

Developing Soft and Parallel Programming Skills Using Project-Based Learning

Fall 2018

Coders3210

Steven Nguyen, Jeffrey Shu, Rabia Khan, Asier Yohannes, Aaron Morrison

Report:**Planning:**

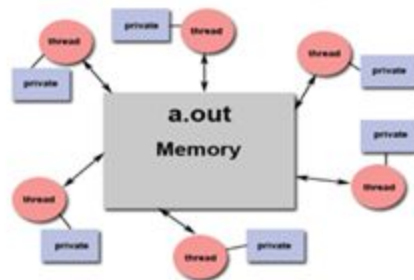
Name:	Email:	Task:	Duration:	Dependency	Availability	Due Date	Note:
Steven Nguyen	snguyen36@student.gsu.edu	Create Written Report and new project on Github, answer questions #1-3, Proofreading	4 hrs	Create the report first so people can start working	Tues/Thurs: 11AM Mon: 10:45-12 Wed: 11:50-3:30 Fri: 9:15-12:30	Report 10/15/18 Answers 10/18/18 Proof 10/25/18	Make sure to update Slack and Github
Jeffrey Shu	jshu3@student.gsu.edu	coding Raspberry Pi, Video, editing video presentation	1.5 hrs	Bring the pi on monday to work on 10/15/18	Mon: 2-3:30 Tues: 12PM Thurs: 3PM	Pi 10/23/18	Meeting with the coordinator on 10/22/18
Rabia Khan (Coordinator)	rkhan19@student.gsu.edu	Helped coding and debugging in Raspberry Pi, writing the lab report.	3 hrs	none	Mon:12:15-4:30 Tues/Thurs/ Fri: free Wed: 1:30 to 3:30PM	Lab report 10/23/18	
Asier Yohannes	ayohannes1@student.gsu.edu	Helped answer reading questions #5-8	1 hr	Do it before the video on 10/18/18	Mon: 11-2PM Tues: after 3PM Wed: 11-2PM Thurs:after 3 Fri: free	Answers 10/18/18	
Aaron Morrison	amorrison16@student.gsu.edu	Helped coding and debugging in Raspberry Pi, helped answer reading question#Q4	2.5 hrs	none	Mon:11-3 Tues:12:15 PM Wed: 12 - 3 PM Thurs: 2 PM Fri: 9:15	Answers 10/18/18	

Parallel Programming Skills:

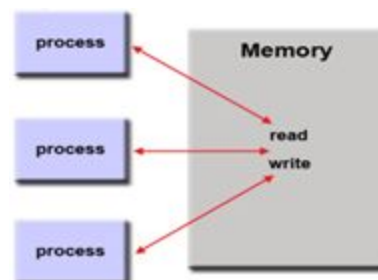
1. Define the following (in your own words):
 - a. **Task:** program or set of instructions that is executed by the processor, parallel programs will have multiple tasks being done on multiple processors
 - b. **Pipelining:** type of parallel programming that splits tasks into smaller steps that are then done by multiple processors
 - c. **Shared Memory:** in terms of hardware = when processors have direct access to the physical memory, in programming = a model where all tasks can see the same memory and access or address it regardless of where the actual location is.
 - d. **Communications:** how parallel tasks exchange data either through a shared memory bus or network, but technically speaking any method to share data counts as communication
 - e. **Synchronization:** how parallel tasks work together, usually done by blocking a task at certain points until another has finished theirs or reached a certain point
2. Classify parallel computers based on Flynn's taxonomy.
 - a. Computer architectures can be classified as either Instruction Stream or Data Stream, and in each category you can be in one of two states; Single and Multiple
 - b. **SISD** : serial computing, deterministic execution, older types of computers
 - i. Single Instruction : one instruction stream done by CPU per clock cycle
 - ii. Single Data : one data stream as input per clock cycle
 - c. **SIMD** : parallel computing, best for specialized problems with high regularity, synchronized and deterministic execution, modern computers
 - i. Single Instruction : all processors execute the same instruction each clock cycle
 - ii. Multiple Data : each processor can work on different data
 - d. **MISD** : parallel computing, rare type
 - i. Multiple Instruction : processors work on data independently through separate data streams
 - ii. Single Data : one data stream is fed to all processors
 - e. **MIMD** : parallel computer, Execution is adaptable (synched or not & deterministic or not), modern supercomputers
 - i. Multiple Instructions : each processor can may be executing a different instruction stream
 - ii. Multiple Data : each processor may be using a different data stream
3. What are the Parallel Programming Models?
 - a. Abstractions above hardware and architectures, & are not specific to any machine
 - i. Shared Memory (without threads)
 - ii. Threads
 - iii. Distributed Memory / Message Passing
 - iv. Data Parallel
 - v. Hybrid
 - vi. Single Program Multiple Data (SPMD)
 - vii. Multiple Program Multiple Data (MPMD)

