22CSC 3210- Computer Organization and Programming

Assignment 3: Developing Soft and Parallel Programming Skills Using Project Based Learning

(Spring 2020)

Team Name: Team 0101

Thu Vo

Kirby Liu

Humaira Ridi

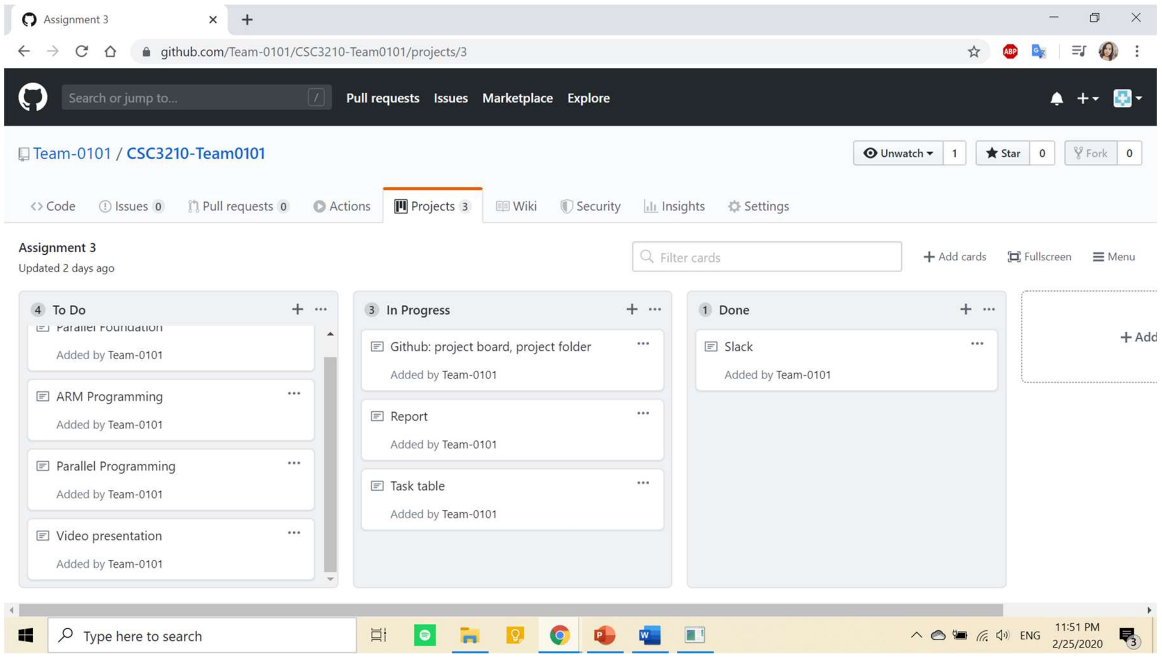
Dhananjay Khazanchi

Dhruv Parikh

Task 1- Planning and Schedule:

Name	Email	Task	Duration	Dependency	Due Date	Note
Thu Vo	Tvo55@student.gsu.edu	Team coordinator	3 hrs	Video, Parallel prog. and ARM	3/5/20	Have everything done by Thursday
Humaira Ridi	Hridi1@student.gsu.edu	Parallel Programming & ARM Assembly	3 hrs	none	3/5/20	Complete parallel programming foundation
Kirby Liu	Kliu8@student.gsu.edu	Youtube Editing	3.5 hrs	Team input	3/5/20	
Dhruv Parikh	Dparikh5@student.gsu.edu	Parallel Programming & ARM Assembly	3 hrs	none	3/5/20	
Dhananjay Khazanchi	Dkhazanchi1@student.gsu.edu	Parallel Programming & ARM Assembly	3 hrs	none	3/5/20	

Task 2 – Github screenshot:



Assignment 3
Updated 2 days ago

4 To Do
- Parallel Foundation — Added by Team-0101
- ARM Programming — Added by Team-0101
- Parallel Programming — Added by Team-0101
- Video presentation — Added by Team-0101

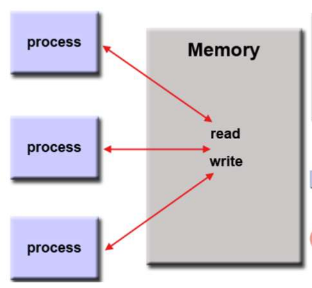
3 In Progress
- Github: project board, project folder — Added by Team-0101
- Report — Added by Team-0101
- Task table — Added by Team-0101

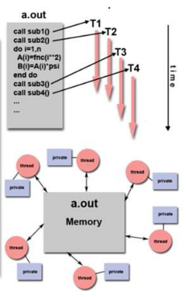
1 Done
- Slack — Added by Team-0101

Dhananjay Khazanchi

Task 3 - Parallel Programming Skills:

1. Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)
    - Task – A portion of work that must be processed by the computer
    - Pipelining – A form of parallel computing where the task is broken down into steps that can be processed at the same time.
    - Shared Memory – all processors have direct access to a common physical memory as well as being able to address and access the same memory regardless of where the physical memory is located
    - Communications – an event in where data is exchanged
    - Synchronization – affects the way parallel tasks are executed in real time by requiring a certain task to wait for another task to first finish processing
2. Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.
    - SISD – A serial (non-parallel) computer that is the oldest type of computer only capable of processing one instruction and data stream at a time
    - SIMD – A parallel computer only capable of processing one instruction stream at a time, but it can apply that instruction to multiple data streams
    - MISD – A parallel computer where multiple instruction streams operate on the same data independently of each other
    - MIMD – The most common type of parallel computer where a processor can be processing both different instruction and data streams
3. What are the Parallel Programming Models?
    - The programming models are shared memory (without threads), threads, distributed memory/message passing, data parallel, hybrid, single program multiple data (SPMD), and 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?
    - Uniform Memory Access (UMA) - processors are identical and equally share memory access and have similar memory access times, and it is cache coherent
    - Non-Uniform Memory Access (NUMA) - processors are physically linked which provides access to the memory of the other processors. The processors do not share equal access times and the memory link is slower.
    - OpenMP is UMA architecture because it connects the processors to a main memory where each processor shares similar access and access times.
5. Compare Shared Memory Model with Threads Model? (in your own words and show pictures)
    - The shared memory model is the simplest form of parallel programming where processes share the same space. The threads model takes the main threads and breaks the thread down into much smaller, more lightweight threads. Each of these threads have local data, but they also are able to see the global memory.



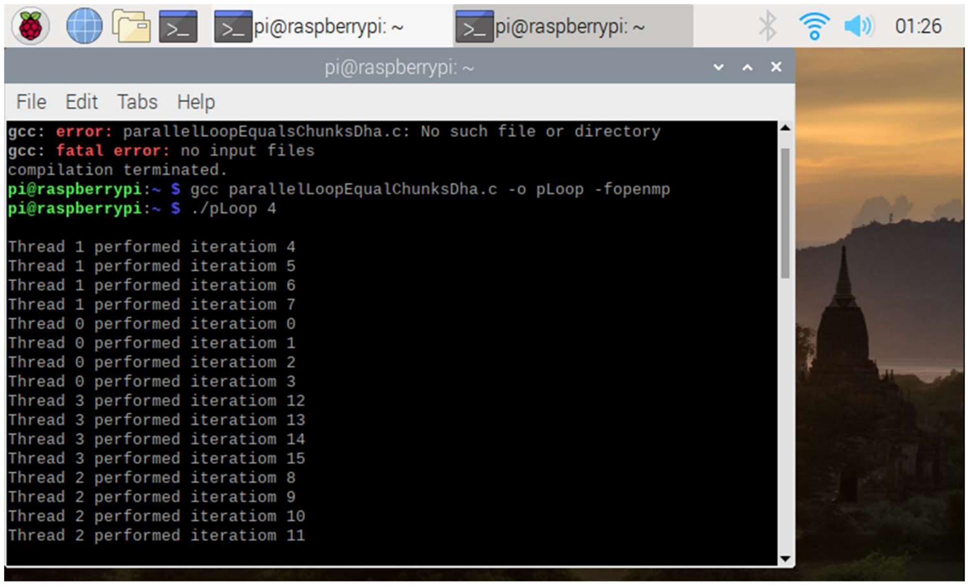


6. What is Parallel Programming? (in your own words)
    - Parallel programming consists of anything that utilizes the components of a computer to perform multiple tasks at the same time whether it be related to the instruction or data streams
7. What is system on chip (SoC)? Does Raspberry PI use system on SoC?
    - A system on chip contains the CPU, the GPU, memory, USB controller, and more all on a single silicon chip.
    - The Raspberry PI uses SoC.
8. Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components
    - The greatest advantage of a SoC is size. A SoC is much smaller than a CPU and all of its various connections
    - Since SoCs are smaller, they also require less wiring and in turn use less power and cost less to produce.

