Fall 2018

Developing Soft and Parallel Programming Skills Using Project-Based Learning

Mad Scientists

Samuel Fekadu
Jason Moon
Pavel Beliaev
Christopher Wilson
Pranthi Cavuturu

Planning and Scheduling:

Assignee Name	Email	Task	Duration (hours)	Dependency	Due Date	Note	On Time?
Pranthi Cavuturu (coordinator)	pcavuturu1@student.gsu.edu	Planning and Scheduling	1 hour	N/A	10/9/18	Task 1: Divide tasks evenly. Remind requirements under "Note". Send updated version to Christopher on 10/25/18.	Y
		Group Meeting: Finish Task 3, Parallel Programming Basics	1 hour	Everyone must be present.	10/15/18	This way, everyone can see what's happening with the Pi.	Y
		Group Meeting: Record video	1 hour	Everyone must be present.	10/22/18	Prepare for video in advance. Follow A3, Task 4 directions.	Y
		Task 3 Foundation questions	2 hours	Read Introduction to Parallel Computing	10/22/18	Send to Jason to verify answers.	Y

		Verify completion of all tasks with each group membe	1 hour	Group members' status of task completio n.	10/25 /18	All tasks except formatting report should be complete by this point.	Y
Pavel Beliaev	pbeliaev1@ student. gsu.edu	Group Meeting: Finish Task 3, Parallel Programming Basics	1 hour	Everyone must be present.	10/15 /18	This way, everyone can see what's happening with the Pi	Y
		Group Meeting: Record video	1 hour	Everyone must be present.	10/22 /18	Prepare for video in advance. Follow A3, Task 4 directions.	Y
		Update GitHub	30 mins	N/A	10/18 /18	Send screenshot to Christopher through slack.	Y
		Optional: Verify Task 3 Foundation Questions	1 hour	Pranthi's completio n of Task 3 Foundatio n questions.	10/25 /18	Add/Edit if necessary.	Y
Jonghan Moon (Jason)	jmoon22@s tudent.gsu.e du	Group Meeting: Finish Task 3, Parallel Programming Basics	1 hour	Everyone must be present.	10/15 /18	This way, everyone can see what's happening with the Pi.	Y

		Group Meeting: Record video	1 hour	Everyone must be present.	10/22/18	Prepare for video in advance. Follow A3, Task 4 directions.	Y
		Verify Task 3 Foundation questions.	1 hour	Pranthi's completion of Task 3 Foundation questions.	10/25/18	Add/Edit if necessary.	Y
Samuel Fekadu	sfekadu1@student.gsu.edu	Group Meeting: Finish Task 3, Parallel Programming Basics	1 hour	Everyone must be present	10/15/18	This way, everyone can see what's happening with the Pi	Y
		Recording group presentation video	1 hour	Everyone must be present	10/22/18	Prepare for video in advance. Follow A3, Task 4 directions.	Y
		Edit and upload video	2 hour	Video must be recorded.	10/25/18	Upload to Youtube as usual.	Y
		Optional: Verify Task 3 Foundation Questions	1 hour	Pranthi's completion of Task 3 Foundation questions.	10/25/18	Add/Edit if necessary.	Y
Christopher Wilson	cwilson94@student.gsu.edu	Group Meeting: Finish Task 3, Parallel Programming Basics	1 hour	Everyone must be present.	10/15/18	This way, everyone can see what's happening with the Pi.	Y

		Group Meeting: Record video	1 hour	Everyone must be present.	10/22/18	Prepare for video in advance. Follow A3, Task 4 directions.	Y
		Format report	1 hour	Pranthi's completion of Task 3 Foundation questions.	10/25/18	Follow A3, Task 5 directions. Links to slack, youtube, etc can be found in previous assignments.	Y
		Optional: Verify Task 3 Foundation Questions	1 hour	Pranthi's completion of Task 3 Foundation questions.	10/25/18	Add/Edit if necessary.	Y

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

- Task: a part of the instructions in a solution to a computational question that are to be executed by the processor. In parallel programming, multiple processors are used for tasks.
- Pipelining: Pipelining allows an increased number of instructions to be performed in a given period of time. It allows the architecture to fetch instructions while the processor is performing arithmetic operations. Fetching instructions is continuous. So, it acts as an assembly line. It does not wait for the operation in the instruction to be performed.
- Shared Memory: Memory that can be accessed simultaneously by multiple processes regardless of where the physical memory exists.
- Communications: the event of any kind of data exchange among processes

- Synchronization: At least one task waits for others to be at the same standard of progress or logical equivalence. This causes an increase in execution time.