4. List and Briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?
 - a. Uniform Memory Access (UMA) - a type of shared memory architecture in which identical processors pull from a central memory location; any update to a location in memory is readily apparent to all processors sharing the memory.
 - b. Non-Uniform Memory Access (NUMA) - a type of shared memory architecture in which processors have local and non-local memory at their disposal; in this architecture, access to local memory is faster than to non-local, so processors can't access all memory locations with equal speed.
 - c. OpenMP is able to use both Uniform and Non-uniform memory access architectures; it is designed in such a way to increase OpenMP's flexibility and ability to work with different kinds of memory layouts.
5. Compare Shared Memory Model with Threads Model?

The main difference is Threads model is a type of shared memory programming that has a single main process and can handle multiple smaller processes at the same time.



Threads Model



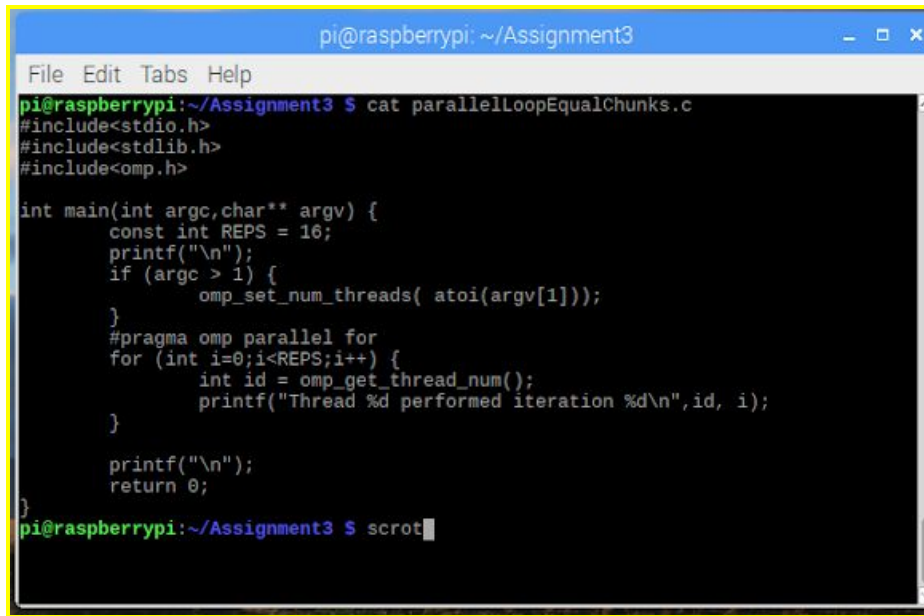
Shared Memory

6. What is Parallel Programming?
 - a. Parallel Programming is basically the use of multiple different computers to handle a certain issue (usually a computational problem).
7. What is system on chip(SoC)? Does Raspberry PI use system on Soc?
 - a. A system on chip is a chip that basically contains a large amount of the computer's components such as a GP, memory, USB controller, power management circuits, and wireless radios. The Raspberry does use a system on SoC.
8. Explain what the advantages are of having a System on a chip rather than separate CPU, GPU and RAM components.
 - a. The main advantage of a system on chip is the ability to have a lot more functionality in a much smaller size. Another small advantage is being able to take much less power which is very beneficial to mobile computing. Small size also leads to less physical material which then results in a cheaper cost to make them.

