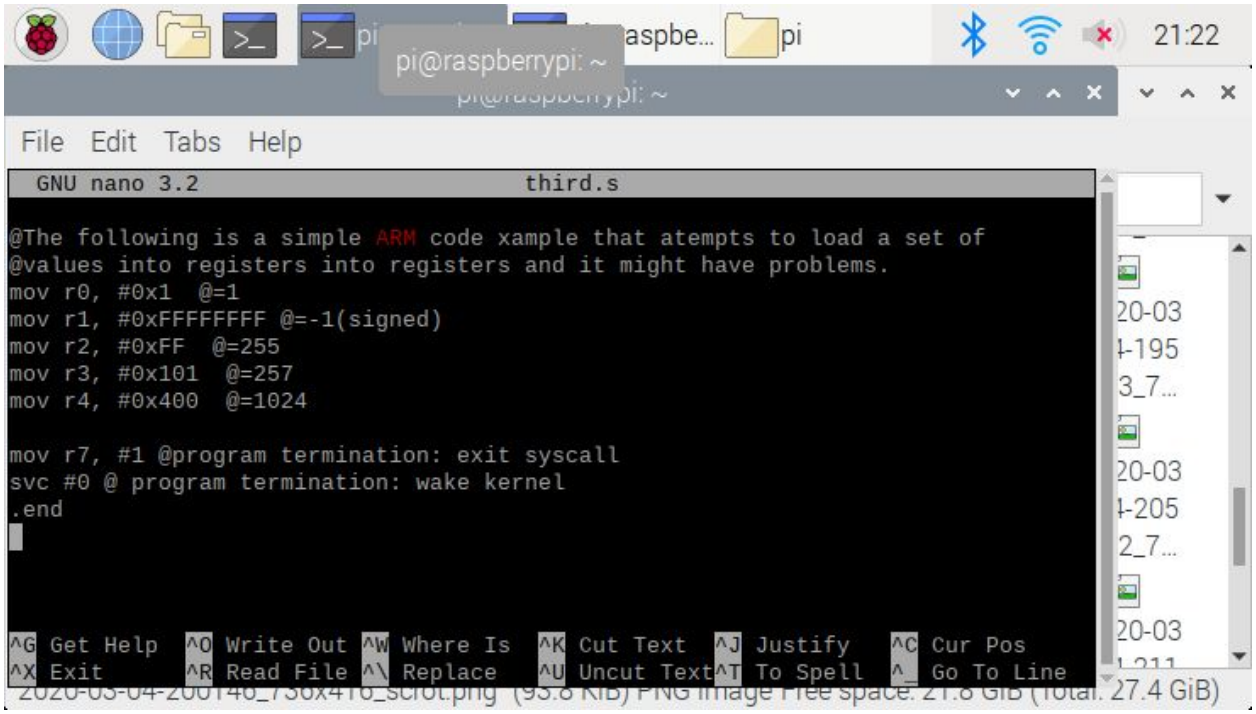


In this screenshot, I displayed the memory address of the signed value; however, you can see that some encrypted output appeared.



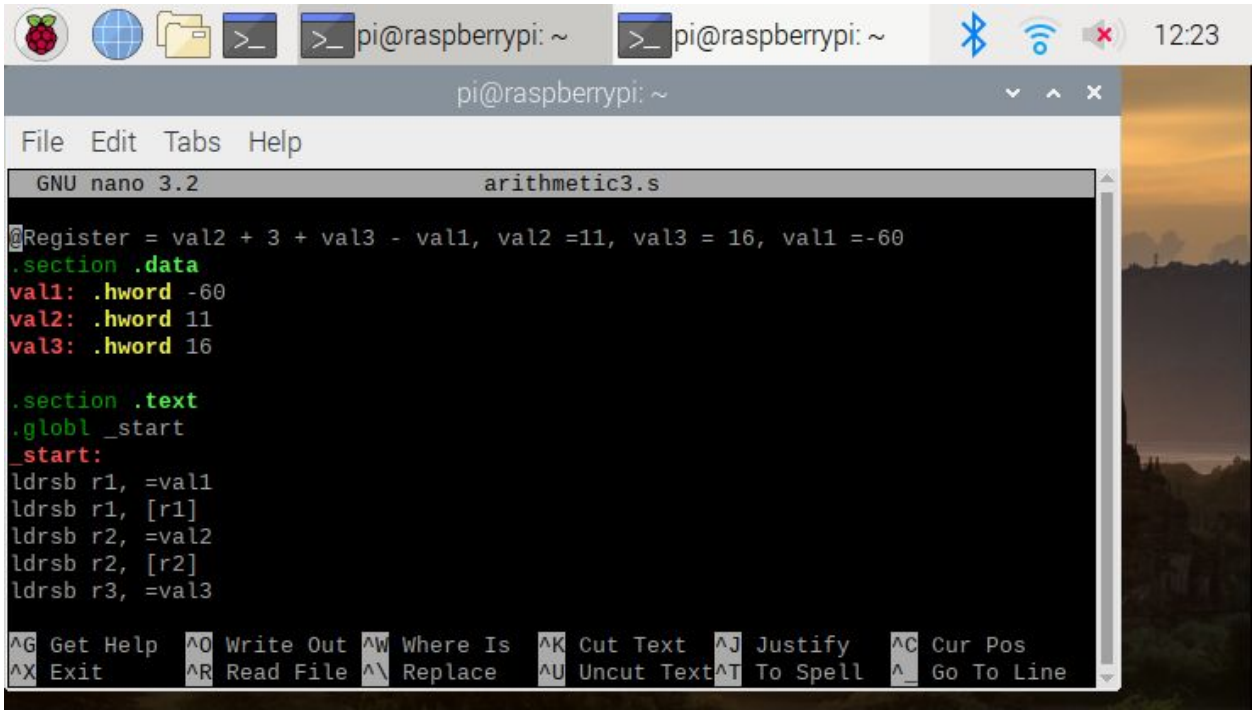
In this screenshot, I displayed the registers where you can see the designated values were assigned. The 2's complement of 0xffffffff would be -1 and all the other values are correctly displayed.

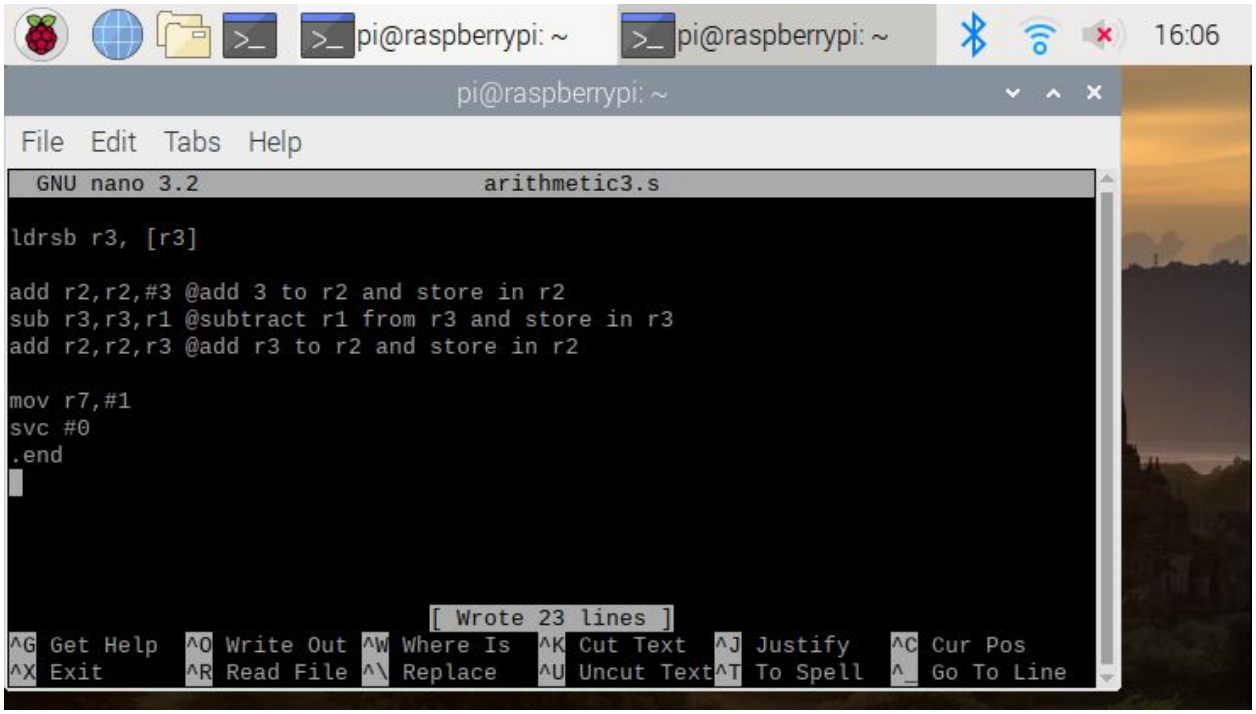




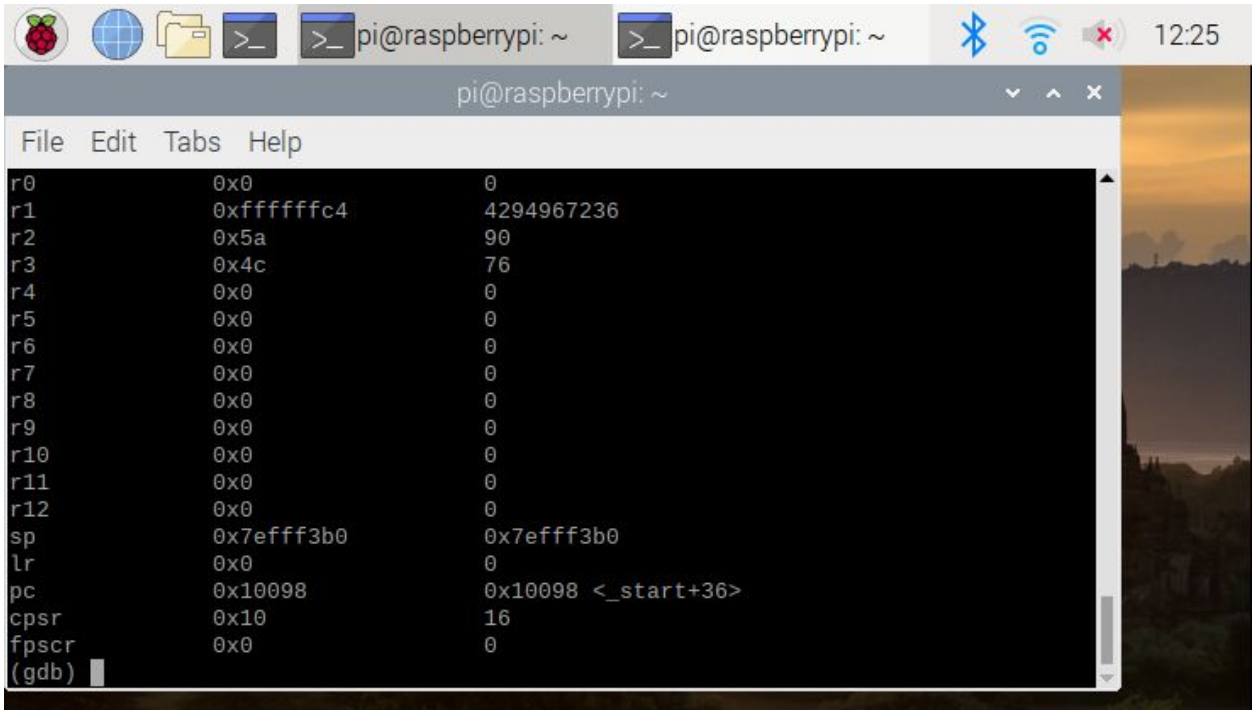
The two screenshots above are for part A of the arm assembly programming. In the given code, it would not work because .shalfword does not exist. It would give me an assembler message where it says “Error: unknown pseudo-op: ‘.shalfword’” therefore I changed it to .hword since .hword would be the equivalent to a 16-bit signed integer.

Part B

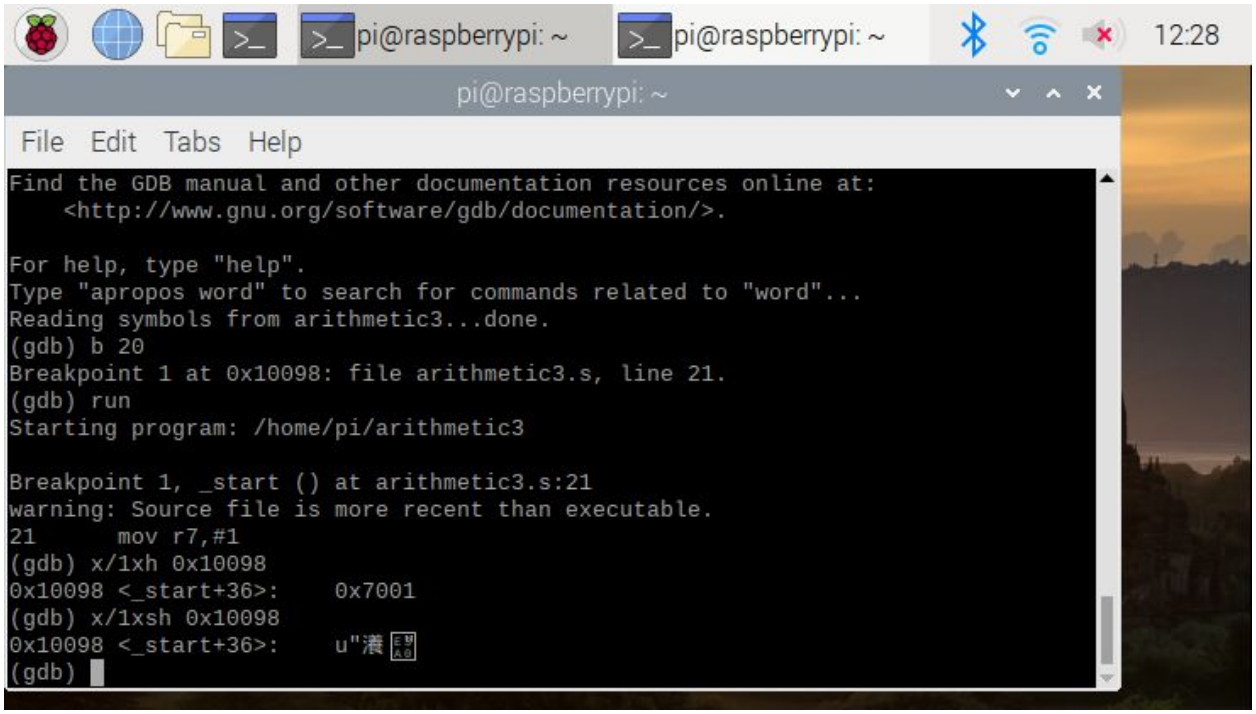




The two screenshots above are the code for part b of the ARM assembly programming.