(8p) Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.

Classified by input stream and data stream and each with two possible states- single or multiple:

- Single instruction, single data: the oldest type of computer where only one instruction is being acted on by the CPU and only data stream is being used as input during a given clock cycle
- Single instruction, multiple data: All processing units execute the same instruction, but each on a different data element.
- Multiple Instruction, single data: A single data stream is fed into multiple processing units in which each unit operates on the data independently.
- Multiple instruction, multiple data: Each processor may be executing a different instruction and working with a different data stream

(7p) What are the Parallel Programming Models?

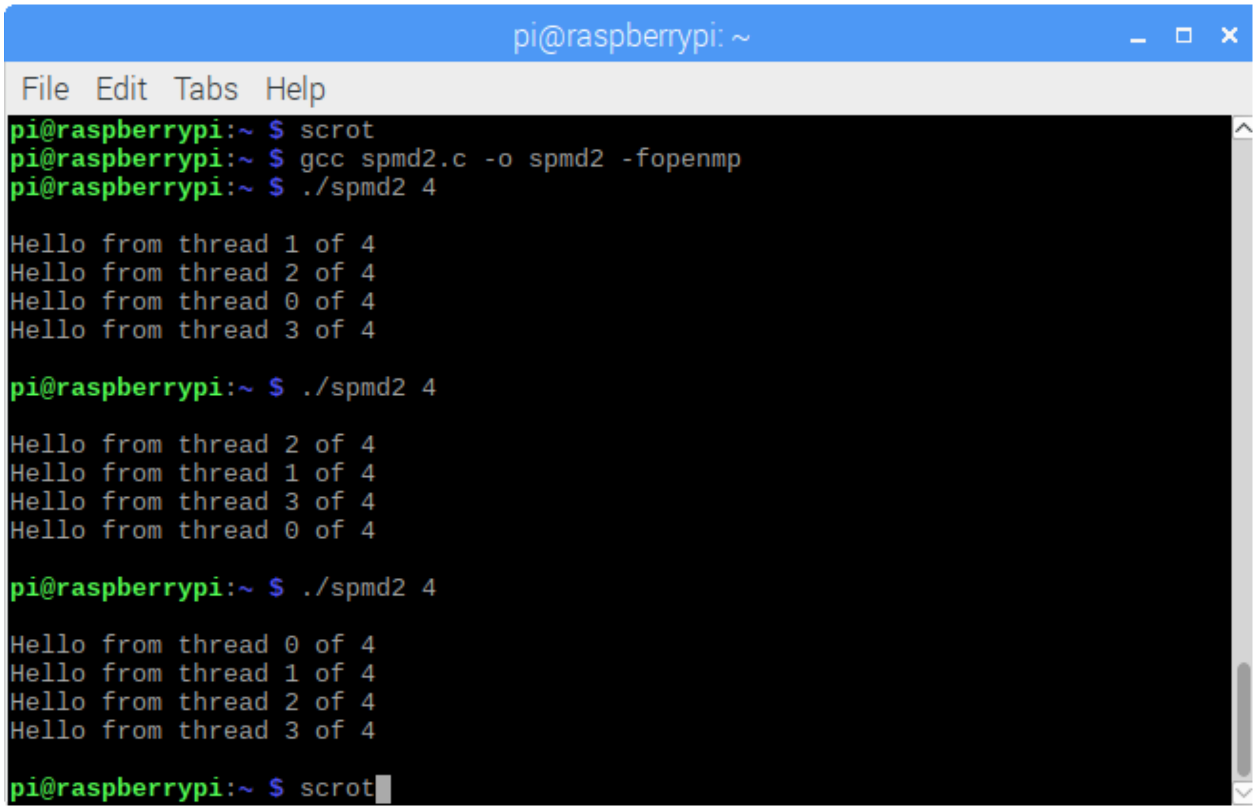
- Shared memory
- Threads
- Distributed Memory/Message Passing
- Data Parallel
- Hybrid
- Simple Program Multiple Data
- Multiple Program Multiple Data

(12p) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?