Lab Report:

The first two code samples were about running loops in parallel. The first one is called Parallel Loop Equal Chunks. If the order isn't important then the code in the loop can be split between

forked threads and this was done by data decomposition pattern bc we are decomposing the amount of work to be done across multiple threads. We do this by using the omp parallel for pragma on line 10 in our code.



```

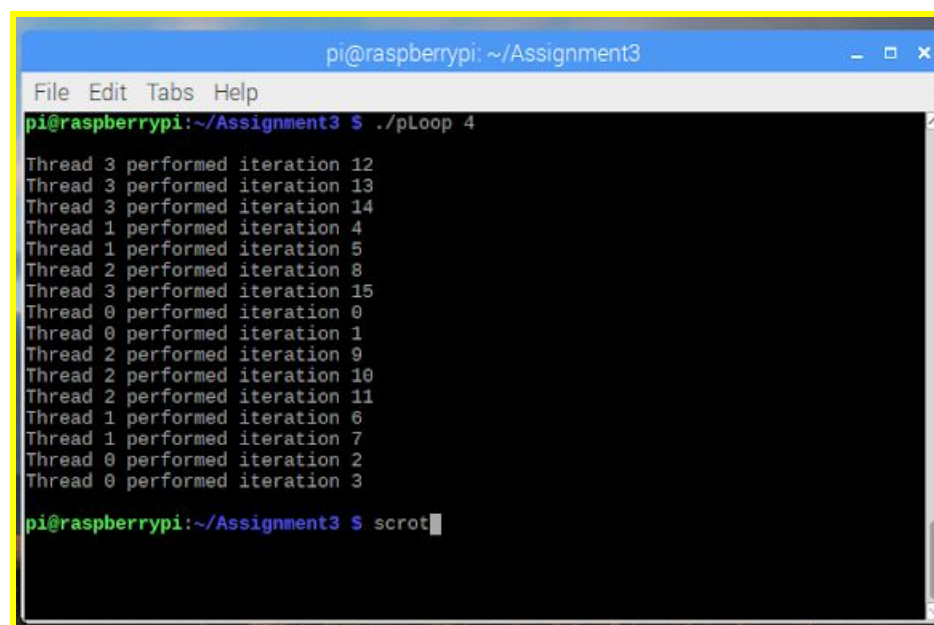
pi@raspberrypi: ~/Assignment3
File Edit Tabs Help
pi@raspberrypi:~/Assignment3 $ cat parallelLoopEqualChunks.c
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

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
    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:~/Assignment3 $ scrot

```