In this screenshot, you can see the registers and the values in them. In r2, you can see the computation of the values, which is 90(5a hex), and it is the correct result if you do the given arithmetic. You can also see that in the cpsr you have 0x10(16).



In this screenshot, you can see that when you try to get the memory address of x/1xsh 0x10098, you get some encrypted output. However, when you do x/1xh 0x10098, you get a standard memory address.

Parallel Programming Skills Foundation Miguel Romo:

- **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
  - **Task -** A set of instructions executed by a processor.
  - **Pipelining -** Dividing the working in smaller bits by different processors, like an assembly line.
  - **Shared Memory -** Processors having the ability to access the same information, regardless of where the information is stored.
  - **Communications -** The ability to exchange information through shared memory bus or over a network.
  - **Synchronization** - Waiting for another task to accomplish before proceeding with another and/or the task(s) start at the same local equivalence point
- **Classify parallel computers based on Flynn's taxonomy.Briefly describe every one of them.**
  - **Single Instruction, Single Data -** Only one instruction stream is being acted on by the CPU during any one clock cycle. Only one data stream is being used as input during any one clock cycle.
  - **Single Instruction, Multiple Data (SIMD)** - All processing units execute the same instruction at any given clock cycle. Each processing unit can operate on a different data element. Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
  - **Multiple Instruction, Single Data** - Each processing unit operates on the data independently via separate instruction streams. Single Data: A single data stream is fed into multiple processing units. Few (if any) actual examples of this class of parallel computer have ever existed.
  - **Multiple Instruction, Multiple Data** - Every processor may be executing a different instruction stream. Multiple Data: Every processor may be working with a different data stream. Execution can be synchronous or asynchronous, deterministic or nondeterministic.
- **What are the Parallel Programming Models?**
  - Exist as an abstraction above hardware and memory architectures.
- **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**.
  - **Uniform Memory Access** - Identical processors and has equal access and access times to memory
  - **Non-Uniform Memory Access -** Often made by physically linking two or more SMPs.
  - OpenMP uses UMA because memory is shared with all processors.
- **Compare Shared Memory Model with Threads Model?**
  - **Shared Memory -** processes/tasks share a common address space, which they read and write to asynchronously.
  - **Thread Memory** - is a type of shared memory, but have thread that branch out, each has local data, but also shares resources.
- **What is Parallel Programming?**

- Parallel programing allows tasks to be shared so that can be accomplished simotaiouly.

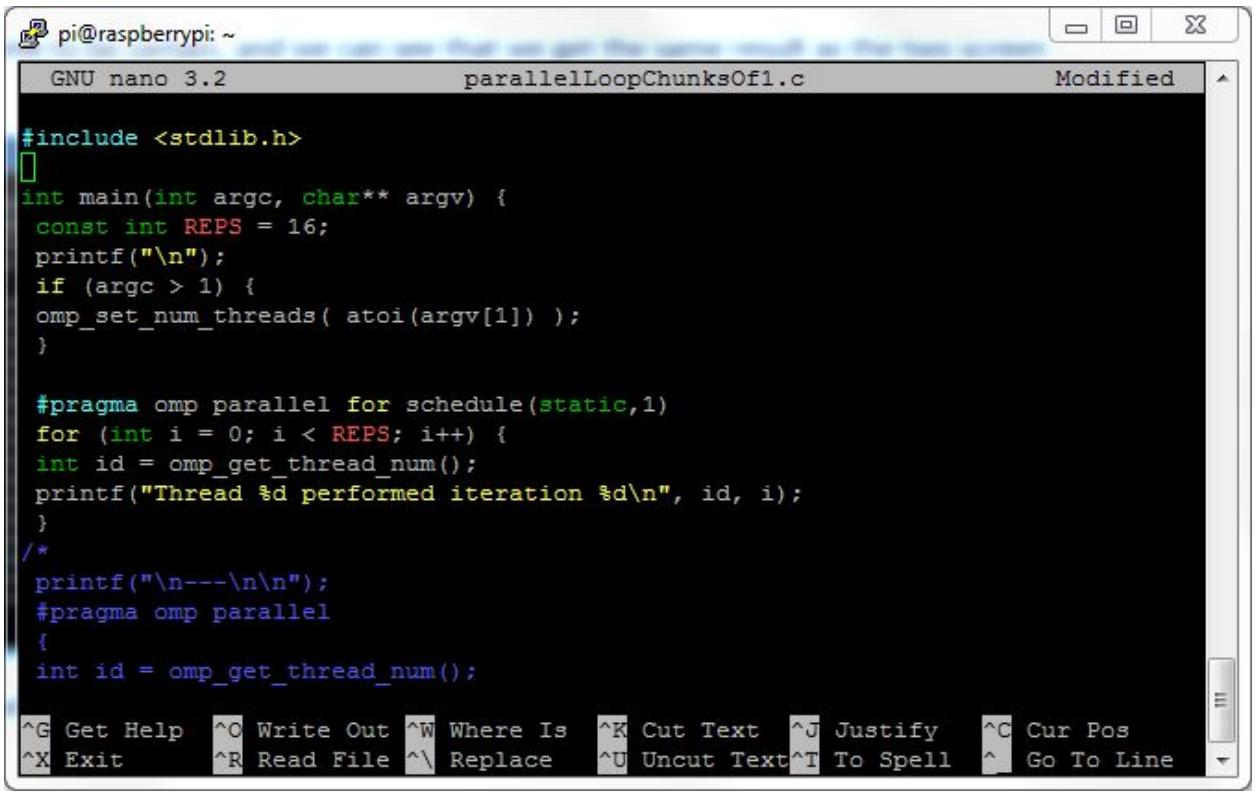
➢ **What is system on chip (SoC)? Does Raspberry PI use system on SoC?**

- SoC is essentially the entire computer on one chip. The Raspberry PI does use Soc.

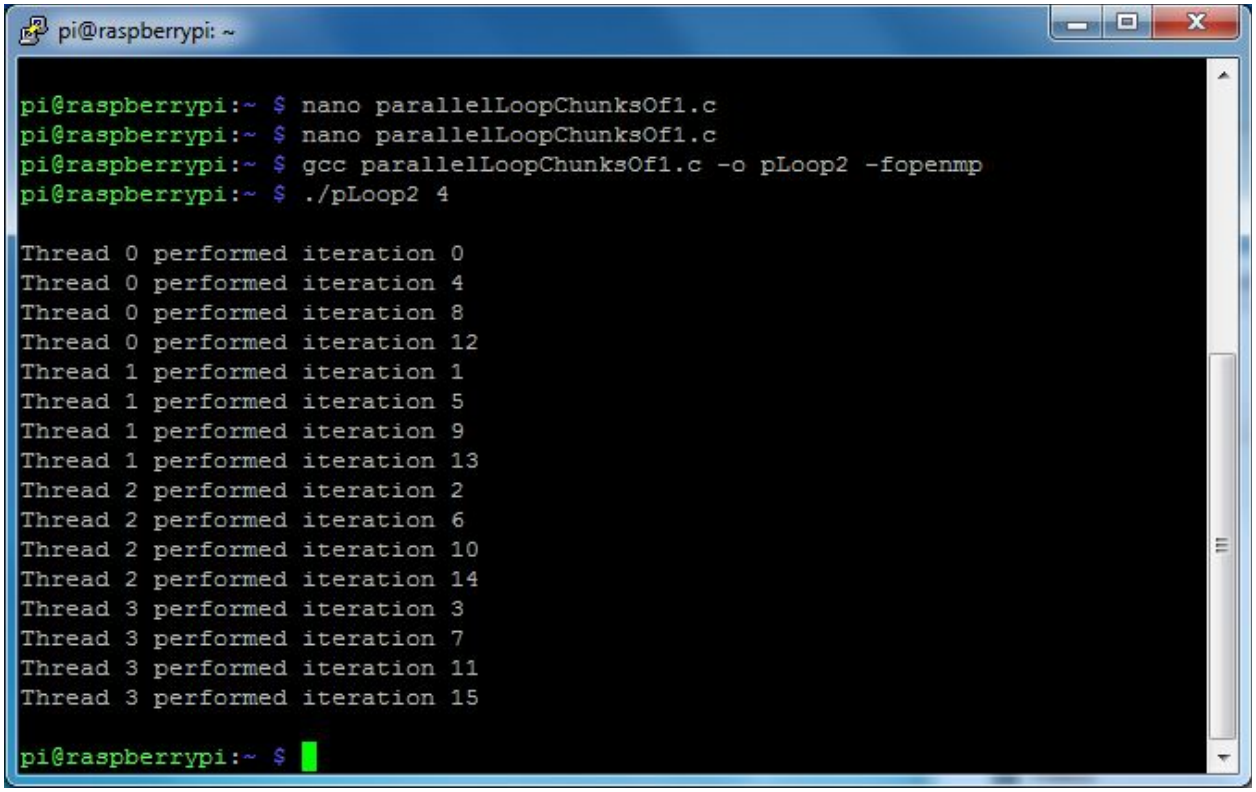
➢ **Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

- Advantages include, size, it is only a bit larger than a CPU and contains more functionality. They use less power, and cost less.

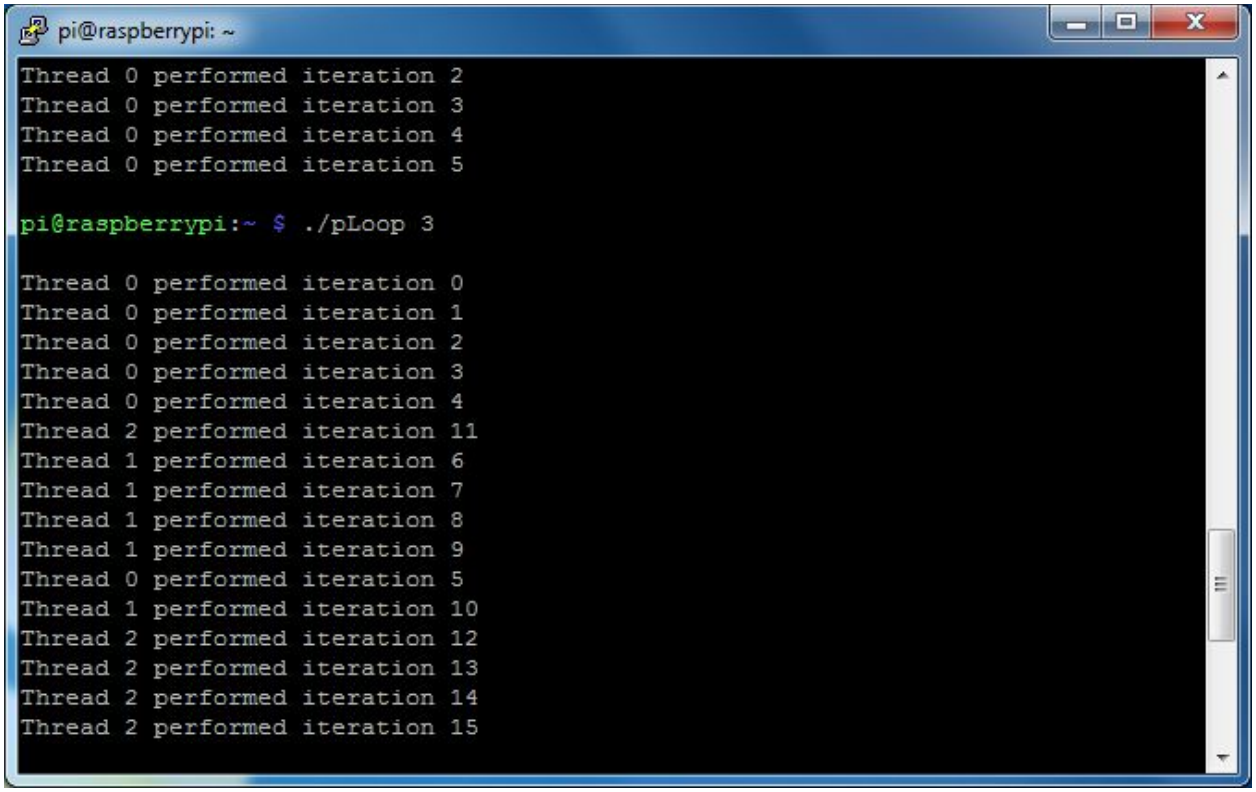
Parallel Programming Basics Miguel Romo:



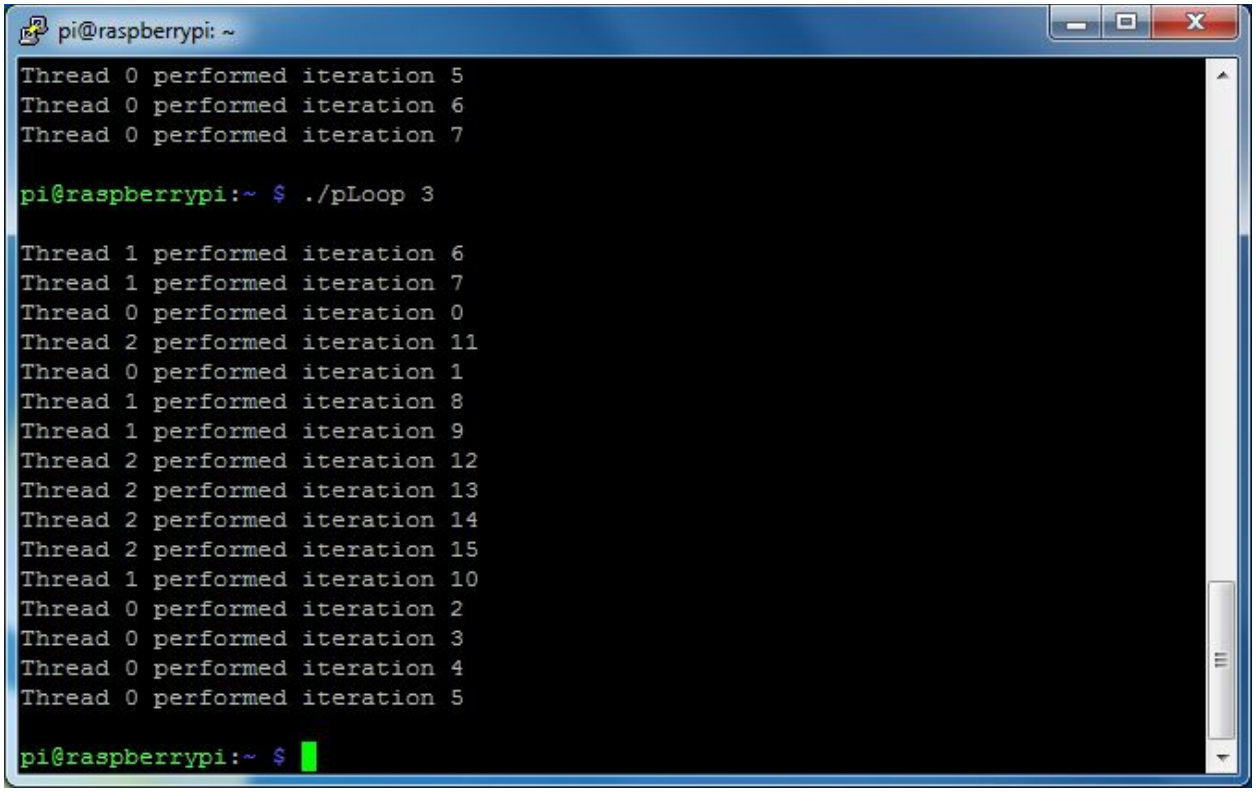
This is the code for parallelLoopEqualChunks.c.



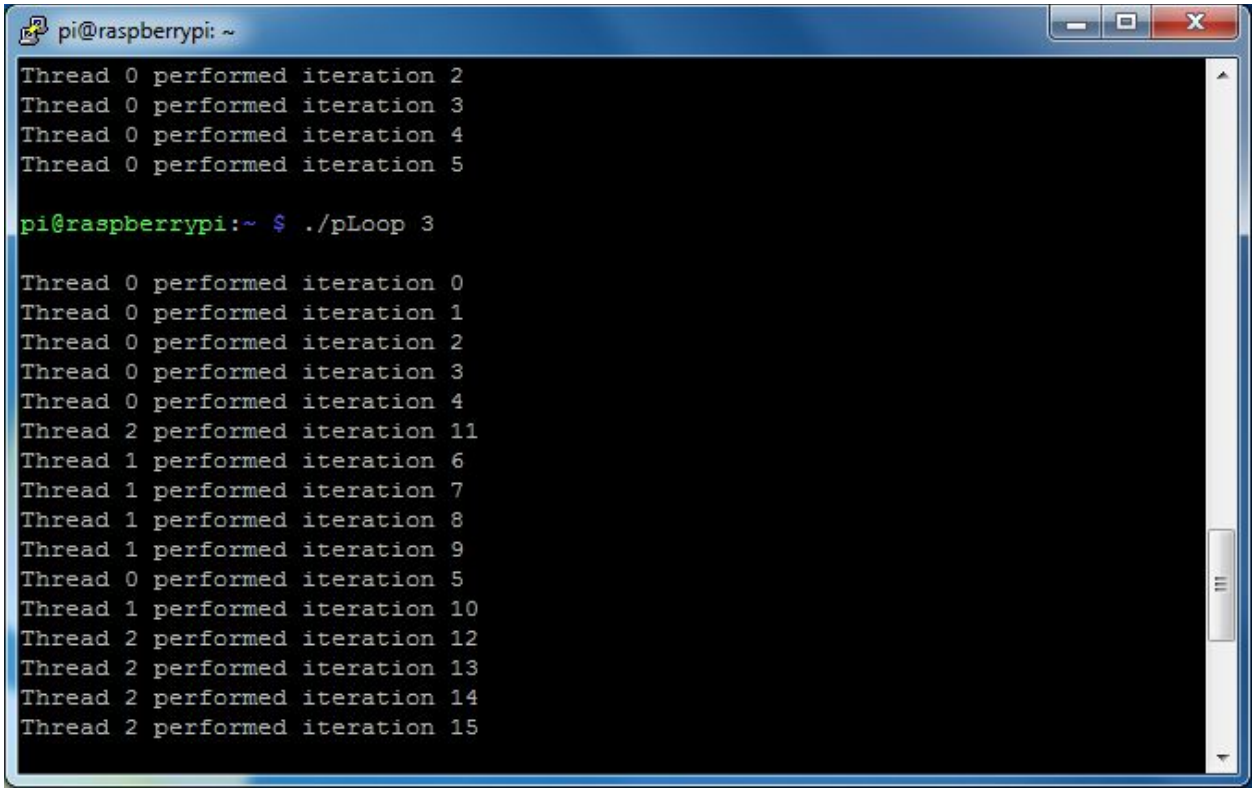
Here, I ran the program using ./Loop 4. The 4 is called a command-line argument that indicates how many threads to fork. We use 4 since the Raspberry Pi has a 4-core processor. We verify that the behavior is the same sort of decomposition based on the diagram in the Parallel Programing document.



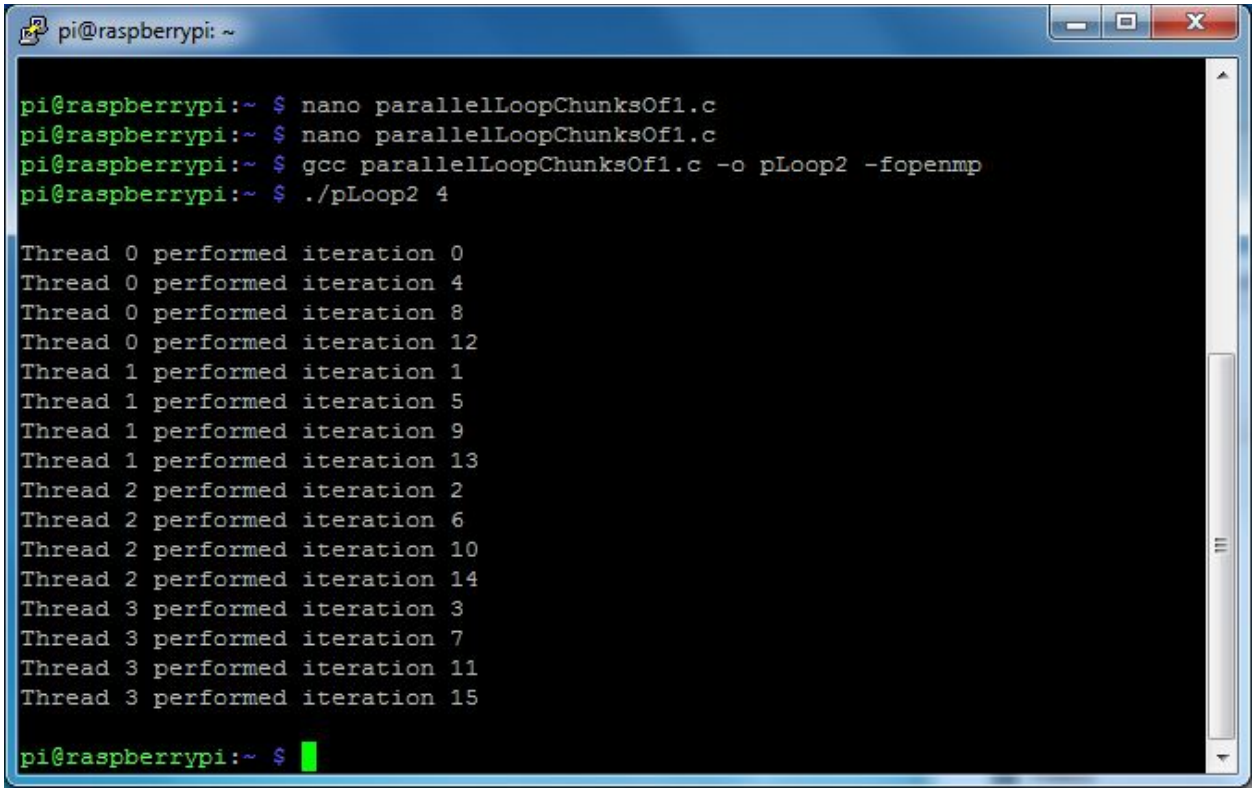
We run the program again, this time replacing the 4 with the value of 3. We do this to test what happens when the number of iterations is not evenly divisible. We can see from the first test that the first thread, thread 0 complete the additional iteration.



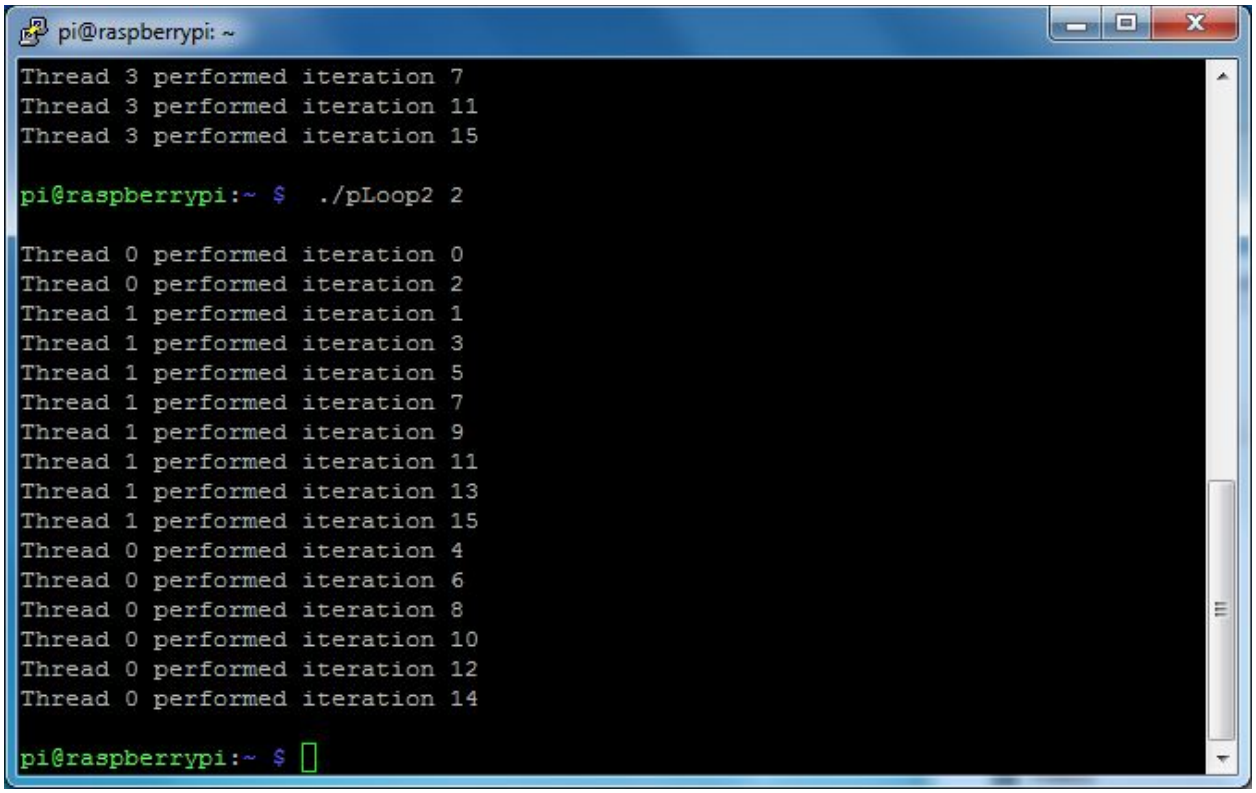
We try running it again to be certain that this is the case. You can see that we get the same results as above.