- Uniform Memory Access: All processors share the physical memory uniformly. Processors are identical. They have equal access and equal access times to memory. All processors know of a memory update.
- Non Uniform Memory Access: Memory access time depends on the memory location relative to the processor. Not all processors have equal access time to all memories. Made by linking two or more Symmetric Multiprocessors. Memory access across a link is slower.
- OpenMP is flexible and can work with different kinds of memory layouts. OpenMP is able to use both uniform and non-uniform memory access architectures.

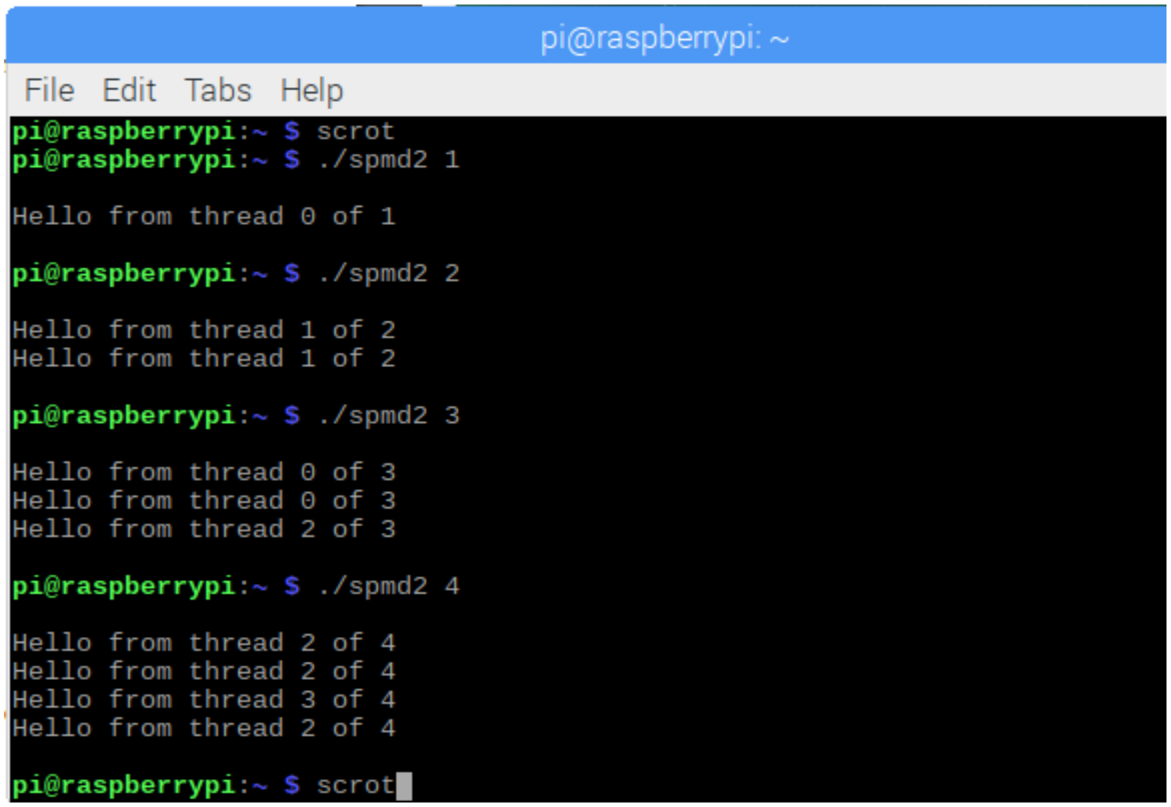
(10p) Compare Shared Memory Model with Threads Model? (in your own words and show pictures)

- In the shared memory model, processes/tasks share common address space in which they read to and write to asynchronously. The threads model is a type of shared memory programming in which heavyweight process has multiple execution paths at the same time
- In Assignment 2, we saw an example of the threading model, since the iteration task was divided into four threads.



```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 2 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from thread 0 of 4

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

pi@raspberrypi:~ $ scrot
```

- The following seems like an example of shared memory modeling in which a variable was declared outside of the block that will be forked and run in parallel on separate threads, all threads share that variable’s memory location. In the shared memory model, processes/tasks share common address space in which they read to and write to asynchronously.



```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./spmd2 1

Hello from thread 0 of 1

pi@raspberrypi:~ $ ./spmd2 2

Hello from thread 1 of 2
Hello from thread 1 of 2

pi@raspberrypi:~ $ ./spmd2 3

Hello from thread 0 of 3
Hello from thread 0 of 3
Hello from thread 2 of 3

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ scrot
```

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

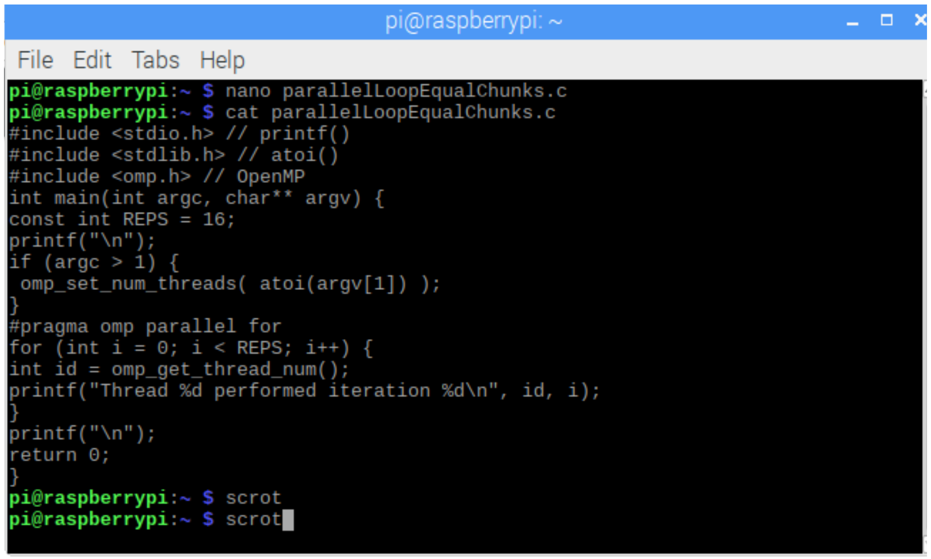
- In parallel programming, the execution of processes are carried out simultaneously. A large problem is divided into smaller problems which can each be solved at the same time, thus the concept of multicore. Each “small problem” can be solved on each core.

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

- Where as a CPU must be aided with other silicon chips to function fully as a computer, a system on chip integrates the external components of a CPU into a single silicon chip. An SoC contains a graphics processor, memory, USB controller, power management circuits, and wireless radios. It is possible to build a computer with a just an SoC. Raspberry PI uses SoC.

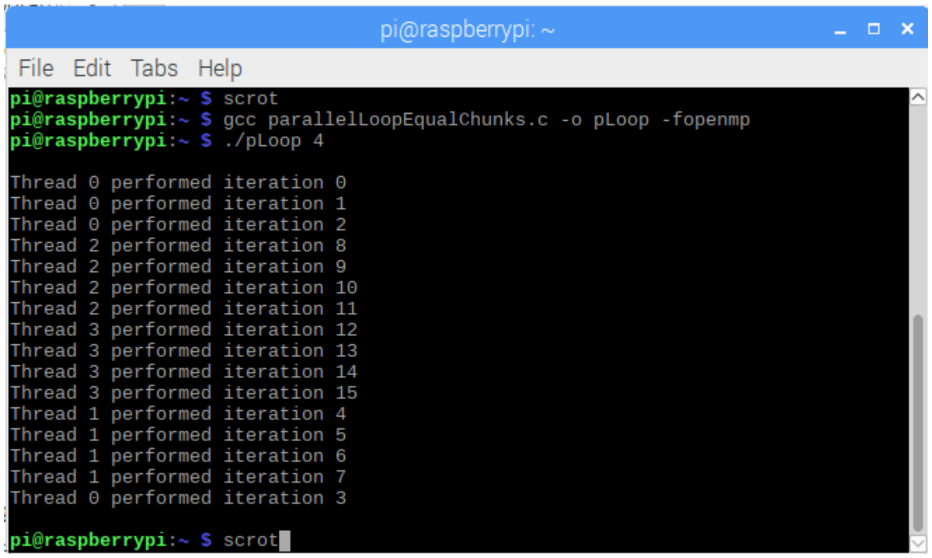
(5p) Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.