So at first we iterated the loop through equal sized chunks of the index range. This is helpful when one wants to access adjacent memory locations. The output was equally distributed for each number id but the order was random as shown below. The 4 in “./pLoop4” is a command-line argument that indicates how many threads to fork. Since the Raspberry Pi has a 4 core processor, we tried 4 threads. Note: When we tried to use three threads, it was not evenly distributed. Thread Id 0 was repeated 6 times while 1 and 2 ids were repeated 5 times. There were no dependencies between calculations.



```

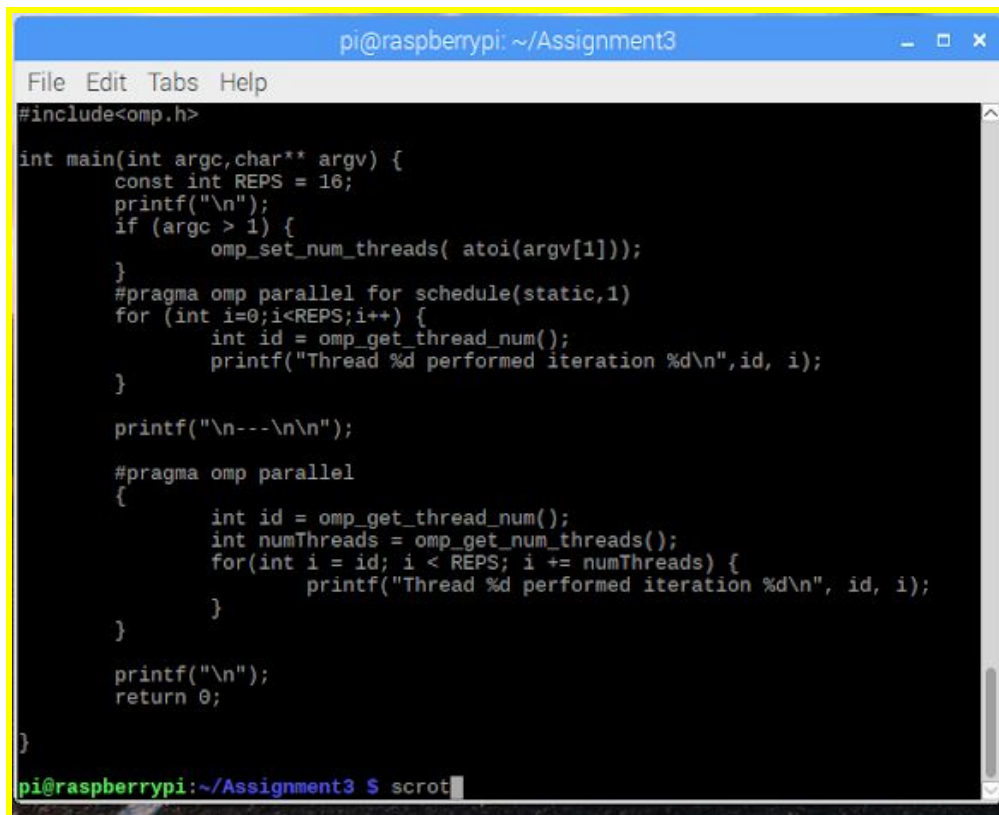
pi@raspberrypi: ~/Assignment3
File Edit Tabs Help
pi@raspberrypi:~/Assignment3 $ ./pLoop 4
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 2 performed iteration 8
Thread 3 performed iteration 15
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 0 performed iteration 2
Thread 0 performed iteration 3

pi@raspberrypi:~/Assignment3 $ scrot