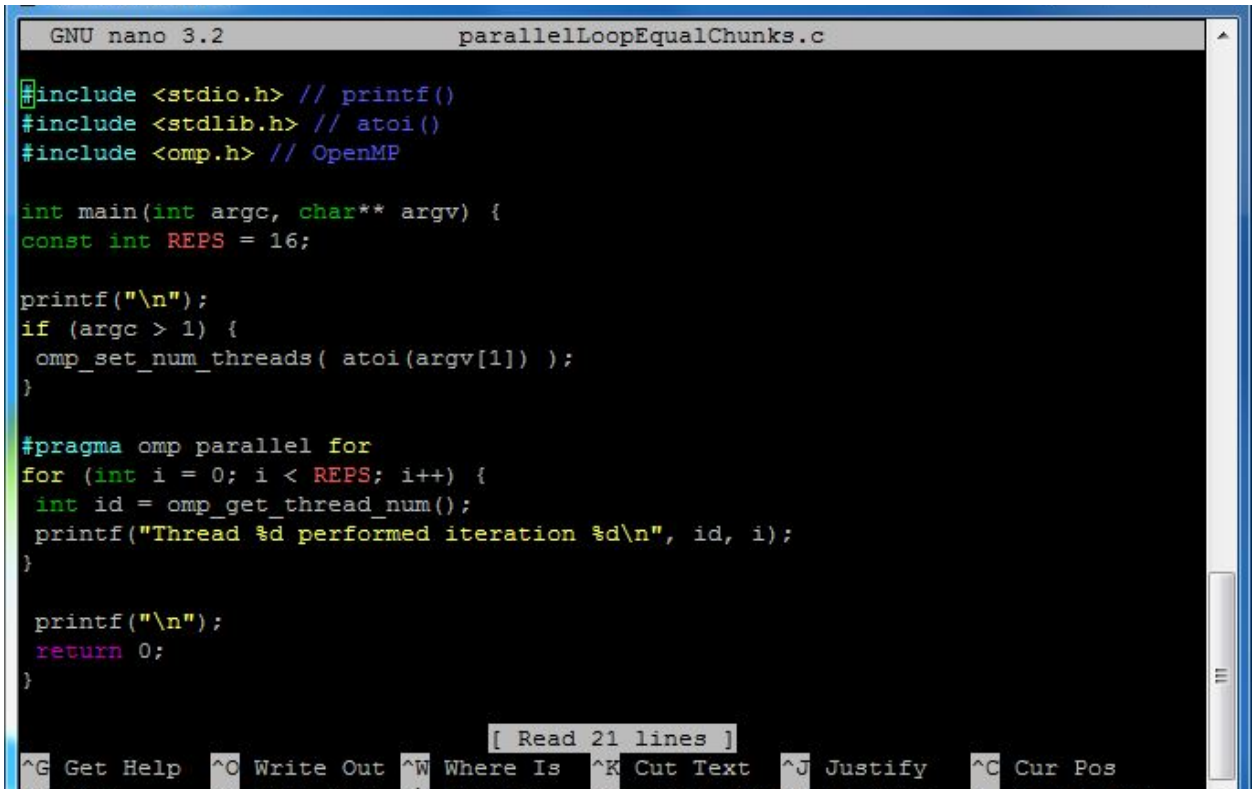
We run it one last time to be certain, and we can see that we get the same result as the two screenshots above.



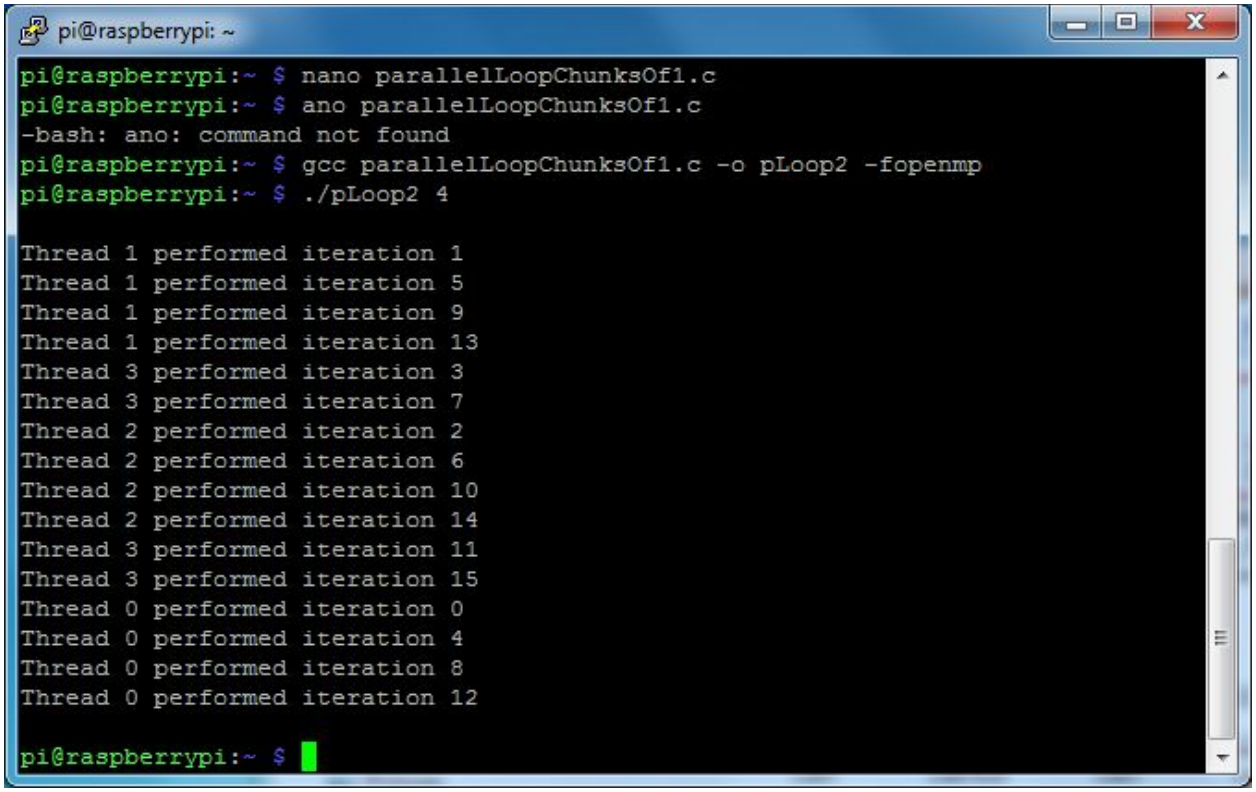
Here we run “parallelLoopChunksOf1.c” using “./pLoop2 4”.



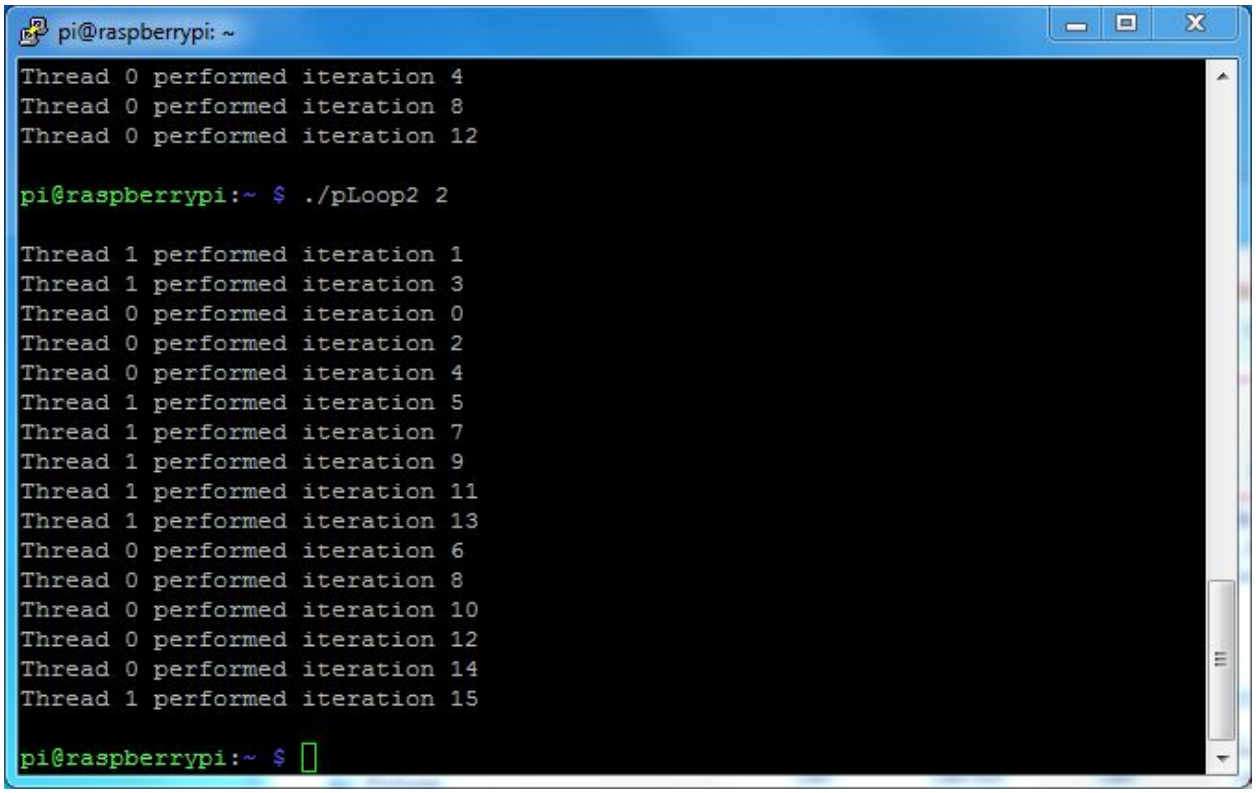
We replace 4 with 2, to change the number of threads.



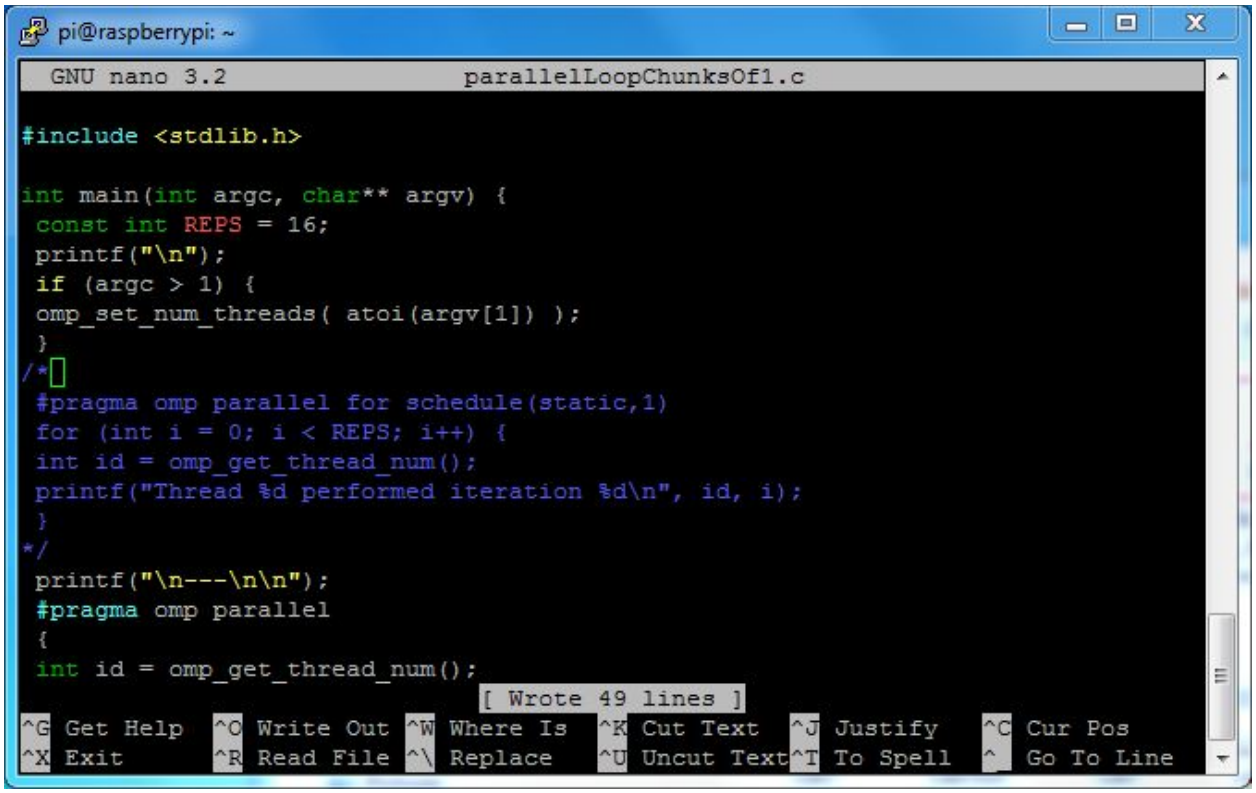
Here is our code for “parallelLoopChunksof1.c”.



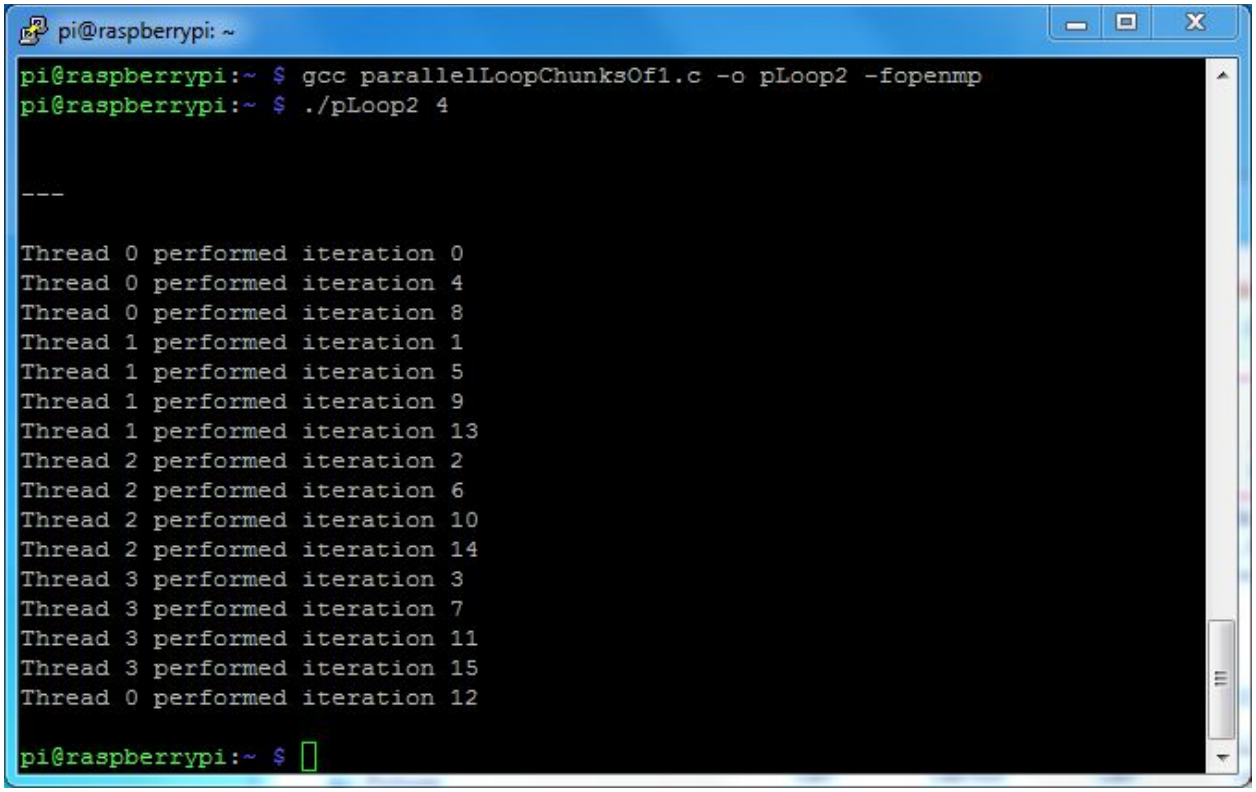
After creating an executable file, we run it using "./Loop2 4"



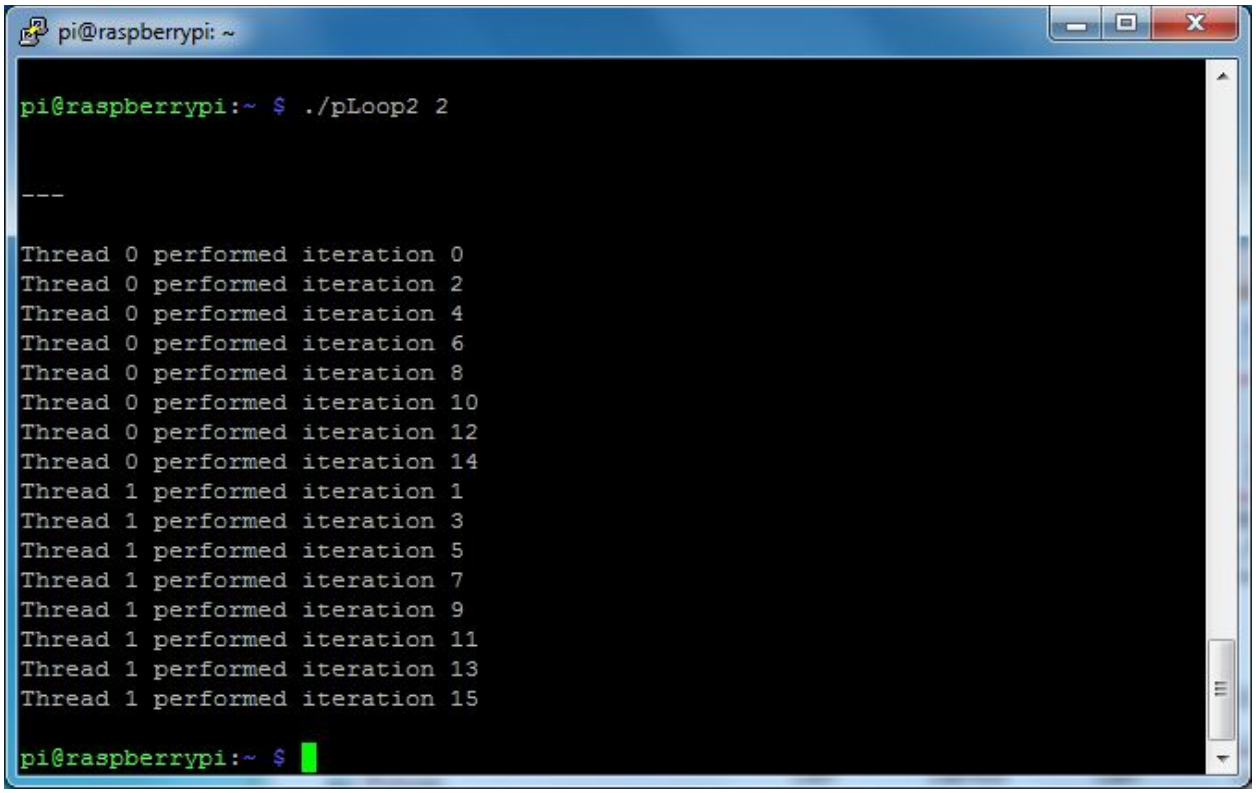
We replace "./Loop2 4" with "./Loop2 2". We can conclude that in the uncommented section there is a repetition or a pattern that occurs. That is not true for the first part and it look as though the iteration occur randomly.



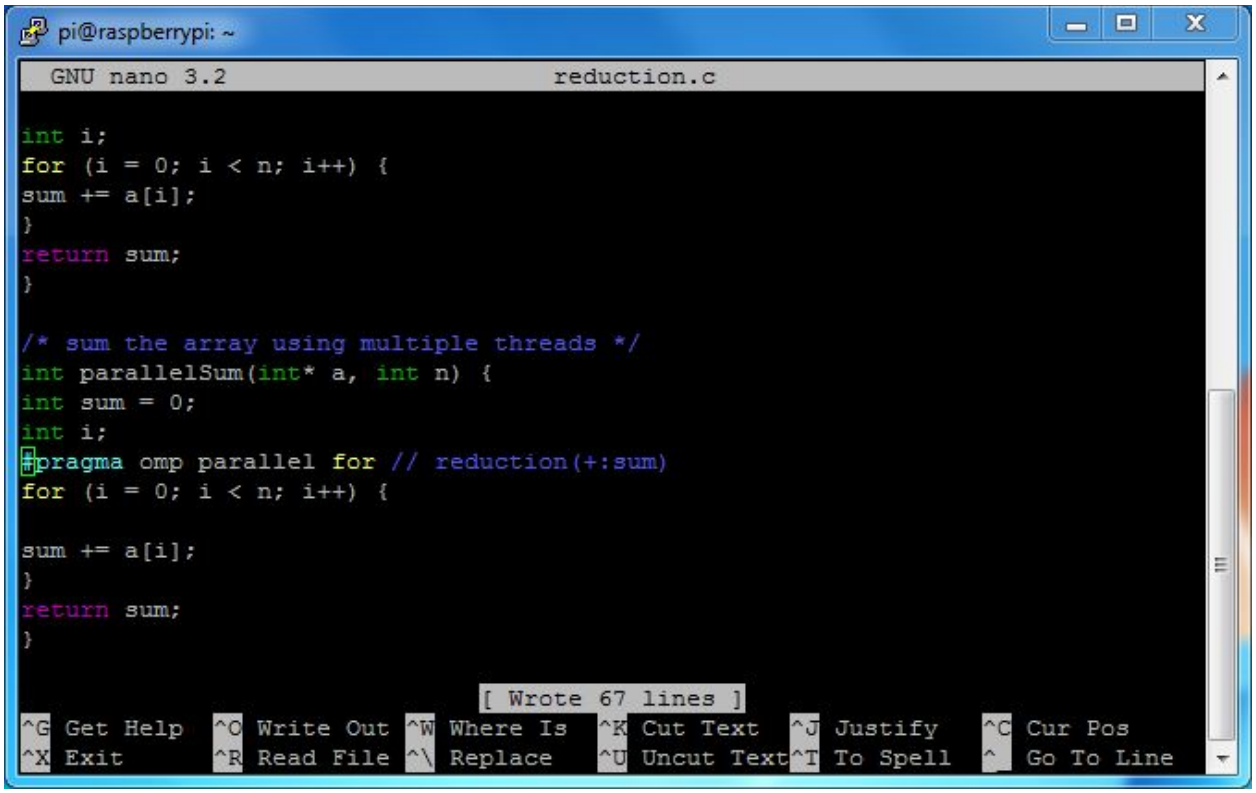
We go back to our code and uncomment lines 16 – 25 and comment out lines 10 – 15.



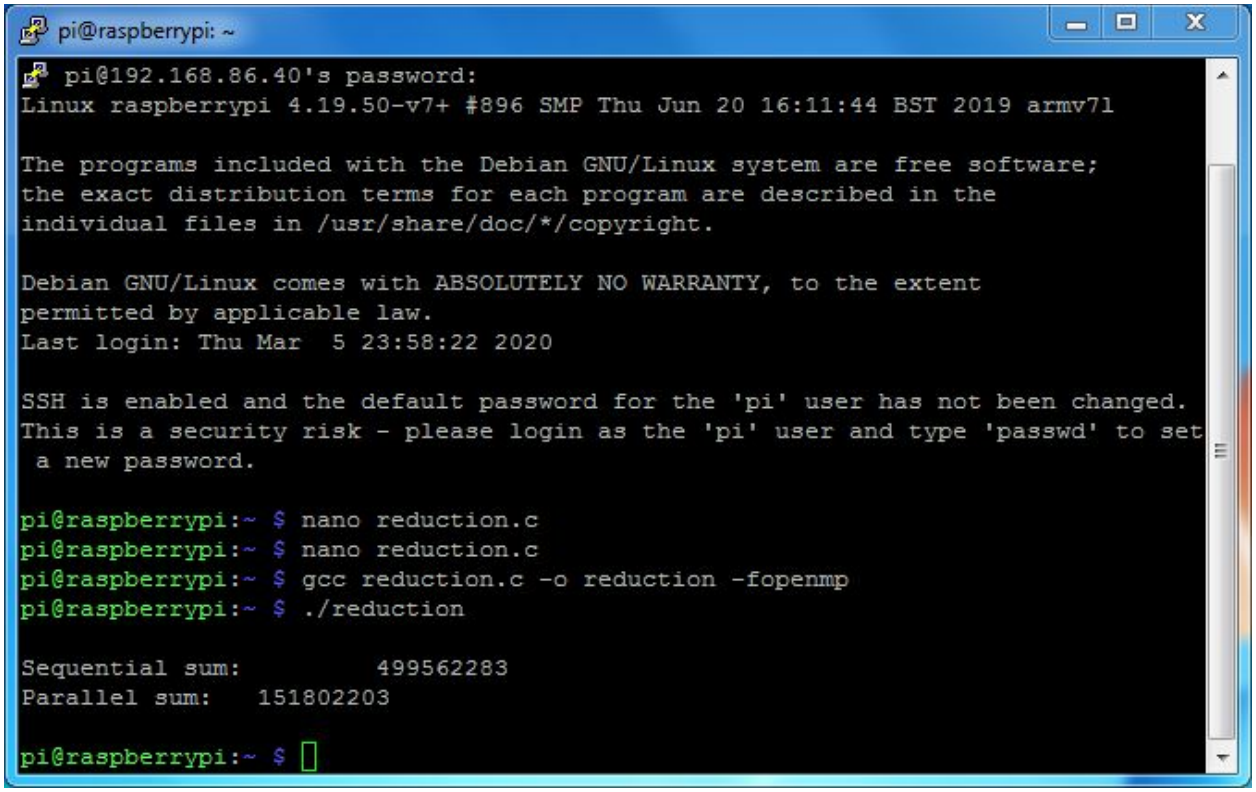
Here are the results after creating an executable file, we run it using "./Loop2 4"



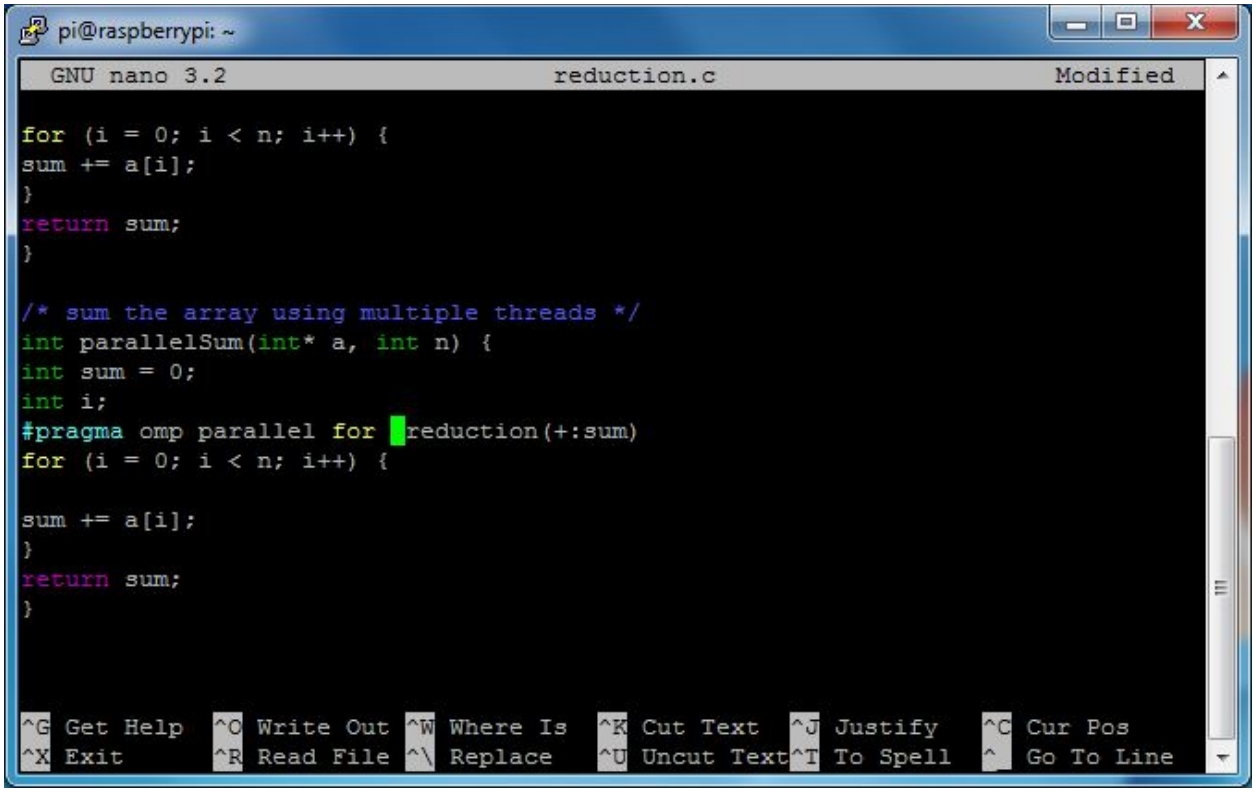
We replace “./Loop2 4” with “./Loop2 2”.



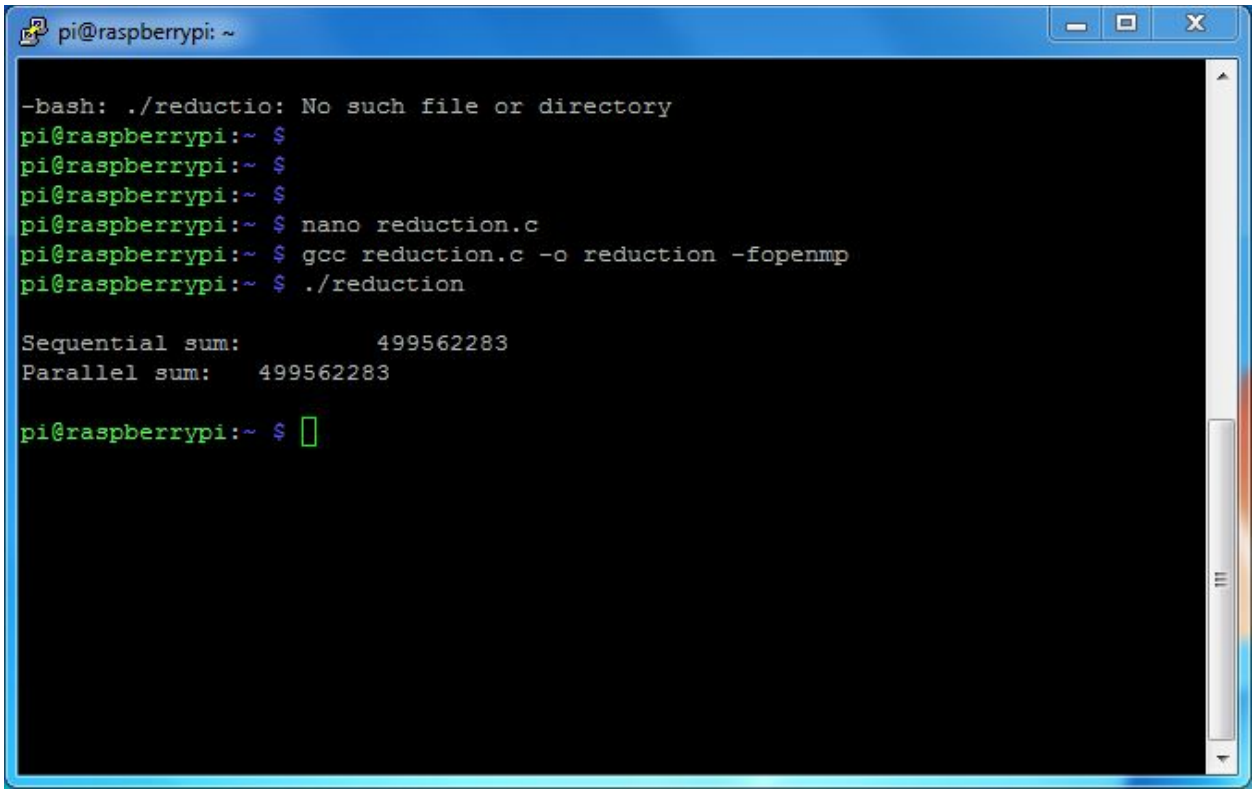
Here is the code for “reduction.c”. With the first “//” removed from line 39.



After creating an executable file, we run the program using "./reduction ", and these are the results.

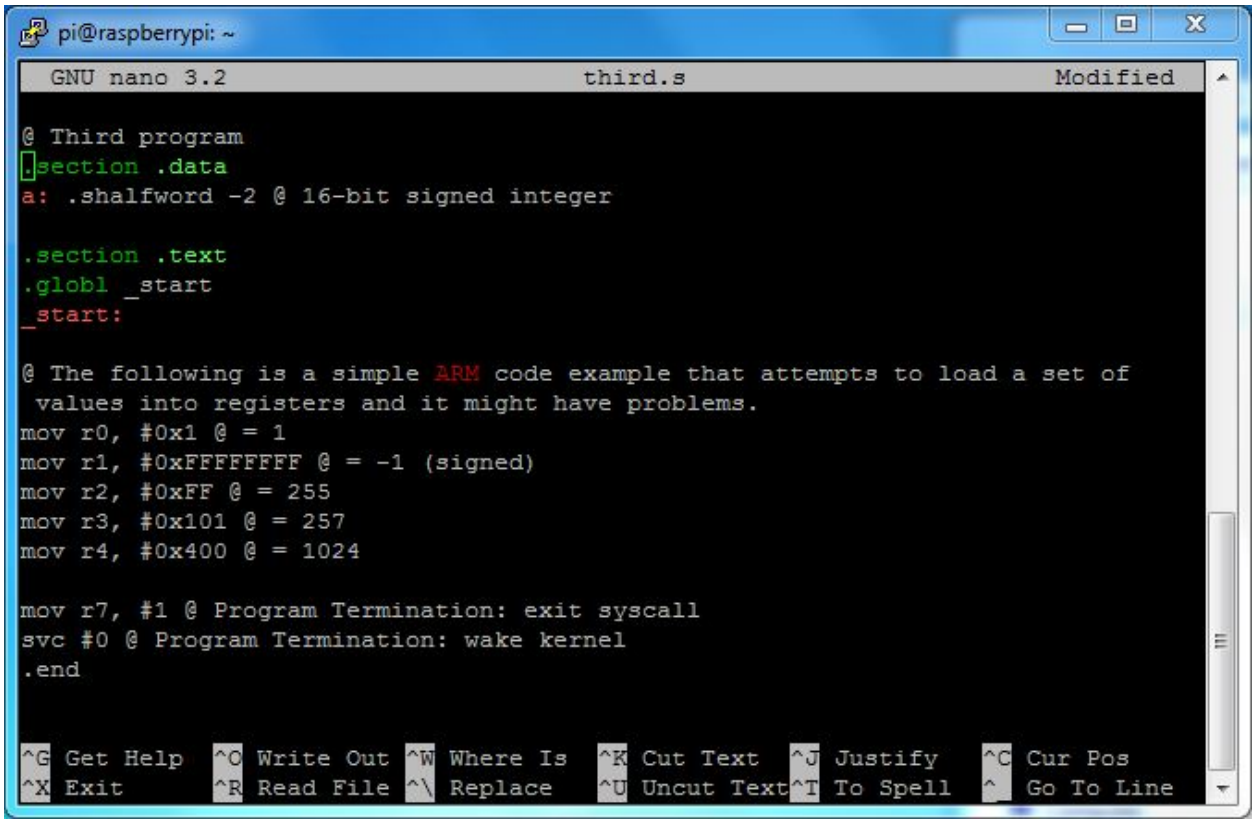


Here is the code for “reduction.c”. With the first and second “//” removed from line 39.

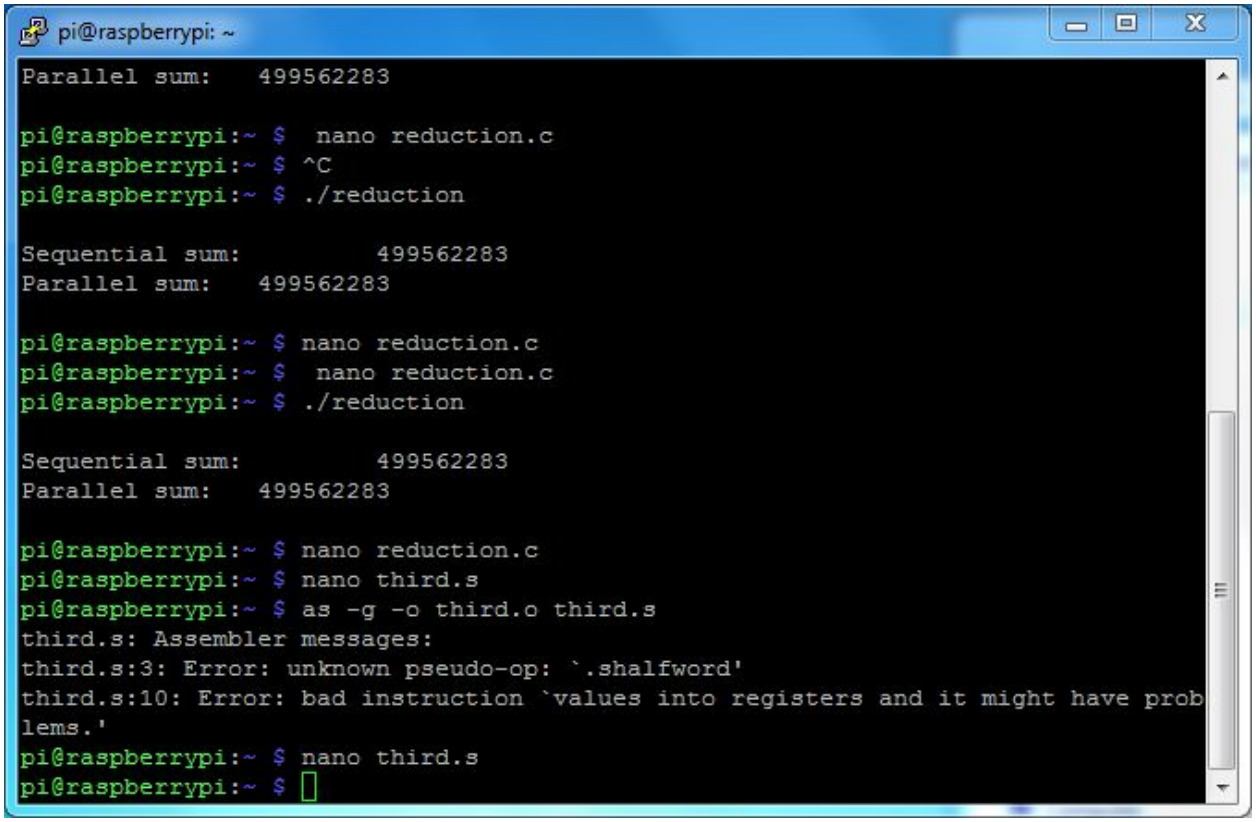


Here are the results after creating a new executable file and running the program using, "./reduction". For this I can see that it is necessary to have both parts uncomment to get the correct answer.

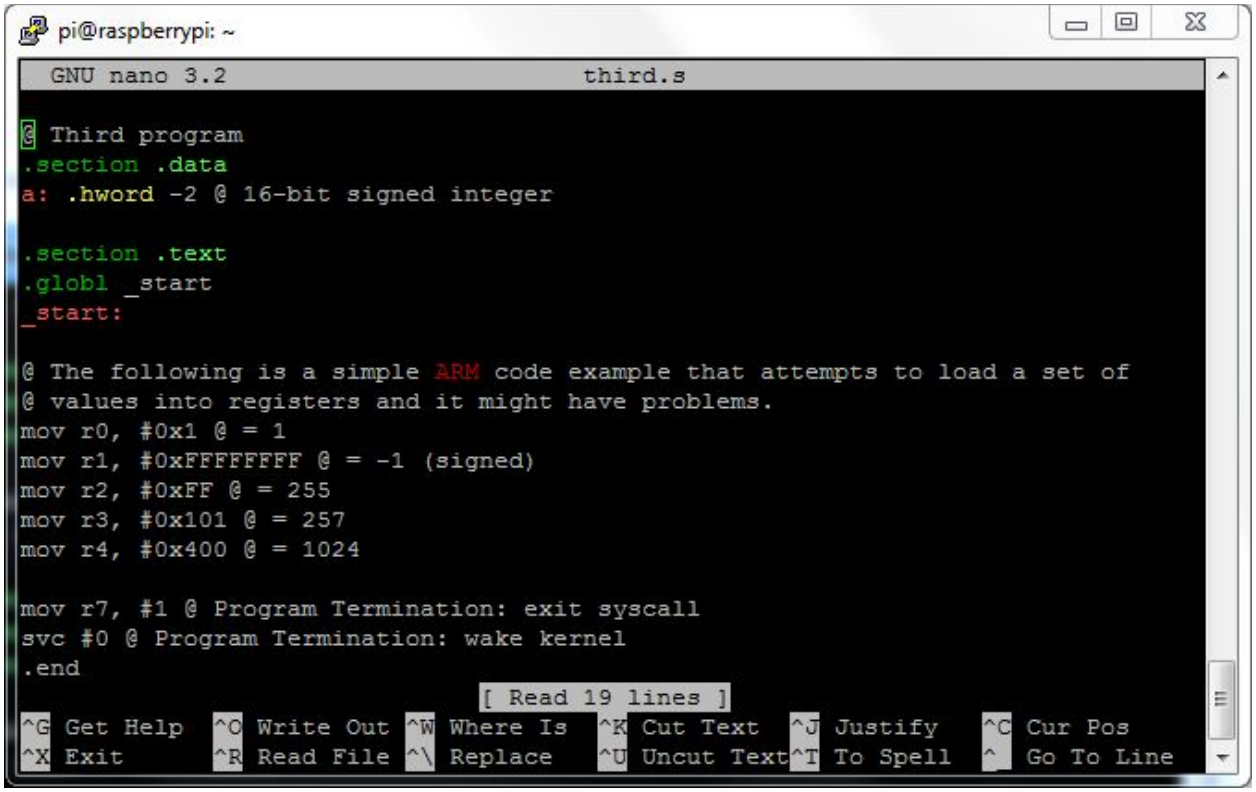
ARM Assembly Programming Miguel Romo:
Part A:



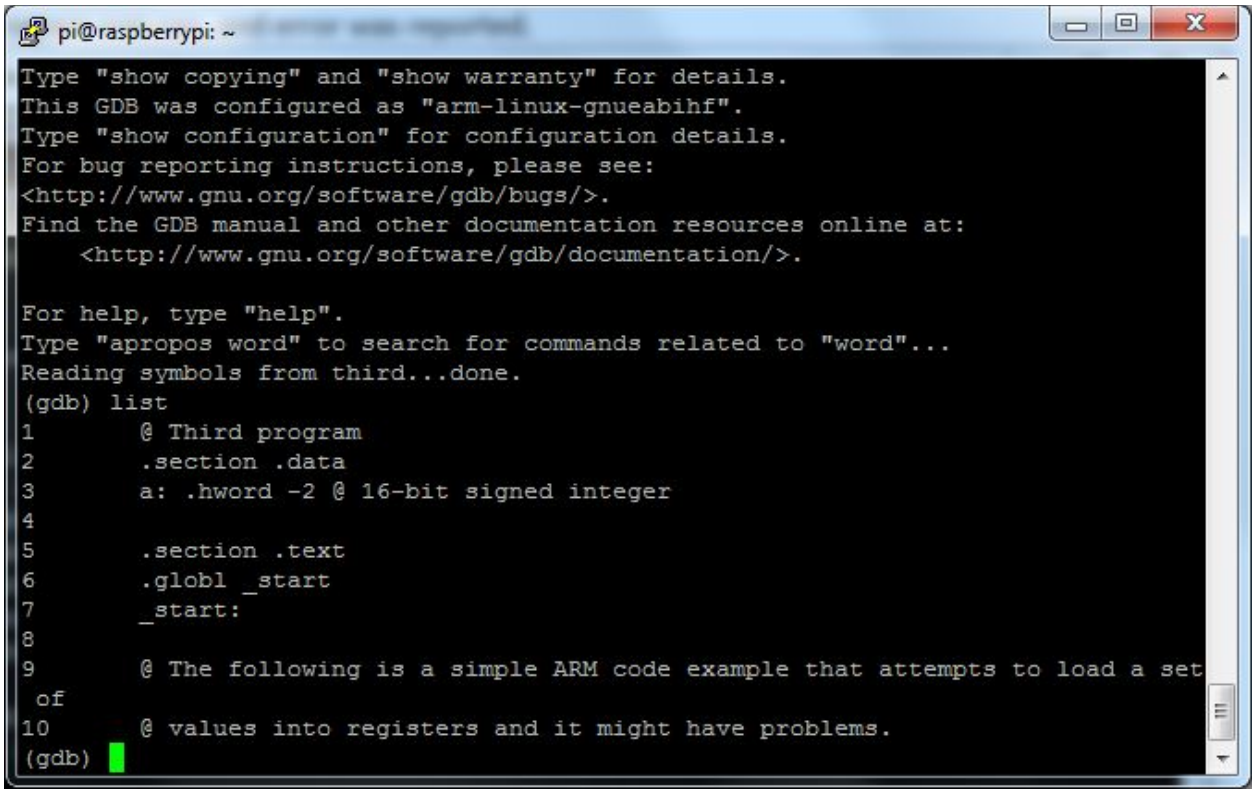
Here is the code for "third.s".



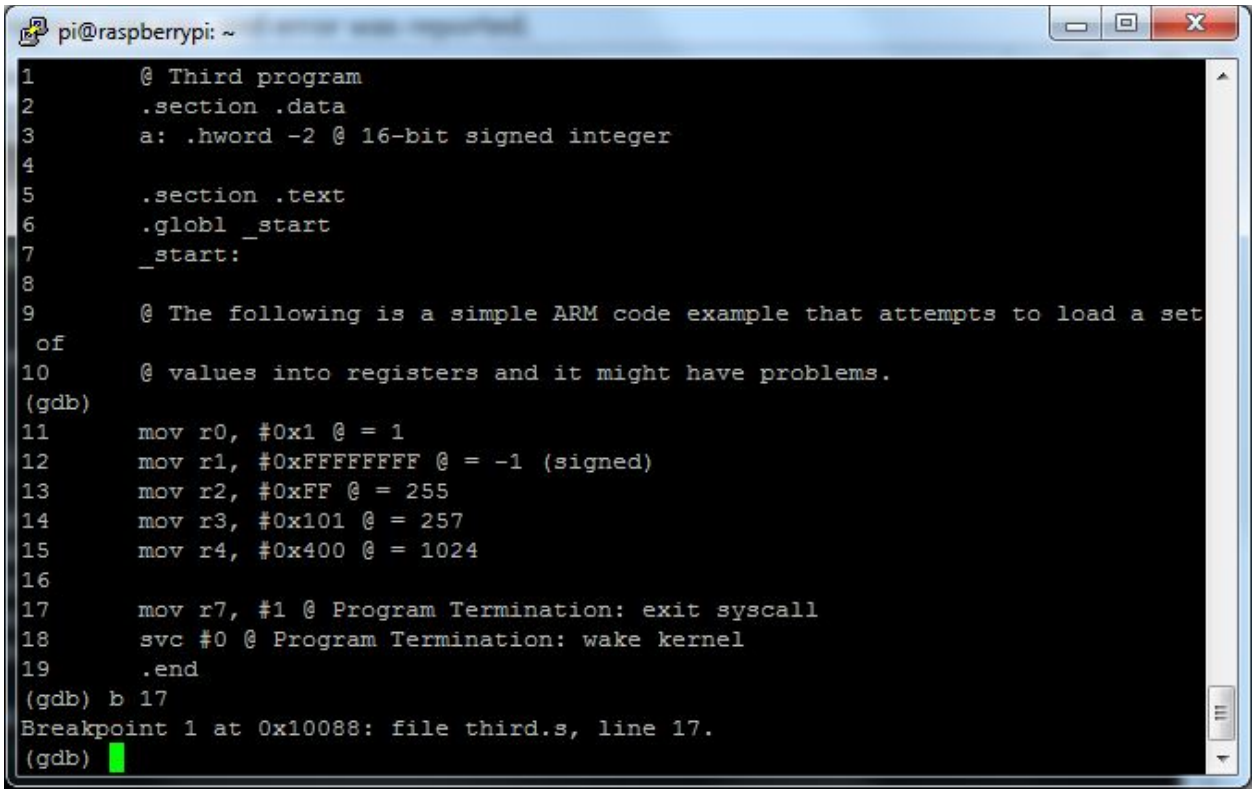
After the program was run, an error was reported.



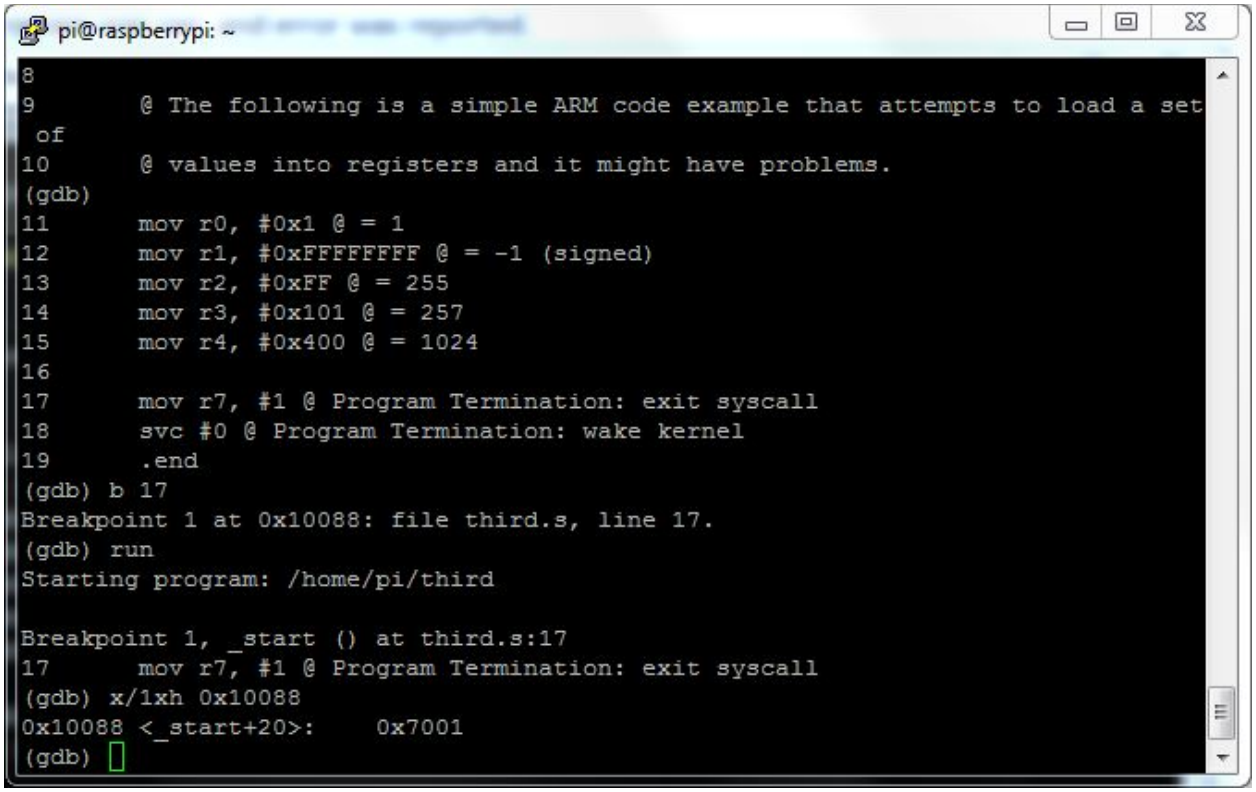
I went back to the code and replaced ".shalfword" with "hword".



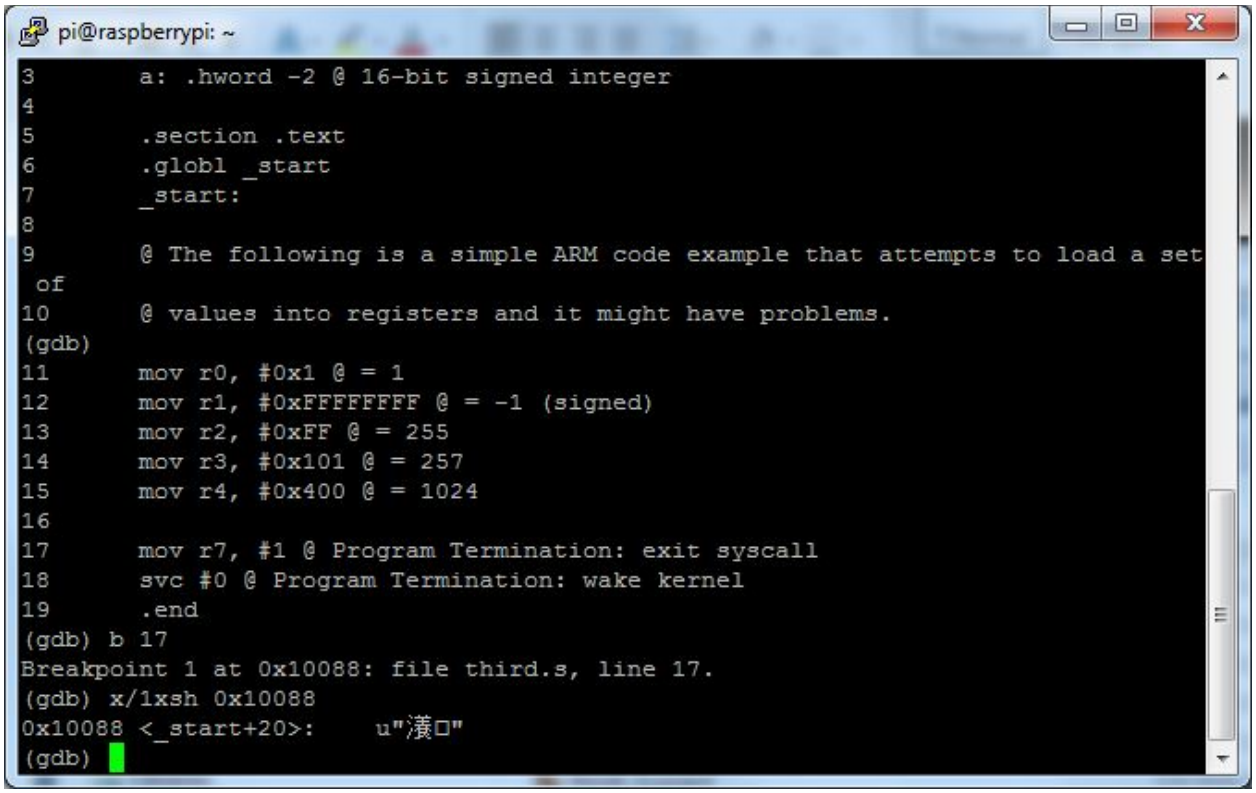
Code executed successfully and began debugging.



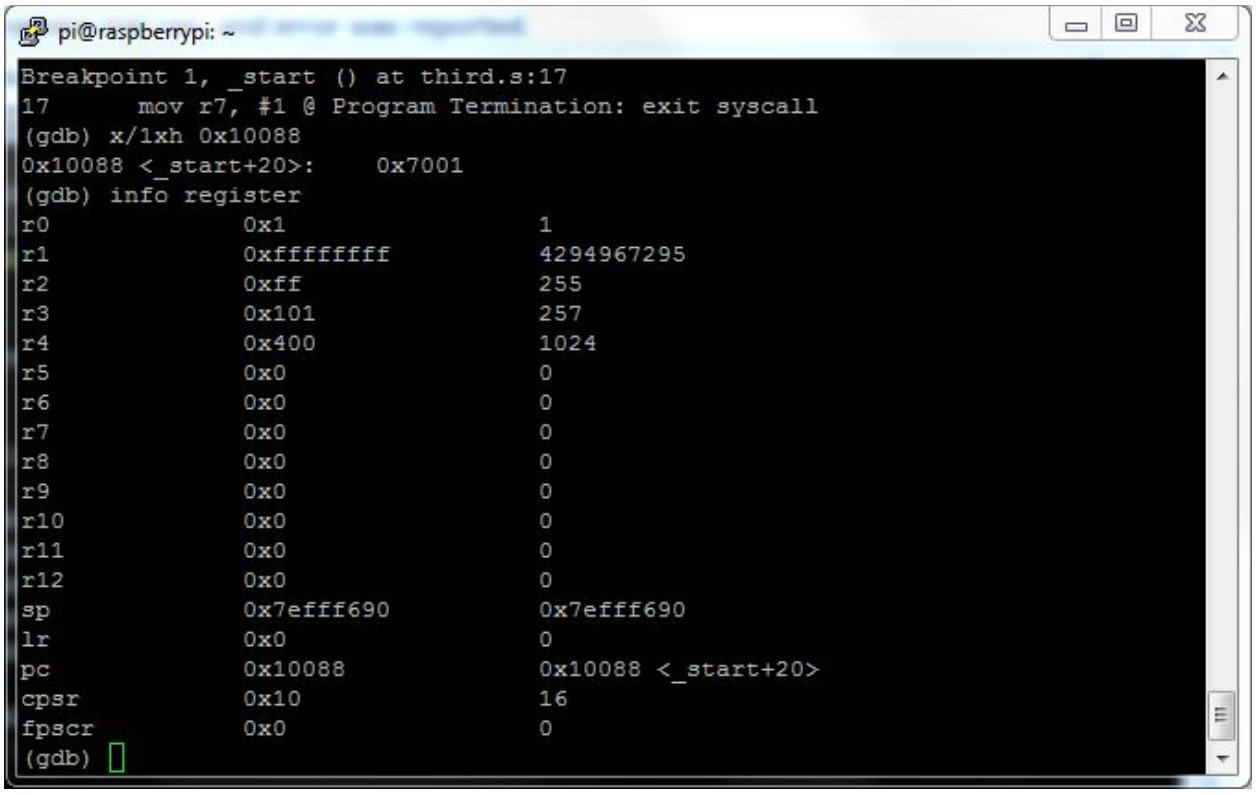
Set a breakpoint at line 17, so the entire program can run.



Here is the memory address with h.

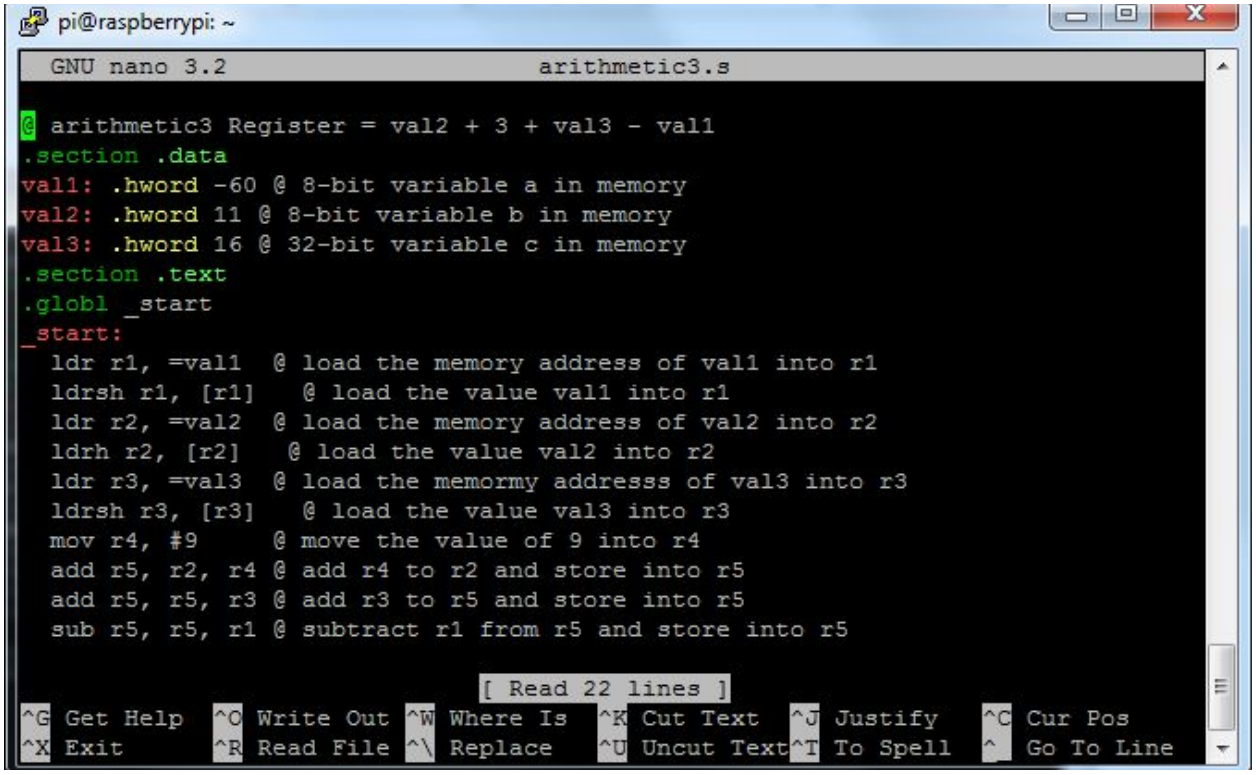


Here is the memory address with sh. The value is recorded but cannot be displayed properly.

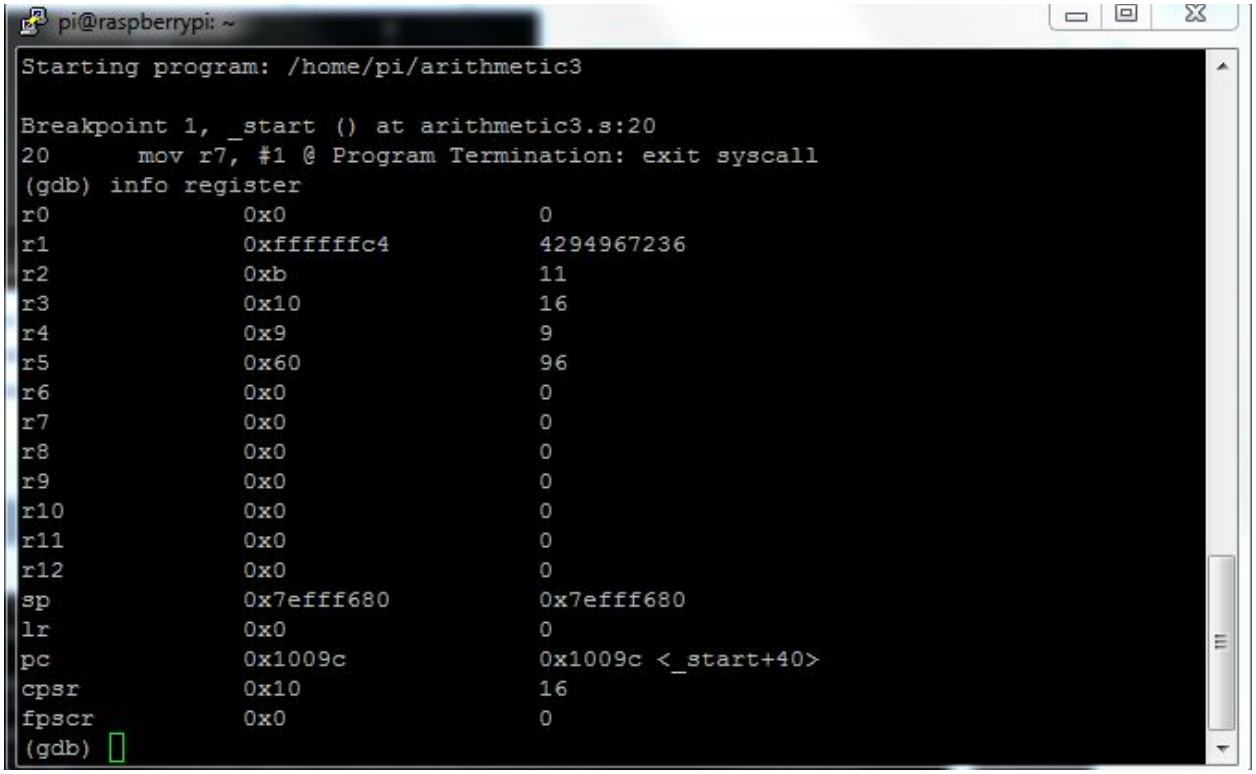


Here is what is in the register to verify the code execute correctly and the registers have the correct values. Negative one is record by 2's complement. If you convert the value in r1, you will see that it is -1.

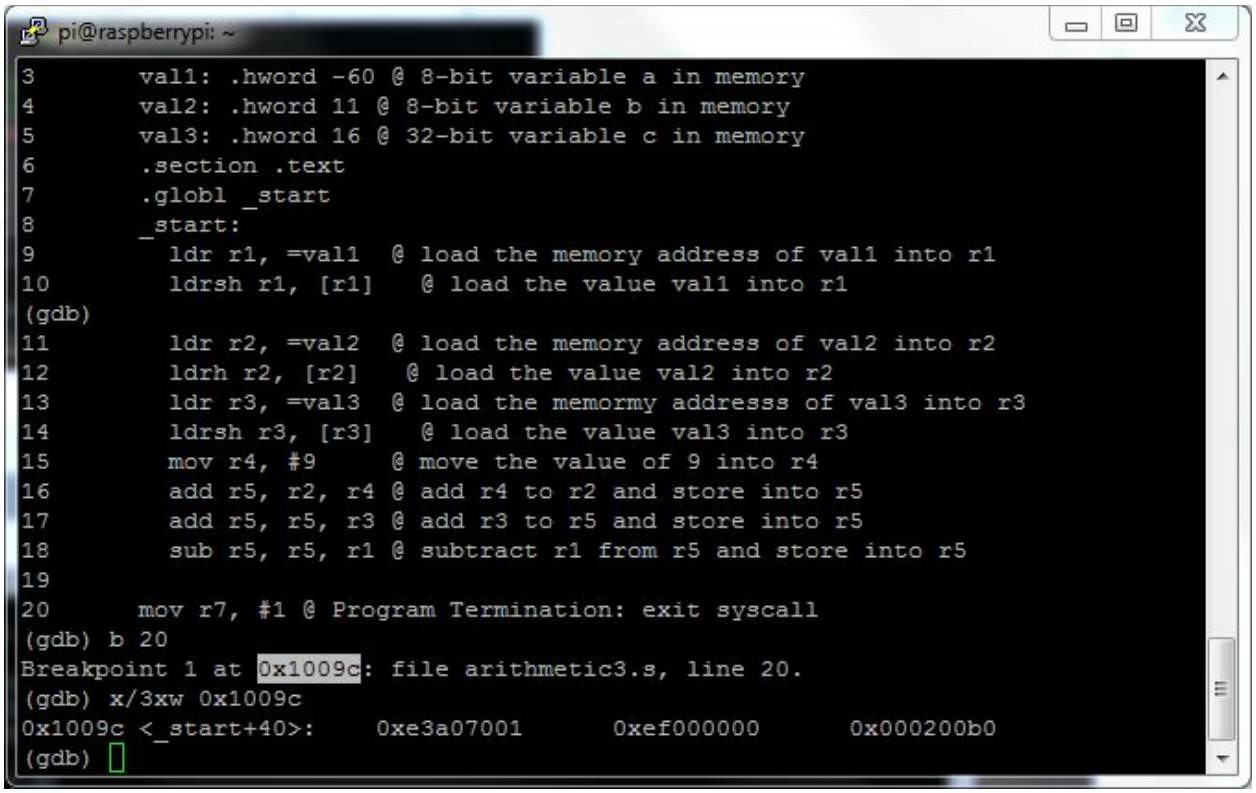
Part B



Here is the code for arithmetic3.



After debugging the program and and setting a breakpoint at line 20, we check the register to verify the correct value.



Here is the memory address.

Appendix

Slack: https://app.slack.com/client/TSWLWS9LK/DTV6YDLSY

Youtube: https://youtu.be/UnylUxrrzO4

Github:https://github.com/Arteenghafourikia/CSC3210-TheCommuters