- The SoC is only a little bigger in size than the CPU but contains much more. SoCs allow computers in smartphones and tablets. SoCs allow plenty of space for batteries.
- Power consumption is in data and bus address labeling. Since all components are on the same chip and internally connected, and their size is also very small, an SoC uses a considerably less amount of power.
- Due to the fact that very few physical chips are needed, it is much cheaper to build a computer using an SoC.



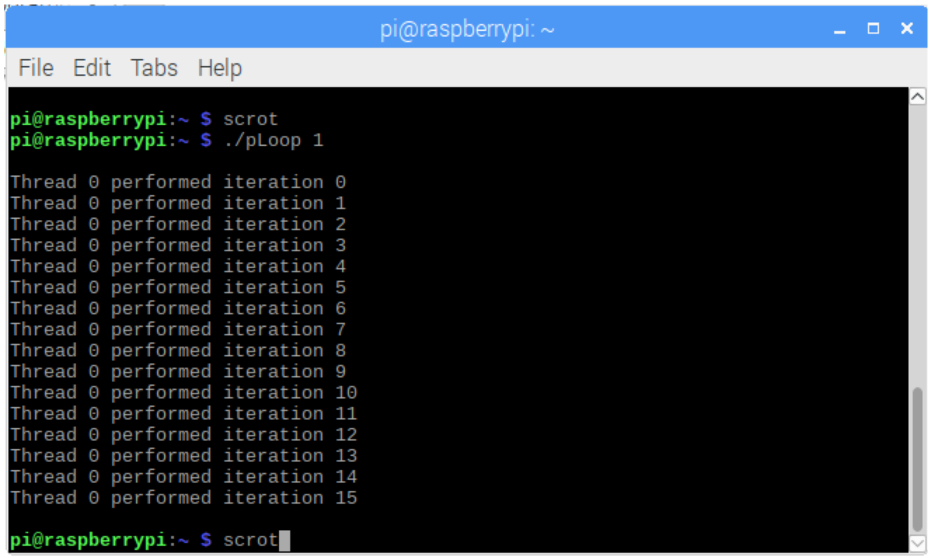
```
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ cat parallelLoopEqualChunks.c
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP
int main(int argc, char** argv) {
const int REPS = 16;
printf("\n");
if (argc > 1) {
 omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n", id, i);
}
printf("\n");
return 0;
}
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
```

This is the code copied and pasted for the first observation.



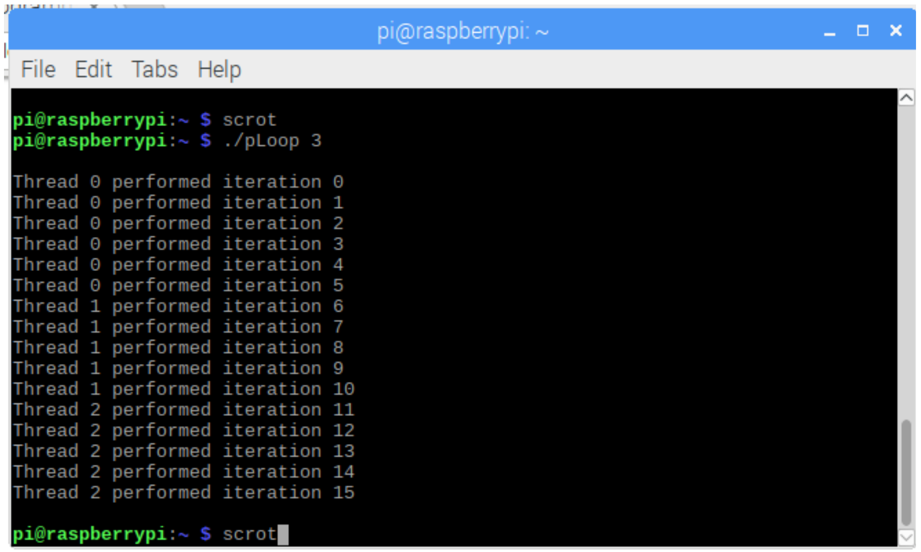
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

