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

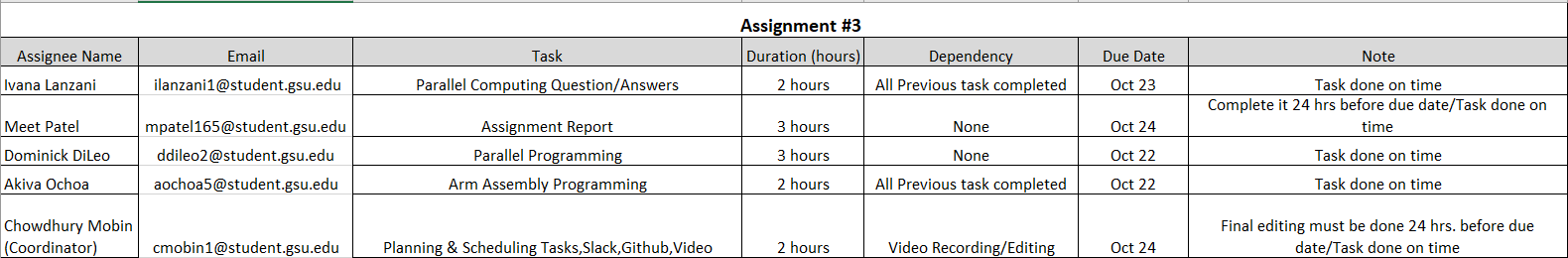
**Fall 2019**

**Group Name: Quinary**

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

**Ivana Lanzani, Akiva Ochoa**

**Assignment Schedule**

****

**Parallel Programming Skills**

1. Task: a task is a program-like set of instructions that is executed by a processor. In the case of parallel programming, multiple tasks are run on multiple processors simultaneously.

Pipelining: it consists in dividing a task into steps that will be executed by different processor units, with inputs streaming through those steps.

Shared Memory: when referring to hardware, shared memory refers to when a computer’s processors all have access to a common physical memory. However, from a programming point of view, it refers to a model where parallel tasks all can access the same logical memory locations regardless of where the physical memory is located.

Communications: it’s the data exchanges of parallel tasks that can be accomplished through different ways like through a shared memory bus or a network.

Synchronization: the coordination of parallel tasks in real time. It is usually implemented by choosing a point within the application that a task cannot go past another task get to the same point or a logically equivalent point.

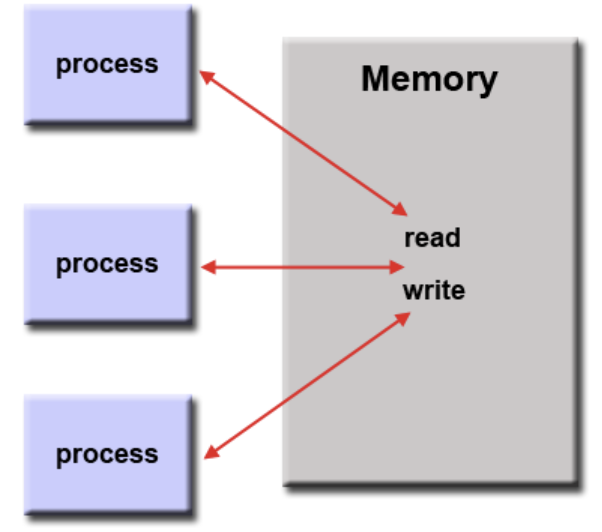
1. Parallel computers can be classified as SIMD (Single Instruction, Multiple Data), MISD (Multiple Instruction, Single Data) and MIMD (Multiple Instruction, Multiple data), according to Flynn’s Taxonomy.

In SIMD parallel computers, the same instruction executes by all processing units at any given clock cycle, but each processing unit can use a different data element. It is best suited for problems that follow a path of regularity because of its characteristics.

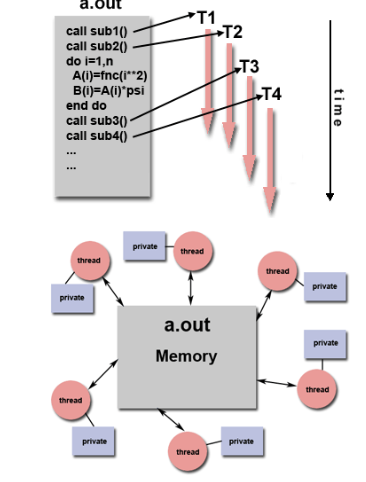
In the case of MISD, it is also a type of parallel computer, but the processing units operate via separate instruction streams. It is single data because a single stream of data is fed into multiple processing units. This type of computer has very few applications and it is not in use.

Lastly, in MIMD computers, the processing units execute different instruction streams with different data. Execution can be both synchronous or asynchronous, deterministic or not deterministic. Supercomputers fall in this classification.

1. The most commonly used parallel programming models are Shared Memory, Threads, Distributed Memory/Message Passing, Data Parallel, Hybrid, Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD).
2. The types of parallel computer memory architectures are: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA).UMA architectures have identical processors, each with equal access and access time to memory. It is sometimes called Cache Coherent UMA (CC-UMA), because if one processor updates a location in the shared memory, the other processors are aware of the update. It is commonly represented by Symmetric Multiprocessor (SMP) machines.NUMA architectures are usually made by two or more SMPs where one can access directly access the memory of another SMP, and processors do not have equal access time to all memories. They can also be cache-coherent (CC-NUMA). OpenMP uses UMA Computer architectures because the architecture of those computers are more suitable to run parallel processes. In UMA, single, multiple and crossbar busses are used and NUMA uses hierarchical and tree type of busses and network connection. Also, the memory accessing time for each processor is the same in UMA while, in NUMA, the memory accessing time changes as the distance of memory from the processor changes.
3. In the Shared Memory Model, all processes share a common address space that they can read and write asynchronously. It’s a very simple parallel programming model but it is harder to control and manage data locality.



In the Threads Model the process is divided into subroutines called threads. When the main program runs, it collects the system and user resources and then creates threads to execute tasks that can run parallely. The threads share the memory with the main program but each thread also has local data.



1. Parallel Programming is a form of programming that uses multiple processors or processor’s cores to divide processes into smaller subroutines that execute simultaneously, offering the same result but in less time and more efficiently.
2. A system on chip (SoC) integrates all the components of a CPU into a single silicon chip and can function without the addition of any other chips. All of the Raspberry Pi model use SoC.
3. The advantages of using a system on chip are that it contains more functionality for its size when compared to a CPU, which allows to build highly functional yet small devices like smartphones or tablets. SoC also uses less power than a CPU to run and because it has less added components, it is also cheaper to build a computer using SoC.

**Parallel Programming Code / Report**

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

Task: A program or set of instructions that does a certain task in a program like way. When you split up a program with parallelism, the program is broken into separate tasks.

Pipelining: A way to try to keep as many parts of a single processing unit busy. When some instruction is being used by some parts of the processor, but not others, another instruction that requires the idle parts will start to process and use those parts.

Shared Memory: When all processors, and the parallel tasks on them access the same memory and can read/write it.

Communications: The way in which the parallel tasks exchange information and data to come to the conclusion of the program

Synchronization: When there are parallel tasks, some tasks finish before others but the next step may require all tasks to be done first. Therefore some tasks need to wait and when all are done they are “synchronized” to go onto the next step.

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

Single Instruction, Single Data(SISD): A computer that is serial(goes step by step without parallelism). There is only one instruction and data stream used at once. This is the way the oldest computers ran.

Single Instruction, Multiple Data(SIMD): Every processing unit does the same instruction, but can act on different data. This is good for tasks where it will be very repetitive. Graphical processing units often make use of this

Multiple Instruction, Single Data(MISD): One data stream will be acted upon by different types of instructions. Very few examples of this, but one might be trying to decipher cryptography. The cryptic message would be the data, and the instructions would be different deciphering techniques.

Multiple Instruction, Multiple Data(MIMD): The processing units can have different types of instructions operating on different data streams. In modern day computing, this is the most popular.

**What are the Parallel Programming Models**

Shared Memory(without threads), Threads, Distributed Memory/Message Passing, Data Parallel, Hybrid, Single program multiple Data(SPMD), Multiple Program Multiple Data(MPMD)

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

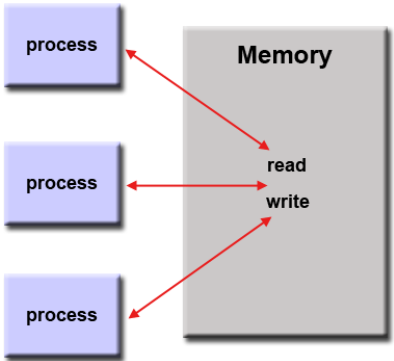
Uniform Memory Access(UMA): Identical processors all having equal access to memory. Commonly represented by Symmetric Multiprocessor (SMP)

Non uniform Memory access(NUMA): Linking of multiple SMPs. Not all processors have equal access time to all memories.

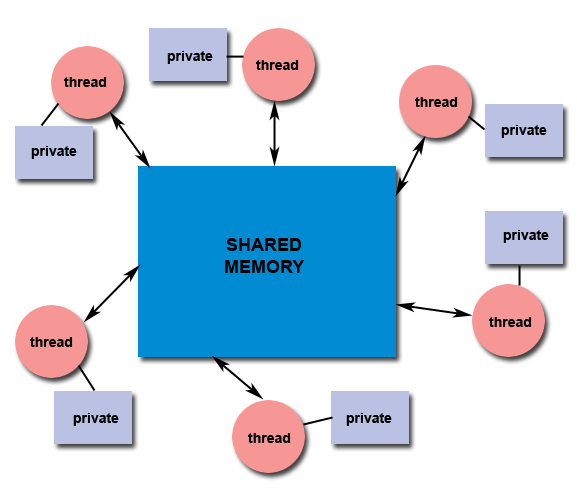
Openmp is using UMA. The program is broken into threads put onto multiple processors. The processors should all have equal access to the memory in order to synchronize well to finish the program. UMA has equal access and access times between the processors.

**Compare Shared Memory Model with Threads Model? (in your own words and show pictures**

Shared Memory(without threads): Processes all use the same address space. It is a simple parallel programming model, but it becomes difficult to manage the data since each process has little restriction.



Shared Memory(with threads): The main program has a memory space it uses, but this program is broken in to smaller threads. These threads each have local data they can use while still being able to access the more global data of the main process. This gets rid of the problem of processes messing with each other and disrupting the program like in shared memory without threads.



**What is Parallel Programming**

Having multiple processes going on at the same time, sometimes related processes and sometimes unrelated processes.

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

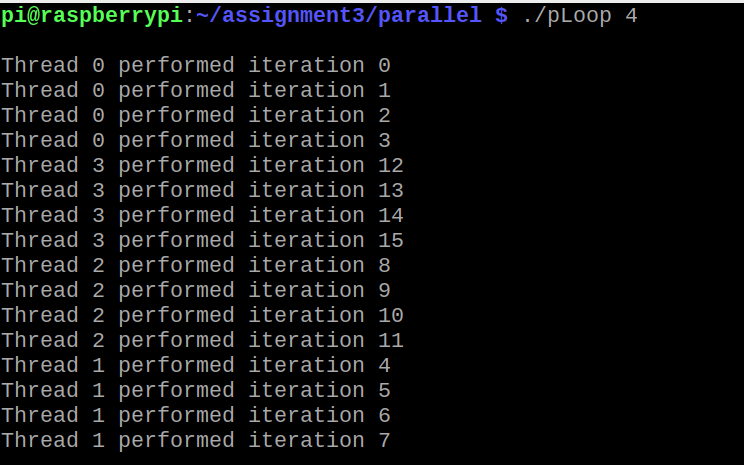
A chip that combines the cpu, the gpu, memory, a usb controller, power management circuits, and wireless radios all into one. This is opposed to the CPU which is normally more separated from everything else on most computers. The Raspberry PI uses a Broadcom System On Chip(SOC)

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

The SOC conserves a lot of space and uses less power. This would allow for entire computers that are normally large to become easy to transport. Since there are less physical separate chips, the computer will be cheaper as well.

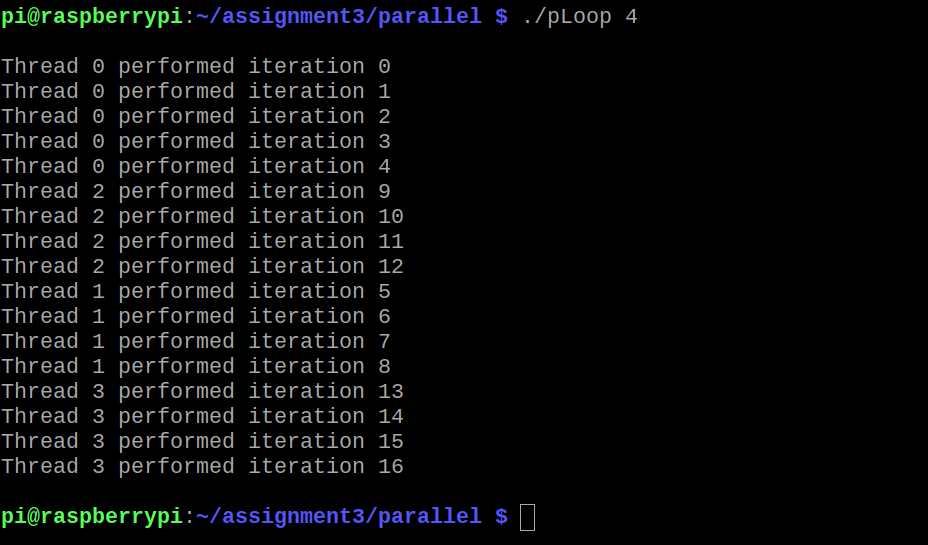
**Parallel Programming Basics**

**Questions and explanations related to the first program: parallelLoopEqualChunks.c**



Here the iterations of the for loop were split between the threads, as opposed to each thread doing 16 iterations like it would without the for statement(in #pragma omp parallel for). The behavior is what is expected based on the diagram we were supposed to compare to in the instructions.

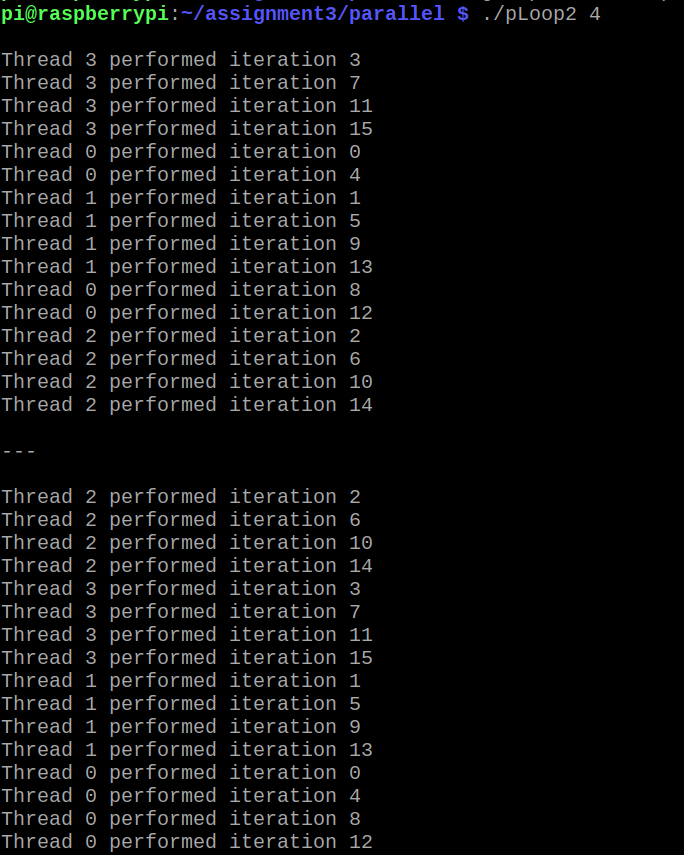
**Change number of iterations to something not evenly divisible by the number of threads:**

****

I changed the number of iterations to 17 which is not divisible by 4. This time 0 got the additional iteration. I tried with other number such as 18 and 19. The pattern seemed to be this:

Any multiple of 4 would result in every thread getting an equal number of iterations. When it was not a multiple of 4, first find (Number of iterations)%(number of threads). Assuming threads = 4, then this will result in either 0,1,2, or 3. 1 means 1 iteration past a multiple of 4, and this is given to thread 0. 2 means two past, so thread 0 gets an iteration and thread 1 gets the other iteration. 3 means thread 0,1, and 2 will get an iteration.

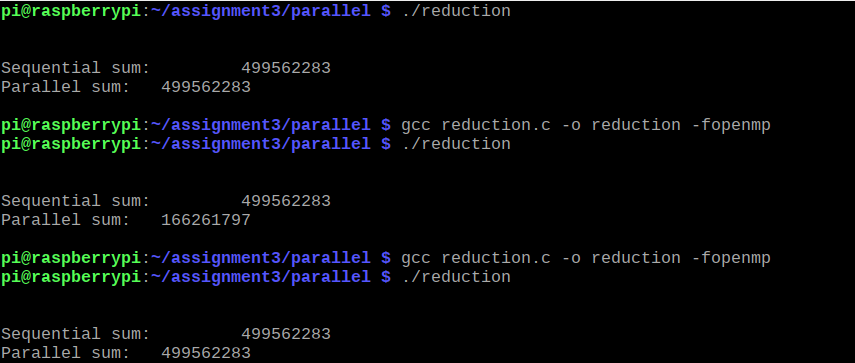
**Questions and explanations related to the second program: parallelLoopEqualChunksOf1.c**



Here is the output when you also uncomment the commented code that does not make use of the schedule(static, 1) clause. Like the comment message at the top of the file says, both methods give the same result.

The difference between this program and the previous program is which threads get which iterations. In the previous program, each thread got consecutive iterations. In this program the iterations were given one by one sequentially to the next thread up(and back to thread 0 when the last iteration was given to thread 3). The schedule(static, 1) means that the chunks are given one at a time in size 1 iteration in the way described above. If schedule(static,2) were written, the chunk size would be 2, so thread 0 would get iterations 0,1, thread 1 would get 2,3, thread 2 would get 4,5 and so on. The number could be changed to 4 as well which would actually give the same output as the first program.

**Questions and explanations related to the third program: reduction.c**

****

The program initialized an array of size 1,000,000 with 1,000,000 random numbers through the initialize function. Then the sequentialSum and parallelSum functions are called.

The first run of the code is when the entire #pragma line is commented out in the parallelSum function. This means that the parallelSum function does the same thing as the sequentialSum function.

The next run is when the #pragma line is uncommented up to the reduction clause in the parallelSum function. The output shows different numbers for each, meaning the parallel sum is produced an incorrect number. The reason this happened is because all the threads are trying to access the sum variable. Some threads may try to access sum at the same time creating what is called a “data race”. This causes unforeseen effects and thus we get a wrong number from the parallelSum function.

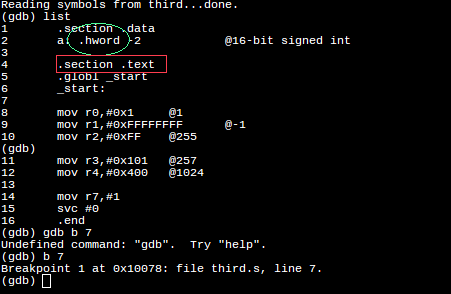
The next run is when the reduction clause is uncommented too. This solves the issue talked about above. Now each thread gets its own local copy of sum, getting rid of the problem of multiple threads trying to access a memory location at the same time. After each thread has gone through its partition of the array and summed them, all the sums are brought together and stored in the sum variable. As we can see now the parallelSum function produces the same number as the sequentialSum function.