```

The second sample of code was an alternate way to divide the work, it's called Parallel Loop Chunks of 1. In this we divided one iteration of the loop to one thread, the next iteration to the next thread, and so on. Each thread has equal amounts of work but this time it's not with consecutive iterations since the work is done statically. This showed how OpenMP map threads to parallel loop iterations in chunks of size 1, but it is not used to access adjacent memory. In line 10, there's a clause to the omp parallel for pragma that used the word "static" which let's us schedule each thread to do one iteration of the loop in a regular pattern.

Firstly, we compiled and ran the code where the there was a part commented out from line 15, i.e: `printf("\n---\n\n");` to line 25, right before the last three lines in the code: `printf("\n"); return 0;` (as shown below)



```

pi@raspberrypi: ~/Assignment3
File Edit Tabs Help
#include<omp.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");

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        for(int i = id; i < REPS; i += numThreads) {
            printf("Thread %d performed iteration %d\n", id, i);
        }
    }

    printf("\n");
    return 0;
}

pi@raspberrypi:~/Assignment3 $ scro

```

When we ran the code, we got the first block of code (before the dashed lines) as shown in the picture below:

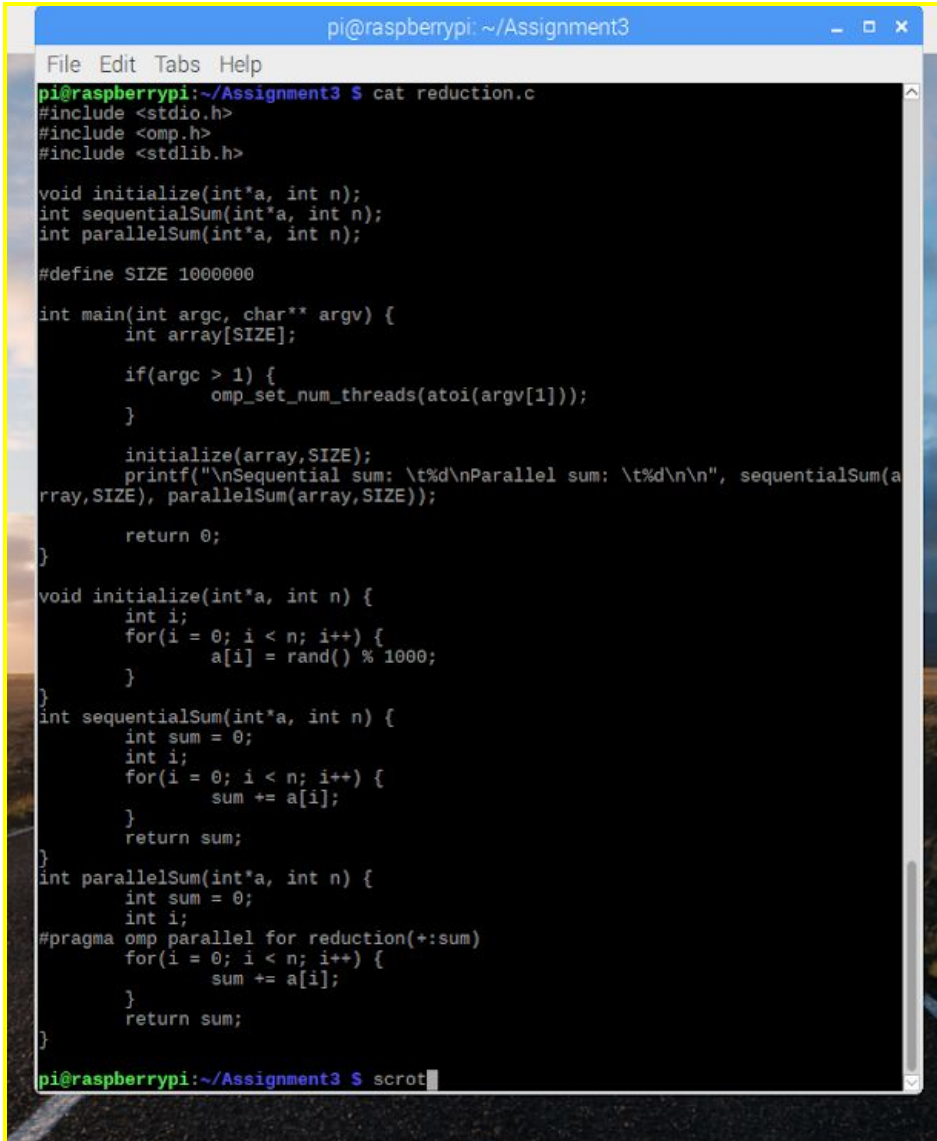
```

pi@raspberrypi:~/Assignment3 $ ./pLoop3 4
Thread 0 performed iteration 0
Thread 1 performed iteration 1
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 5
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 1 performed iteration 9
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 1 performed iteration 13
Thread 3 performed iteration 11
Thread 3 performed iteration 15
---
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
pi@raspberrypi:~/Assignment3 $ scrot

```

The first block of output looks like the equal chunks of version since the order of the threads is random and not in order. Then after uncommenting the part from line 15 and 25 from the code, we got the next block of output with an ordered pattern as shown above in the picture (after the dashed lines). The thread Ids seemed to have the same outputs but the order was different. The first loop was simpler but more restrictive, while the second loop was more complex but less restrictive. This was an alternate way to show how to run loops in parallel.