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

pi@raspberrypi:~ $ scrot
```

The number 4 represents how many forks to thread to finish the iteration task. Here we used 4, so we have thread 0, thread 1, thread 2, and thread 3. This divided the iterations into equal sized chunks. We were familiar with this because of the last parallel programming task assignment and only now, we applied it to iterations.



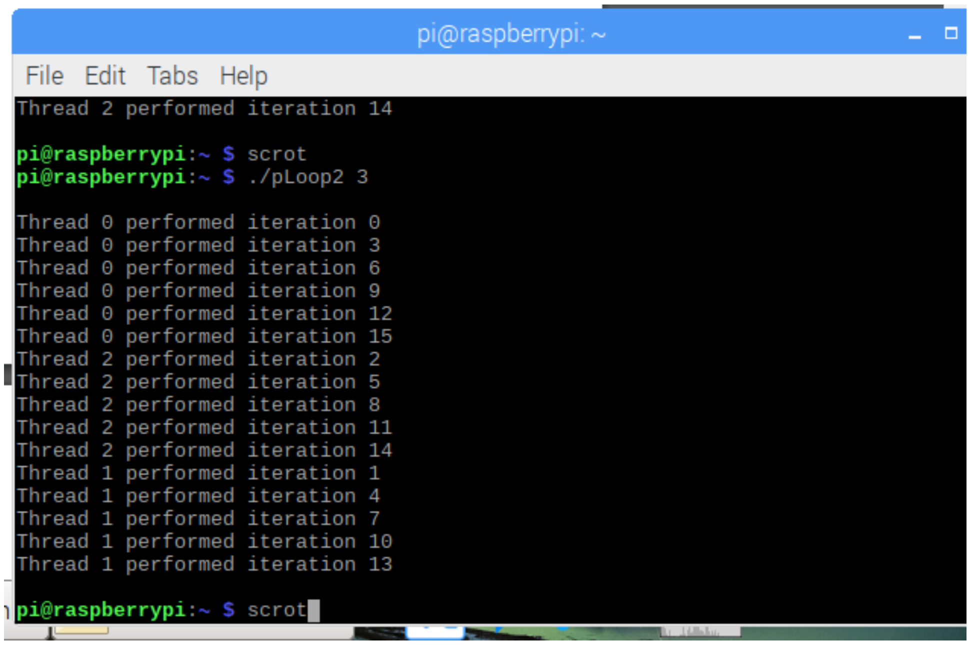
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop 1

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

pi@raspberrypi:~ $ scrot
```

Since we changed the number to 1 here, the iteration task is completed with one core, thread 0. Thread 0 performed all 15 iterations. 16 can be divided by 1 equally and there no distribution in iterations. Only one fork was threaded.



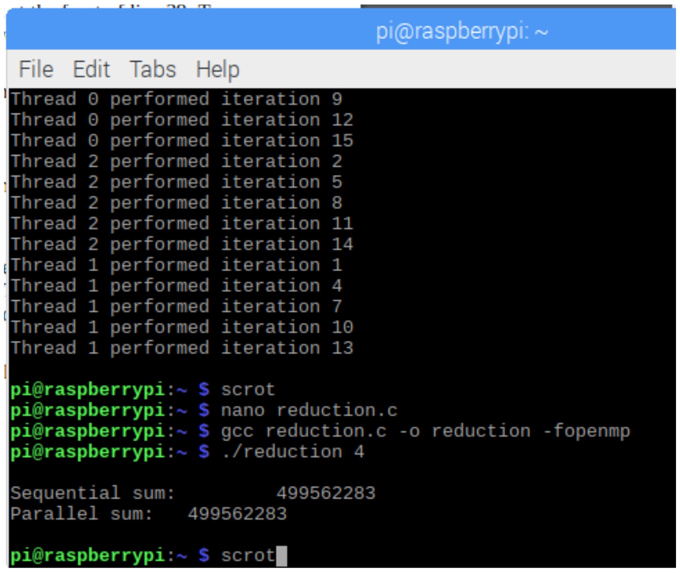
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop 3

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

pi@raspberrypi:~ $ scrot
```

The same step was repeated. This time we used 3 cores by specifying the number 3. So three threads were used, thread 0, thread 1 and thread 2. Three threads are used, so thread 0 performed 6 iterations while 1 and 2 performed 5 iterations.



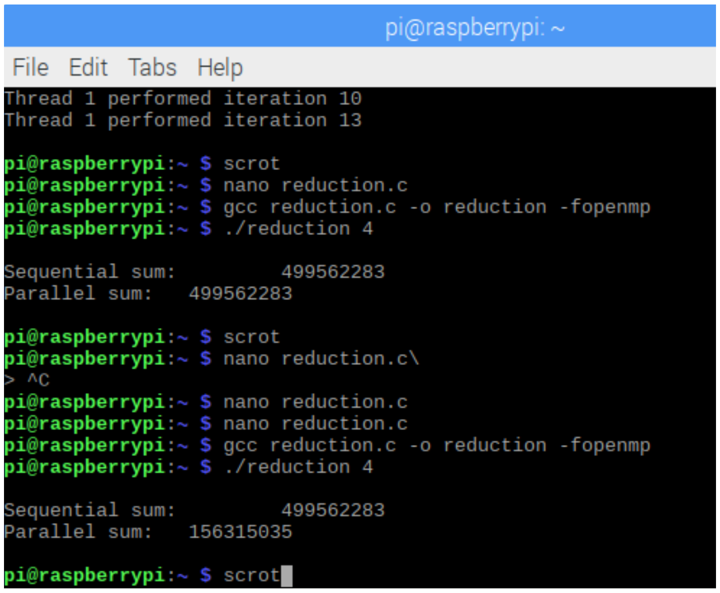
```
Thread 2 performed iteration 14

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop2 3

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

pi@raspberrypi:~ $ scrot
```

Each subsequent iteration was performed on each core one after the other. So thread 0 performed iteration 0, then thread 1 performed iteration 1, and so on. It then cycles back to thread 0 to finish 3 iteration. This is why the iterations thread 0 performed increases by 3. This was confusing at first, but once we noticed a pattern in the iteration numbers, we were able to figure out what was going on.This is because threads that take a short time can pick up the next bit of work while longer threads are still chugging on their piece.



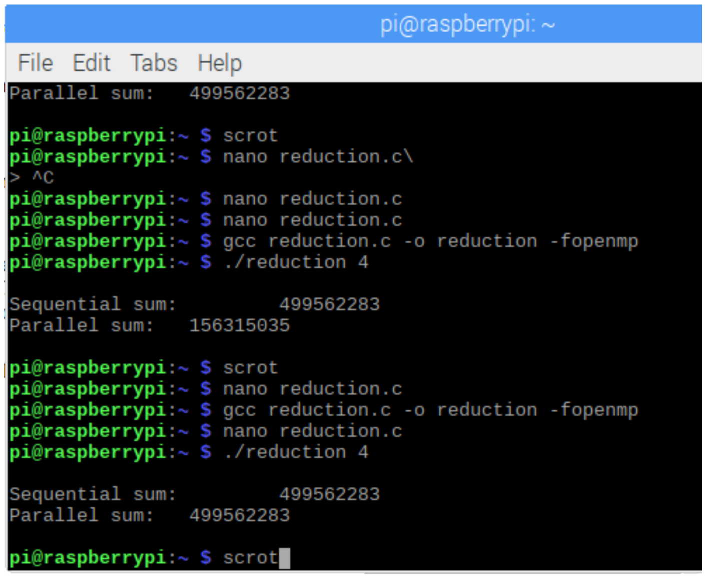
```
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14
Thread 1 performed iteration 1
Thread 1 performed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

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ scrot
```

When we first copied and pasted the code provided in the task instructions, the sequential sum and parallel sum do the exact same thing since line 39 was commented out. Line 39 specifies the clause to use the parallel for OpenMP and a reduction clause is used on the variable sum. However, since it is commented out in line 39, the sequentialSum and parallelSum functions do the same thing.



```
Thread 1 performed iteration 10
Thread 1 performed iteration 13

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano reduction.c\
> ^C
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   156315035

pi@raspberrypi:~ $ scrot
```

The parallel sum and sequential sum are different when the reduction clause is commented out. This is because all values in the array are summed together by using OpenMP parallel for pragma with the reduction clause on the variable sum. In the parallel for loop example, the reduction clause allows the reduction variable to be private as it does its work. So when each thread is finished, the final sum of their individual sums in computed. So since it is commented out in this case, the parallel and sequential sums are different.



```
Parallel sum:   499562283

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano reduction.c\
> ^C
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   156315035

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ scrot
```

Here, since the reduction clause is not commented out, the parallel and sequential sum are sum because of the description given for the screenshot above this one.

Appendix:

Slack group link: https://madscientistsgsu.slack.com
YouTube channel link: https://www.youtube.com/channel/UCwX3csMyKmIZLRPmLBuAk5Q
GitHub link: https://github.com/MadScientists3210/CSC3210-MadScientists-