**ARM Programming Report**

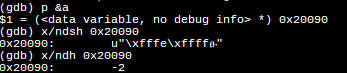
For assignment 3 we explored how the ARM handles storing signed and unsigned integers in the memory and its registers. Along with declaring signed variables. Creating a new file with ‘nano third.s’ I replicated the code for third.s, declaring the variable a and moving various unsigned values and a signed value to separate registers. However, upon assembling the following errors occur:



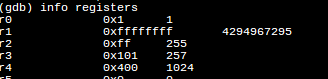
I suspected the reason for the errors is the lack of white space for .section .data/.text and the syntax for declaring signed halfword values. Placing white space between .section and its counterparts did alleviate the first and third errors. But for halfword I tested .shword and to see it functions similar to the x86 style of declaring signed variables. While it didn’t assemble either, I decided to assemble using .hword instead, which did assemble.



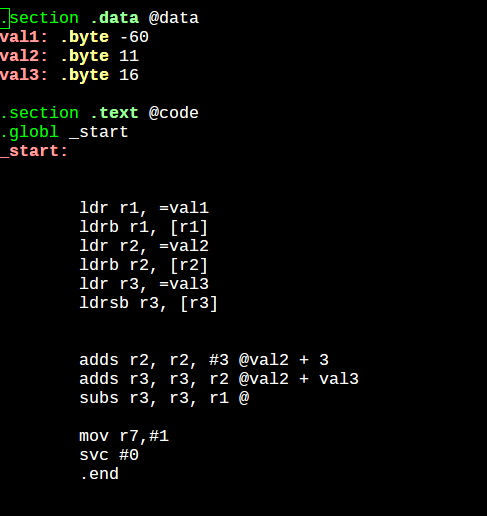
After running third in the debugger and checking the memory where variable “a” was stored, I was able to conclude that .hword has implied signed storage for signed variables.



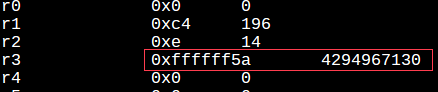
I also observed that registers store similarly to x86, thus interpreting all hex values as unsigned.



For the arithmetic3 assignment I was tasked to create three variables, 2 unsigned and 1 signed, and preform the following function: val2 + 3 + val3 – val1 and store in an arbitrary register. Formatted like this:

****

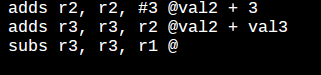
Checking the register during the arithmetic revealed that when the ARM 2’s complements a value to represent a negative under the SUB operand, it extends 1 to the most significant bit after the value. Thus, despite the true value being signed +90(5A), ARM still adds the leading extension.



Another thing that occurred was the lack of bits in the cpsr register. Without it it wouldn’t be possible to check the flags.



We had looked up how to have the bits representing the flags and learned that an ‘s’ must be appended to the instruction operator for each operation you want to observe flag changes with. With the flag order being NF ZF CF OF



Since 8h in binary is 1000, we were able to observe that the negative flag was set from the most significant bit from FFFFFF5A being 1.



**Appendix**

Assignment Chart Screenshot

<https://github.com/Quinary-GSU/Assignment1/blob/master/Assignment%20%233%20Task%20Table.xlsx>

Parallel Programming Part 1

<https://github.com/Quinary-GSU/Assignment1/blob/master/parallel%20p1.PNG>

Parallel Programming Part 2

<https://github.com/Quinary-GSU/Assignment1/blob/master/parallel%20p2.PNG>

Parallel Programming Part 3

<https://github.com/Quinary-GSU/Assignment1/blob/master/parallel%20p3.PNG>

ARM Part 1

<https://github.com/Quinary-GSU/Assignment1/blob/master/A3_ARM_P1_1.png>

ARM Part 2

<https://github.com/Quinary-GSU/Assignment1/blob/master/A3_ARM_P1_2.png>

ARM Part 3

<https://github.com/Quinary-GSU/Assignment1/blob/master/A3_ARM_P2.png>

ARM Code Part 1

<https://github.com/Quinary-GSU/Assignment1/blob/master/A3_Code_P1.png>

ARM Code Part 2

<https://github.com/Quinary-GSU/Assignment1/blob/master/A3_Code_P2.png>

Github

<https://github.com/Quinary-GSU/Assignment1/projects/1>

Slack

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

Youtube Channel

<https://www.youtube.com/watch?v=nDXKhTtCRys&feature=youtu.be>