Finally, the third sample of code was about computing a single value from a set of values by using loops with an accumulator variable in an array or list. We can do this in parallel in openMP, this was called “reduction”. Starting off with sequentialSum, here the sum is added using a for loop in an array. Next, to see how the same thing can be done in parallel with threads, the parallelSum() method is used. Here, the threads communicated to keep the overall sum updated as each of them worked on a portion of the array. As shown on line 39 below (the pragma line) there was a special clause added there with the reduction(+:sum) on the variable sum (computed in line 41):



```

pi@raspberrypi: ~/Assignment3
File Edit Tabs Help
pi@raspberrypi:~/Assignment3 $ cat reduction.c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

void initialize(int*a, int n);
int sequentialSum(int*a, int n);
int parallelSum(int*a, int n);

#define SIZE 10000000

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: %td\nParallel sum: %td\n\n", sequentialSum(array, SIZE), parallelSum(array, SIZE));

    return 0;
}

void initialize(int*a, int n) {
    int i;
    for(i = 0; i < n; i++) {
        a[i] = rand() % 1000;
    }
}

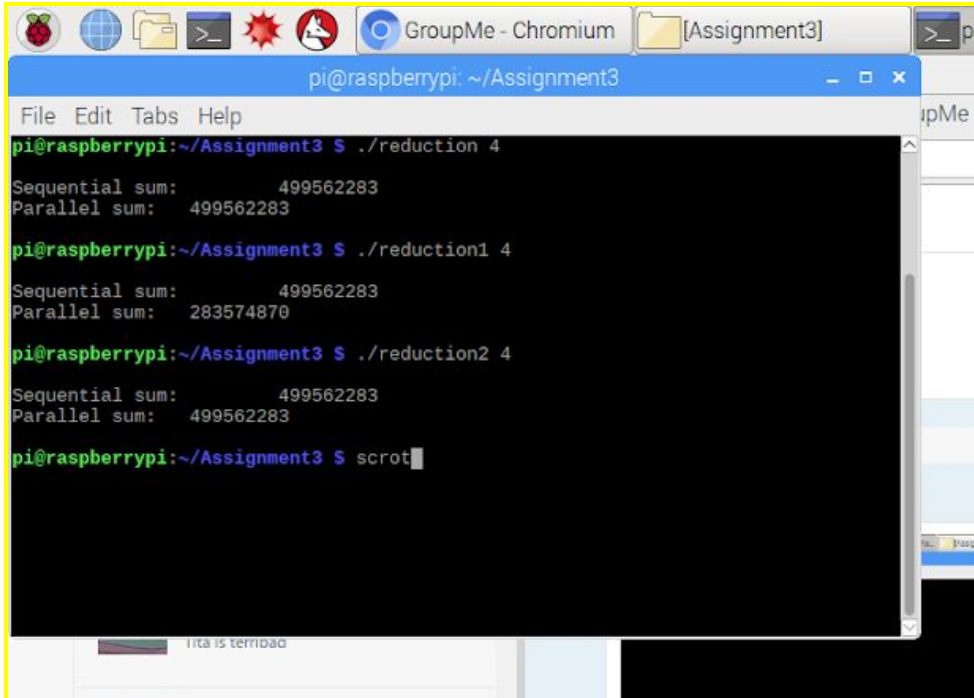
int sequentialSum(int*a, int n) {
    int sum = 0;
    int i;
    for(i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}

int parallelSum(int*a, int n) {
    int sum = 0;
    int i;
#pragma omp parallel for reduction(+:sum)
    for(i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}

pi@raspberrypi:~/Assignment3 $ scrot

```

On line 39 (`#pragma omp parallel for reduction(+:sum)`), it had two commented parts. The first comment was right before `#pragma`, the second comment was on the same line, just before “reduction”. When we ran the code with two commented parts, the output was the same for sequential and parallel sums as shown below (the first block of output on the screen):



```

pi@raspberrypi: ~/Assignment3
File Edit Tabs Help
pi@raspberrypi:~/Assignment3 $ ./reduction 4
Sequential sum:      499562283
Parallel sum:    499562283

pi@raspberrypi:~/Assignment3 $ ./reduction1 4
Sequential sum:      499562283
Parallel sum:    283574870

pi@raspberrypi:~/Assignment3 $ ./reduction2 4
Sequential sum:      499562283
Parallel sum:    499562283

pi@raspberrypi:~/Assignment3 $ scrot

```

Next, we uncommented the first comment right before `#pragma`, and left the second comment there. Then ran the code. The output for sequential sum was different from the output for parallel sum (the second block of output above). This was because the reduction variable, or accumulator called `sum`, needed to be private to each thread as it did its work. The reduction clause in this case was supposed to make that happen.

Finally, we uncommented the second comment to include the reduction clause in our code, and ran it. This time, when each thread was finished, the final sum of their individual sums was computed, hence the correct output. The variable `sum` had a dependency on what all threads were doing to compute it as shown in the last block of code above. That was the most interesting thing we observed in the entire assignment.

Appendix:

- Slack:
→ [Coders3210.slack.com](https://coders3210.slack.com)
- Youtube:

(NOTE: The video was recorded with all of us being recorded individually, however, we were all present there at the same place at the same time.)

Channel: https://www.youtube.com/channel/UCY3UyaKovGc2TEpr_QqfYRw

Video: https://youtu.be/E-p4f_zg8wE

- Github:
→ <https://github.com/Coders3210>

● Screenshot:

