**Developing Soft and Parallel Programming Skills Using Project Based Learning**

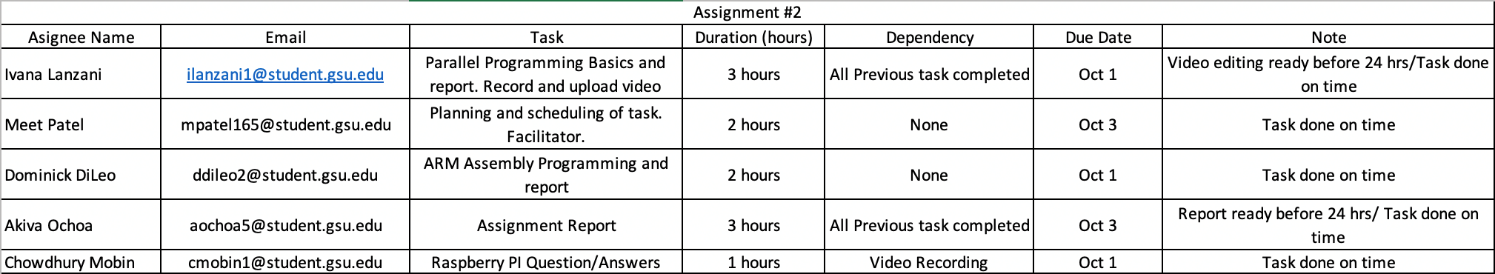
**Fall 2019**

**Group Name: Quinary**

**Members: Meet Patel, Dominick DiLeo, Chodhury Mobin,**

**Ivana Lanzani, Akiva Ochoa**

**Assignment Schedule**

****

**Parallel Programming Skills: Foundation**

1. Raspberry Pi has various components on a single board:

* ARM CPU/RAM,
* Ethernet Port
* Ethernet controller
* Wireless LAN, Bluetooth BLE
* HDMI port
* Display connector
* Camera connector
* 3.5mm Audio Jack
* 4 USB port
* LED’s
* Micro USB Power Port
* SD card slot

2. The Raspberry Pi’s B+ is a single board computer, it has a quad-core CPU. Which means there are four cores in the CPU.

3. Three main differences between X86 (CISC) and ARM Raspberry (RISC):

|  |  |
| --- | --- |
| X86 (CISC) | ARM Raspberry (RISC) |
| It has a large instruction set which allows various complex instructions to access the memory. | It has a much smaller instruction set than the X86, |
| X86 has the Load/Store logic is already built inside more complex inside complex instructions. | ARM uses Load/Store memory model for accessing memory which means only way to access the memory is by using Load/Store instructions. |
| Complex instructions of X86 takes multiple clock cycles per instruction to be executed. | The instructions of ARM can be executed faster, which is approximately one clock cycle per instruction. |

4. The differences between sequential and parallel computation:

|  |  |
| --- | --- |
| Sequential Computing | Parallel Computing |
| Problems are broken into discrete series of instructions which are solved in sequence. | Problems are broken into discrete pieces of work and which can be solved simultaneously. |
| Instructions are executed one after another on a single processor. | Instructions are executed at the same time on multiple processors. |
| Limited compute resources make problem solving slower. | Multiple compute resources such as multiple processor or core make problem solving quicker. |

5. Data parallelism and task parallelism are two general forms of parallelism at the algorithmic level. Data parallelism is a king of parallelism in which same computation is applied to multiple data items. Here, the available parallelism is proportional to the input size, which leads to massive amount of potential parallelism. Global sum is considered to be a data parallel operation.

Task parallelism applies to solutions, where parallelism is organized around the functions to be performed rather than the data. Functional programs exhibit huge amount of task parallelism.

6. A process is the abstraction of a running program. Thread is a lightweight process that allows a single executable to be a decomposed to smaller, independent parts. Processes don’t share memory, but all threads share the common memory of the process they belong to. Single core CPU can only handle one process at a time, to execute more processes at once we need more cores. Threads are scheduled by operating system on separate cores as available.

7. OpenMP or Open Multi Processing is an application programming interface (API). OpenMP is comprised of three components. Which are environment variables, compiler directives and runtime library routines. This API supports Fortran, C and C++ on most platforms, instruction set architectures and operating systems such as Windows, Linus and macOS.

OpenMP pragmas are compiler directives which enables the compiler to generate threaded code. Open MP pragmas are one of the two primary patterns used as program structure implementation strategies which almost every shared memory parallel program have.

8. There are a lot of applications which benefits from multi-core CPU. Here are the four among those applications:

• Multimedia applications (Houdini, Final Cut Pro, Adobe Illustrator etc.)

• Scientific applications (SciLab, MATLAB etc.)

• Compilers (Clang, GCC, javac etc.)

• Web servers (Apache Tomcat, NGINX etc.)

9. Multi-core contains multiple processing units or cores in a single physical processor. Multi-core is supported by most major operating systems like Windows, Linux, and MacOS. The reasons to choose multi-core instead of single-core are the following:

• Multi-core is much faster than single-core and can also multitask.

• Heat problems are also an issue for single-cores, which decreases CPU performance, and sometimes it causes applications to crash.

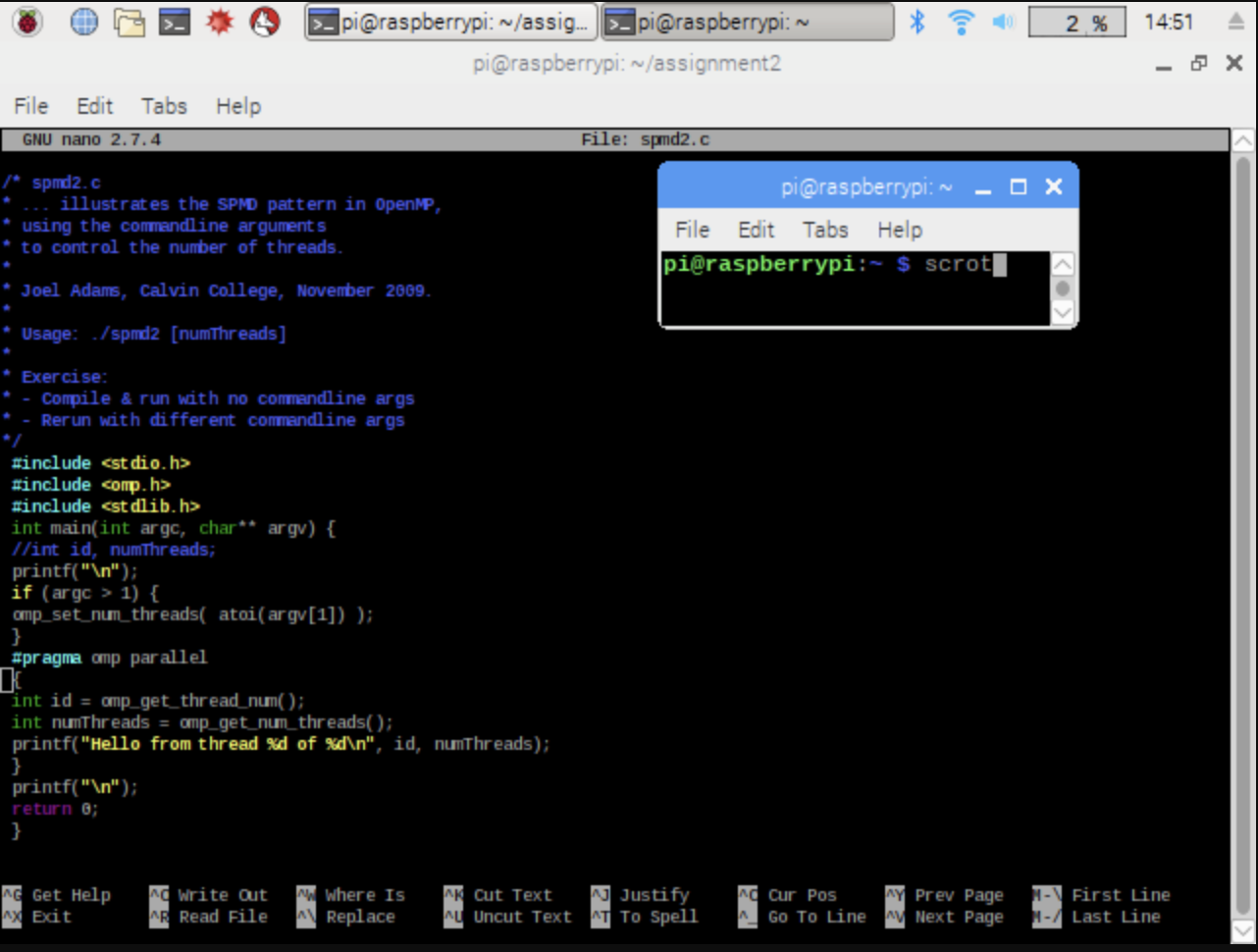
• Many new applications are multithreaded, which runs faster in multi-core CPUs.

• Single-core has higher power consumption than multi-core due to its high clock rates.

**Parallel Programming Skills: Basics**

For this part of the project, we worked with a program to observe how threads behave. A thread is a lightweight process that allows a single process to be decomposed into smaller independent parts. The code used for this assignment uses OpenMP pragmas, a set of compiler directives that allow the compiler to generate a threaded code. The main benefit of threading code is that it enables the execution of different instructions on different data in memory at the same time.

We used the text editor from the terminal to create a file by typing ‘nano spmd2.c’,where we copied the following code:

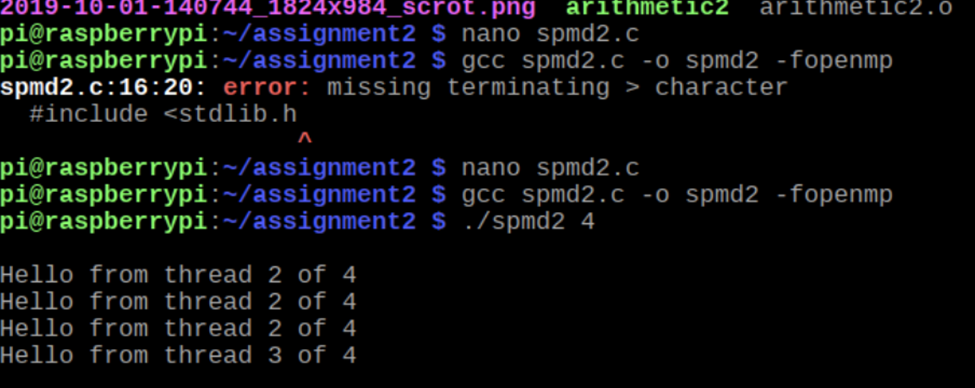


*See Parallel Programming Part 1*

Up to line 10, the code is run on one thread on one core of the Raspberry Pi, while from lines 11 to 15 the code is run on separate threads. Then in lines 15 to 18, the threads are joined back and are ran in thread 0 similar to the start.

After saving and exiting the editor, we compiled our program by using the command ‘gcc spmd2.c -o spmd2 -fopenmp’ and ran the program with ‘*.*/smpd2 4.’. The number at the end of the run command indicates how many threads to fork the program in.

After running the program, we obtained the following output:



*See Parallel Programming Part 2*

The thread id number should not appear listed more than once as each has a unique id. The problem with the code was that all the cores in the Raspberry Pi share the same memory bank and, therefore, the variables’ declaration should occur inside the block that will be forked and run in parallel to avoid having the threads share the variable’s memory location.

To fix this problem, we modified lines 5, 12 and 13 in the following way:

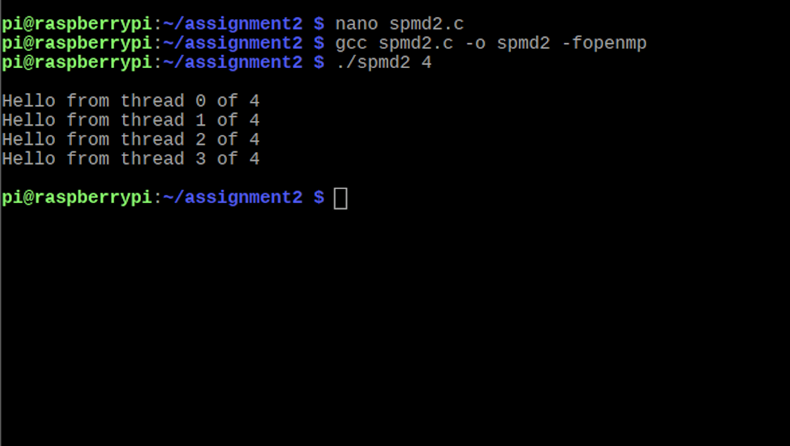
5. //int id, numThreads;

12. int id = omp\_get\_thread\_num();

13. int numThreads = omp\_get\_num\_threads();

This way, we are stating in the code that each thread will have its own copy of the variable’s named ‘id’and ‘numThreads’*.*

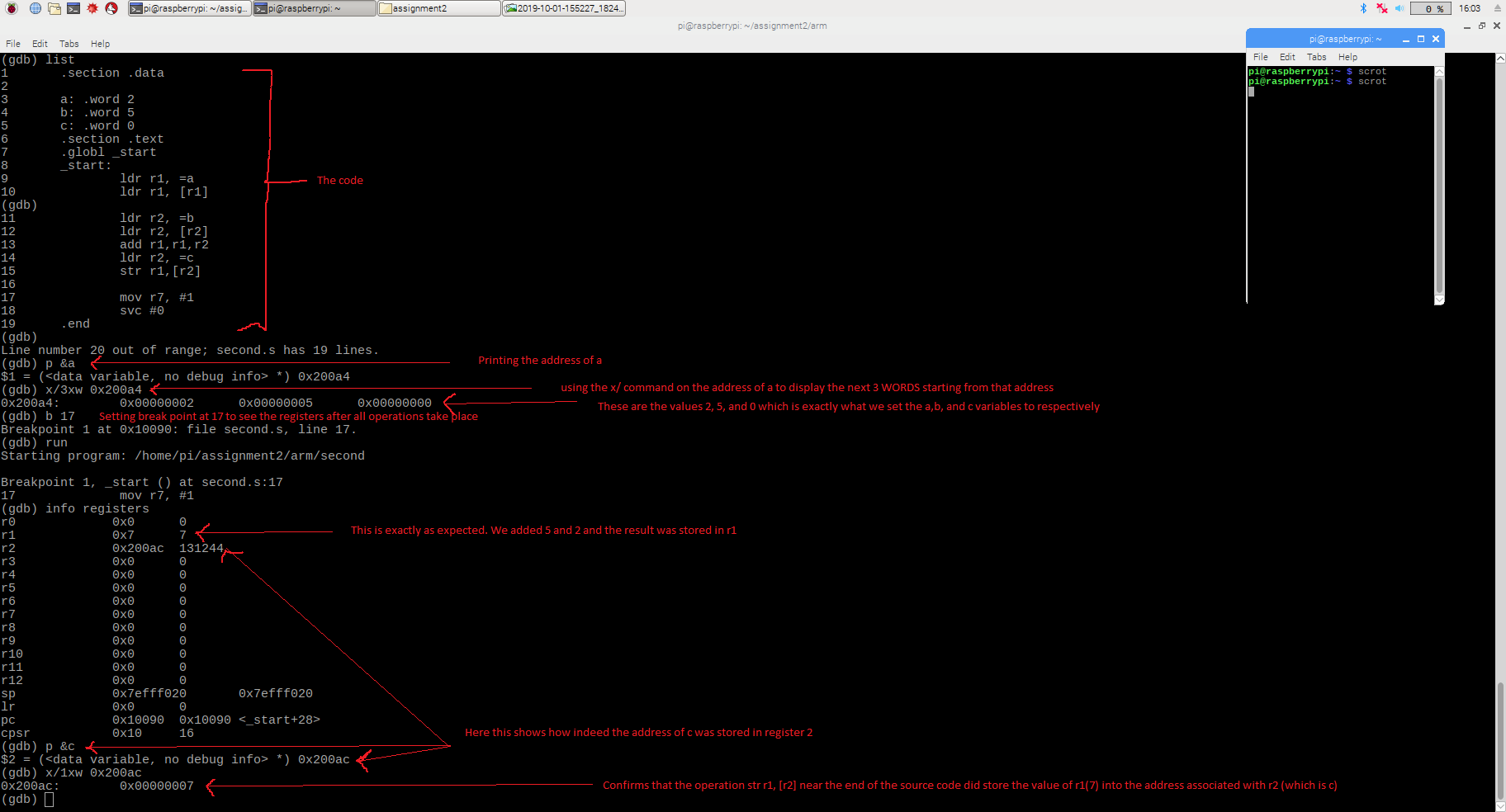
The corrected output was:



*See Parallel Programming Part 2*

**ARM Assembly Programming**

For part 1 we simply typed out all the code supplied in the instructions. We went through the steps given next in order to assemble and view with gdb. We used ‘p &a’ to find the address of the variable a, then used the x/3xw. This showed the first 3 WORDS starting at the address we got from p &a, showing the values 2, 5, and 0. This showed how the variables were stored as expected in memory. We then put a break point at line 17 so that all operations we are concerned with were done and then examined the registers. 7 was stored in register 1, as expected since all that was done was adding 5 and 2 and storing it in r1. R2 contained an address that we confirmed was the address of the c variable with p &c, which is a result of “ldr r2, =c” near the bottom of the source code. The line “str r1, [r2]” was supposed to store r1’s value into the memory address associated with r2, which is the address of the variable c as we confirmed above. To see this, we typed x/1xw 0x200ac (the c address) to see what value was stored at c’s address. The value was 7, which is the same as was in r1.



*See appendix picture (ARM Part 1)*

Part 2 required do the same thing as the mathematical expression but in coding. First we made 3 variables: val1, val2, and val3 and stored the values we were told to in the instructions. We stored the three variables in three different registers, similarly to part 1. Next we added 9 to r2 making 11+9=20. Then we added r3 to r2, making 20+16=36. Next we subtracted r1 from r2, making 36-6 =30. We examined the registers after making a break point and found than indeed register 2 contained 30d.



*See appendix picture (ARM Part 2)*

**Appendix**

Assignment Chart Screenshot:

<https://github.com/Quinary-GSU/Assignment1/blob/master/Screen%20Shot%202019-10-03%20at%202.40.00%20PM.png>

Parallel-Programming Part 1:

<https://github.com/Quinary-GSU/Assignment1/blob/master/A2_ParallelProgramming-nano.png>

Parallel-Programming Part 2:

<https://github.com/Quinary-GSU/Assignment1/blob/master/A2-ParallelProgramming.png>

ARM Part 1:

<https://github.com/Quinary-GSU/Assignment1/blob/master/A2_ARMpart1.png>

ARM Part 2:

<https://github.com/Quinary-GSU/Assignment1/blob/master/A2_ARMpart2.png>

GitHub:

<https://github.com/Quinary-GSU/Assignment1>

Slack:

<https://app.slack.com/client/TN3HREW9X/GNRJ40RT9>

YouTube:

<https://www.youtube.com/watch?v=UD9B_m0UFIQ>