<u>Task 4 – Parallel Programming Basics</u>

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;
}
```

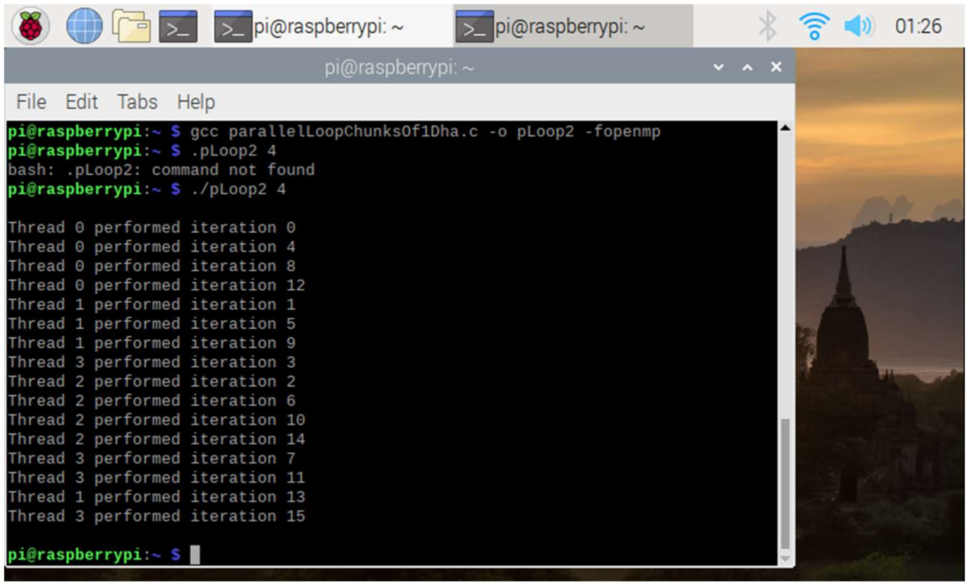


```
gcc: error: parallelLoopEqualsChunksDha.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
pi@raspberrypi:~ $ gcc parallelLoopEqualChunksDha.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

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

There are 16 iterations performed that are evenly distributed among 4 threads. Each thread performs 4 consecutive iterations (Thread 0 does 0, 1, 2, 3), but the output is ordered by the thread number.

ParallelLoopChunksOf1.c

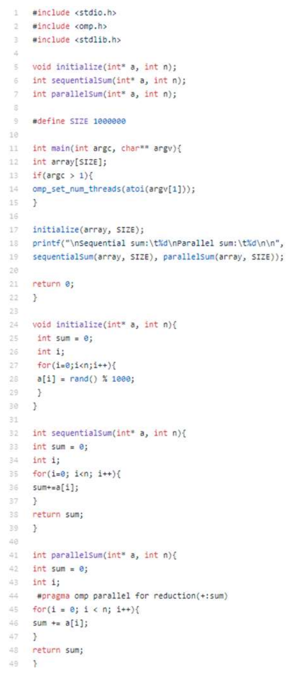
```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

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++){
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n", id, i);
}

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



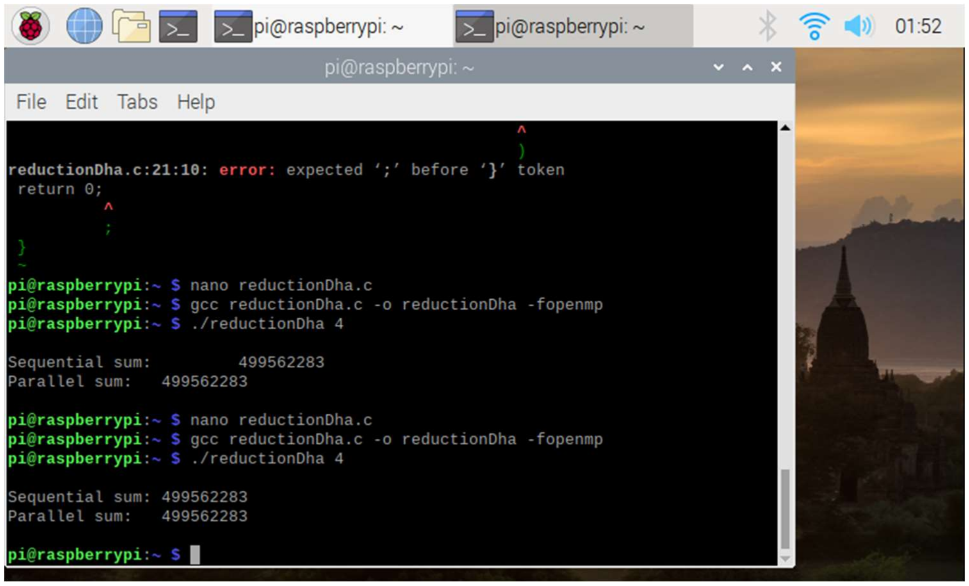
```
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1Dha.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ .pLoop2 4
bash: .pLoop2: command not found
pi@raspberrypi:~ $ ./pLoop2 4

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

pi@raspberrypi:~ $ 
```

In this version, there are still 16 iterations present and each thread performs 4 of the iterations. However, the threads do not perform the iterations in order (Thread 0 performs 0, 4, 8, and 12). The threads take one iteration at the time (Thread 0 does iteration 0 and Thread 1 does iteration 1) because the program allocates the work in chunks of 1.

Reduction.c



```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 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));

return 0;
}

void initialize(int* a, int n){
 int sum = 0;
 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;
}
```



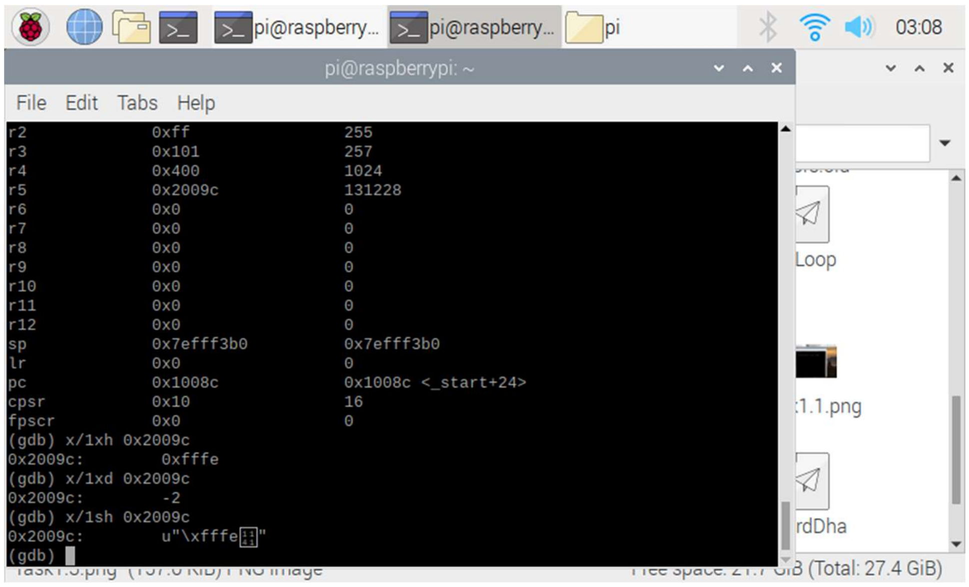
```
                                                               ^
                                                               )
reductionDha.c:21:10: error: expected ';' before '}' token
 return 0;
          ^
          ;
 }
 ~
pi@raspberrypi:~ $ nano reductionDha.c
pi@raspberrypi:~ $ gcc reductionDha.c -o reductionDha -fopenmp
pi@raspberrypi:~ $ ./reductionDha 4

Sequential sum:         499562283
Parallel sum:   499562283

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

Sequential sum: 499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ 
```

This is the result that is seen when the commented portion is present and reduction section is uncommented. Both the sequential and parallel sums are equal because the reduction portion of the code summed up the values in each of the individual threads. If the reduction portion is commented and the #pragma omp parallel is uncommented, then the parallel sum does not equal the sequential sum because it only returns the result of one thread, instead of the combined result.

<u>ARM Assembly Programming</u>

Third.s

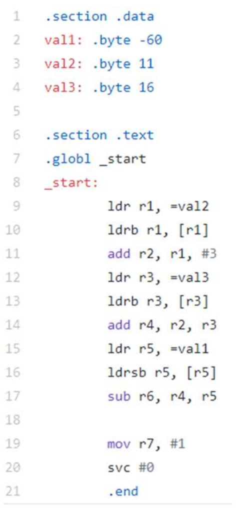
```
.section .data
a: .hword -2

.section .text
.globl _start
_start:
        mov r0, #0x1
        mov r1, #0xFFFFFFFF
        mov r2, #0xFF
        mov r3, #0x101
        mov r4, #0x400
        ldr r5, =a
        ldr r5,[r5]

        mov r7, #1
        svc #0
        .end
```



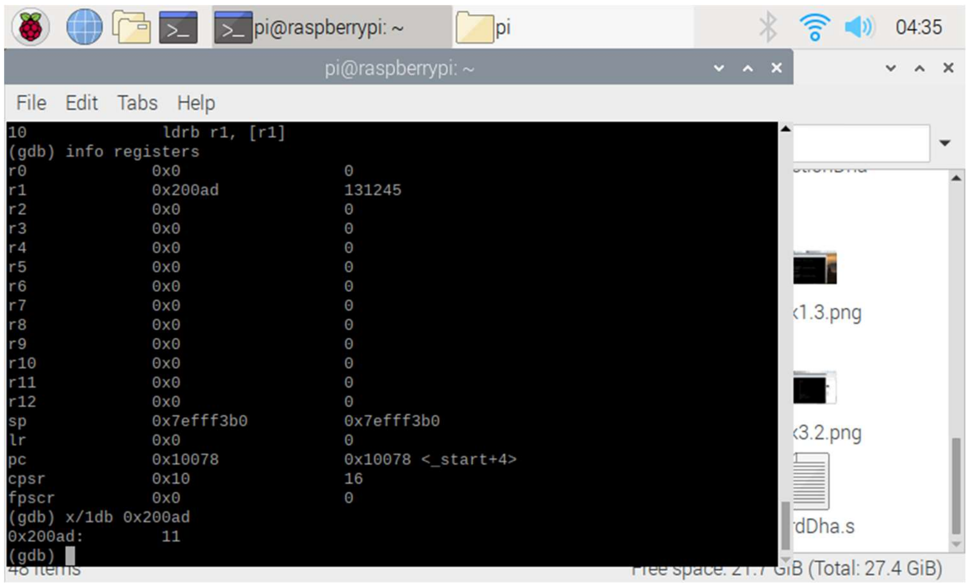
```
r2             0xff                255
r3             0x101               257
r4             0x400               1024
r5             0x2009c             131228
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/1xh 0x2009c
0x2009c:        0xfffe
(gdb) x/1xd 0x2009c
0x2009c:        -2
(gdb) x/1sh 0x2009c
0x2009c:        u"\xfffe"
(gdb)
```

Each of the registers show the corresponding value that is being moved to them. The registers that have negative values show the corresponding unsigned decimal values. However, this can be changed if different loaders are used. Accessing the memory shows that there is a signed value stored in the memory.

Arithmetic3.s

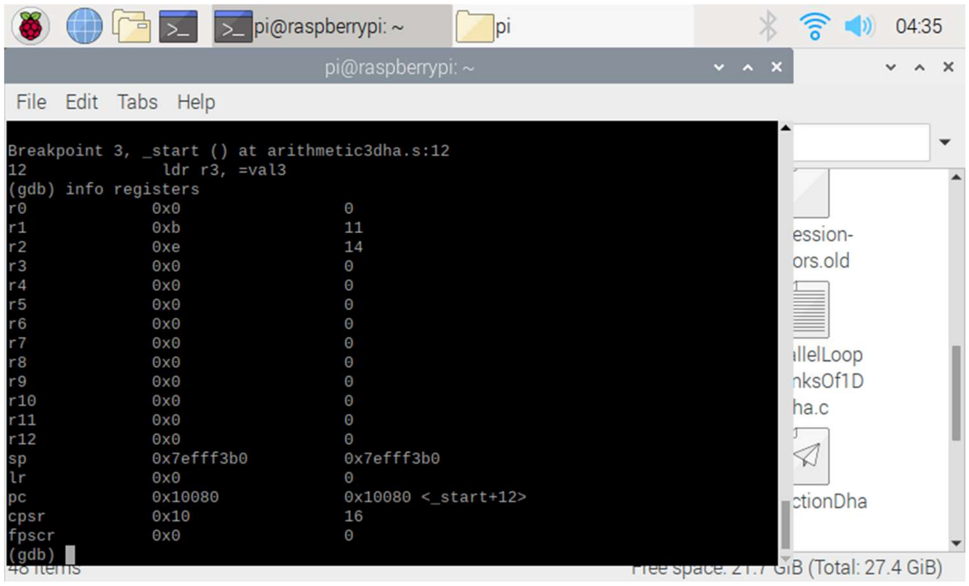


```
.section .data
val1: .byte -60
val2: .byte 11
val3: .byte 16

.section .text
.globl _start
_start:
        ldr r1, =val2
        ldrb r1, [r1]
        add r2, r1, #3
        ldr r3, =val3
        ldrb r3, [r3]
        add r4, r2, r3
        ldr r5, =val1
        ldrsb r5, [r5]
        sub r6, r4, r5

        mov r7, #1
        svc #0
        .end
```



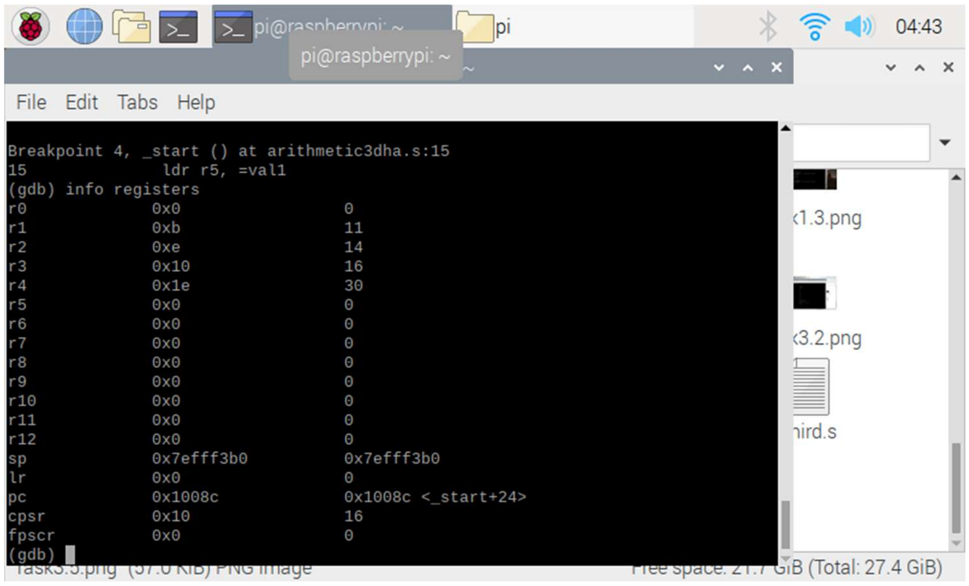
```
10              ldrb r1, [r1]
(gdb) info registers
r0             0x0                 0
r1             0x200ad             131245
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/1db 0x200ad
0x200ad:        11
(gdb)
```

The address of val2 is loaded into r1. This is done because ARM is register/register, so to pull a value from the memory, the memory address must first be loaded into a register, before the value can be loaded. Similar results are seen when the other variables are loaded into the registers.



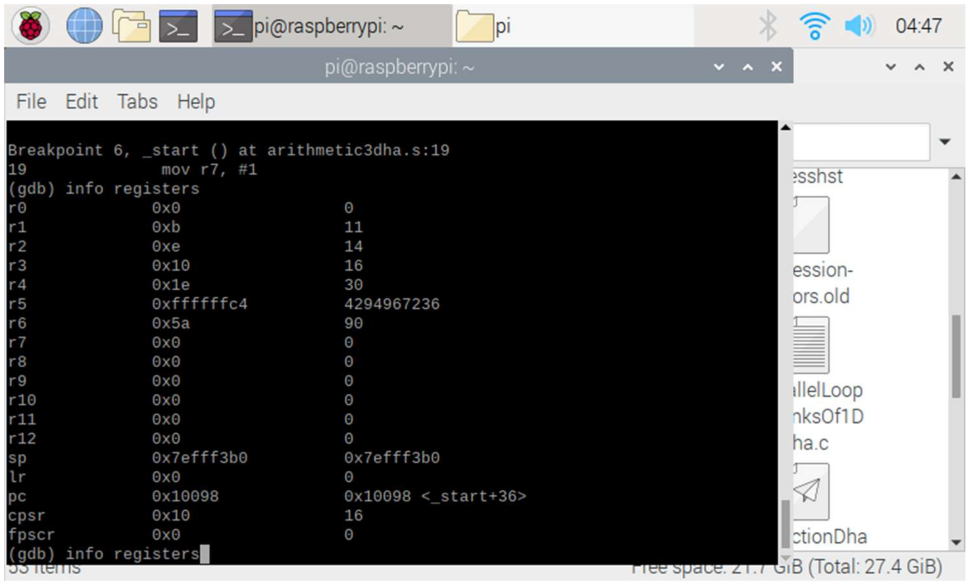
```
Breakpoint 3, _start () at arithmetic3dha.s:12
12              ldr r3, =val3
(gdb) info registers
r0             0x0                 0
r1             0xb                 11
r2             0xe                 14
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10080             0x10080 <_start+12>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

The value 11 is loaded from the memory address stored in r1. Then, the immediate, 3, is added to the value in r1 and stored into r2.



```
Breakpoint 4, _start () at arithmetic3dha.s:15
15              ldr r5, =val1
(gdb) info registers
r0             0x0                 0
r1             0xb                 11
r2             0xe                 14
r3             0x10                16
r4             0x1e                30
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

After the memory address of val3 is loaded into r3 and the value is pulled from the memory, the contents of r2 and r3 are added together and stored into r4.



```
Breakpoint 6, _start () at arithmetic3dha.s:19
19              mov r7, #1
(gdb) info registers
r0             0x0                 0
r1             0xb                 11
r2             0xe                 14
r3             0x10                16
r4             0x1e                30
r5             0xffffffc4          4294967236
r6             0x5a                90
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10098             0x10098 <_start+36>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) info registers
```

The memory address of val1 is loaded into r5 and the unsigned value of –60 is shown in decimal. Then, the value in r4 is added to the value in r5, and the resulting value (90) is stored in r6.

Thu Vo

Task 3 - Parallel Programming Skills:

Foundation

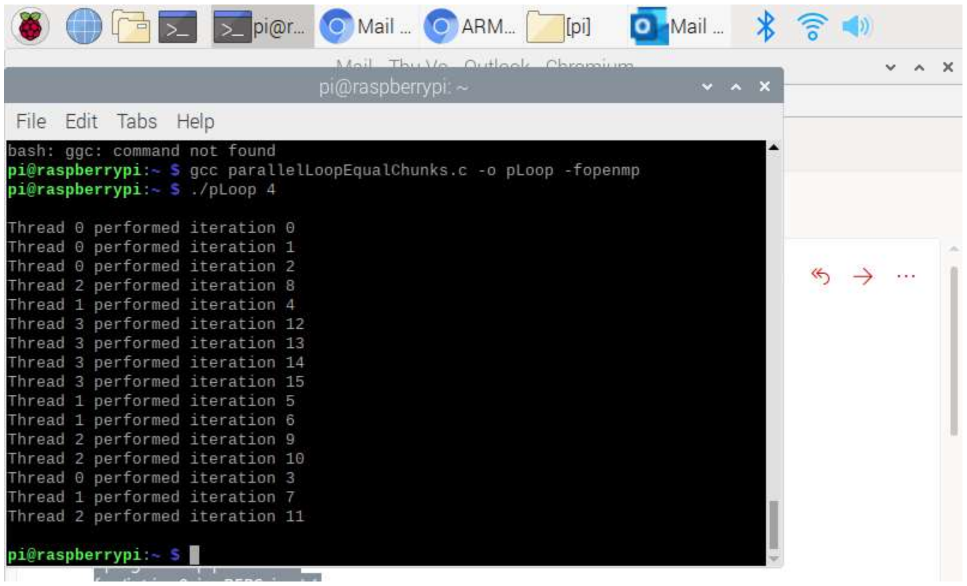
1. Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.
    - Task: A program or a section of a program executed by a processor. In parallel program and multitasking environment multiple tasks running on multiple processors.
    - Pipelining: A type of parallel computing, much like an assembly line. Breaking up instruction into stages so the processors can work on different stages of the instruction in an overlapped timeline.
    - Shared Memory: In hardware, different processors can access the same block of memory. In software, multiple tasks share the same global address space and can read and write the same logical memory locations regardless of the physical memory.
    - Communications: Data exchange between parallel tasks. Parallel program communicating through shared memory bus or memory over a network.
    - Synchronization: Waiting for one task or tasks to be completed or get to the same logical point for another task to continue. Synchronization is one of the key problems in parallel computing, because of the increased waiting time.
2. Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.
    - SISD: Sequential computer, one CPU execute one single instruction with one data stream as input. Oldest type of computing exploits no parallelism.
    - SIMD: Execute the same instruction on different elements of data. Best suited for graphic processing, a specialized problem with high degree irregularity.
    - MISD: Multiple processing units operate different instruction on same the stream of data. It is an uncommon architecture generally used for fault tolerance, multiple cryptography algorithms attempting to crack a single coded message.
    - MIMD: All processors may execute different instructions on different streams of data. Most current supercomputers are examples of this architecture.
3. What are the Parallel Programming Models?
    - A parallel programming model is an abstraction above hardware and memory of parallel computer architecture.
    - Examples: Threads, data parallel, shared memory, hybrid, distributed memory, MPSD, SPMD.
4. List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?
    - Uniform Memory Access (UMA): All processors have equal access and access times to physical memory.

- Non-Uniform Memory Access (NUMA): Each processor has its own local memory but can also access memory owned by other processors. Access time depends on the memory location relative to the processor.
- OpenMp is used by Uniform Memory Access, because all the threads in OpenMp shared the same memory.

5. Compare Shared Memory Model with Threads Model?
    - Shared Memory Model: Processes share a common address that they can read and write to and locks are used to prevent processes from accessing the memory at the same time, which will create a deadlock.
    - Threads Model: threads or lightweight processes happened in an overlapping timeline, each thread can have their local data and share global data with other threads, global data is how threads communicate to each other. However global data requires threads to not access it at the same time.
6. What is Parallel Programming?
    - Programming that utilized the multicore processor. That means dividing large sequential tasks into smaller tasks that can be executed concurrently.
7. What is system on chip (SoC)? Does Raspberry PI use system on SoC?
    - System-on-a-chip integrated components like CPU, graphics processor, memory, essentially all components of a computer into a single silicon chip.
    - The Raspberry Pi has a SoC unit with CPU, RAM, and GPU in one component.
8. Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components
    - A System on a Chip saves more space than having separate components, this means more space for battery pack. System on a Chip also has shorter wiring, so it uses less power. So, SoC is an advantage when it comes to mobile computing.

Parallel Programming Basics

<u>parallelLoopEqualChunks.c</u>

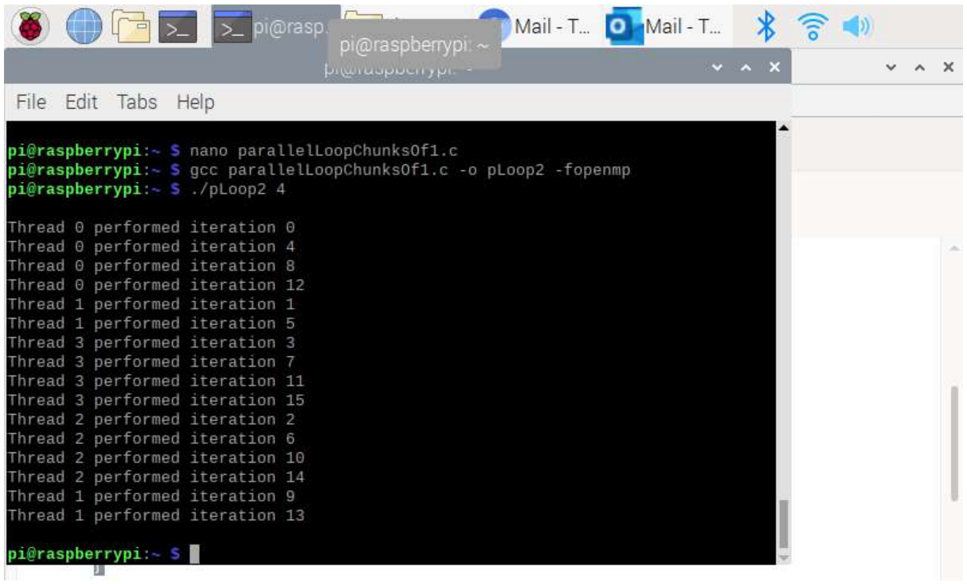


```
bash: ggc: command not found
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

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

pi@raspberrypi:~ $ 
```

The number of iterations, in this code is 16, are divided equally among the four threads. Each thread will take 4 consecutive iterations.

When the number of iterations is not evenly divisible by the number of threads, some of the threads will perform one more or less iteration.

parallelLoopChunksOf1.c

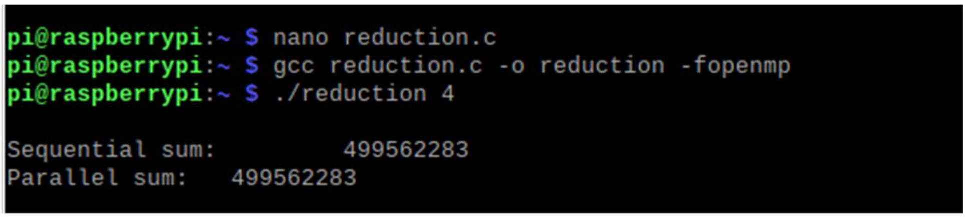


```
pi@raspberrypi:~ $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4

Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 5
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 1 performed iteration 9
Thread 1 performed iteration 13

pi@raspberrypi:~ $
```

It looks like each thread starts with the iteration number same as the thread id. For example, Thread 0 starts at iteration 0, Thread 1 starts at iteration 1 and so on. Then thread current will start from where last thread left off, so the iterations are nonconsecutive. For example, Thread 0 does iteration 0, 4, 8, 12 and Thread 1 does iteration 1, 5, 9, 13 and so on.

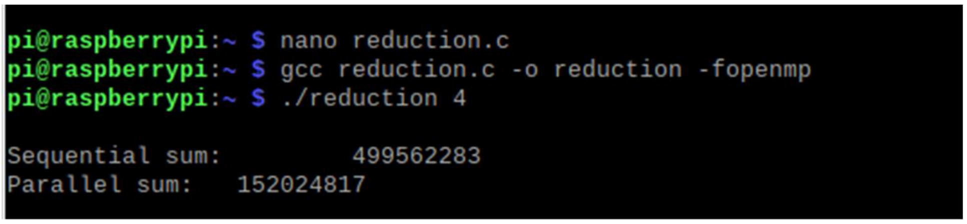
Because the scheduling is static and 1, each thread is taking one iteration at a time.

reduction.c



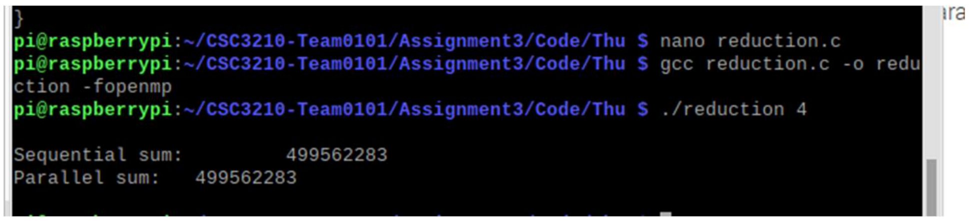
```
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283
```

This is the code without removing any comment font. The sequential and parallel are the same because the code in parallel is not forked.



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

Sequential sum:         499562283
Parallel sum:   152024817
```

This is the code when removing the first // in line 39, this forked the code into multiple threads. However, the result of sequential and parallel parts are not the same. This is because the different threads did not communicate with each other, so the result is from one of the threads.

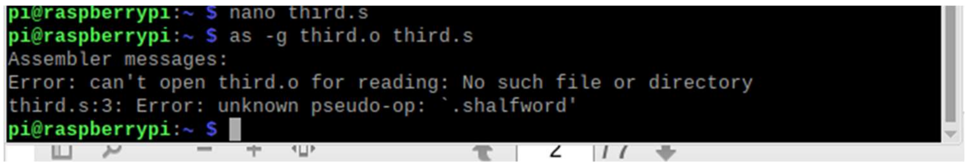


```
}
pi@raspberrypi:~/CSC3210-Team0101/Assignment3/Code/Thu $ nano reduction.c
pi@raspberrypi:~/CSC3210-Team0101/Assignment3/Code/Thu $ gcc reduction.c -o redu
ction -fopenmp
pi@raspberrypi:~/CSC3210-Team0101/Assignment3/Code/Thu $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283
```

This is the code with second // removed so that the reduction clause is uncommented. The result of sequential and parallel parts are matching. The reduction(+:sum) summed the partial loops' results into the sum variable.

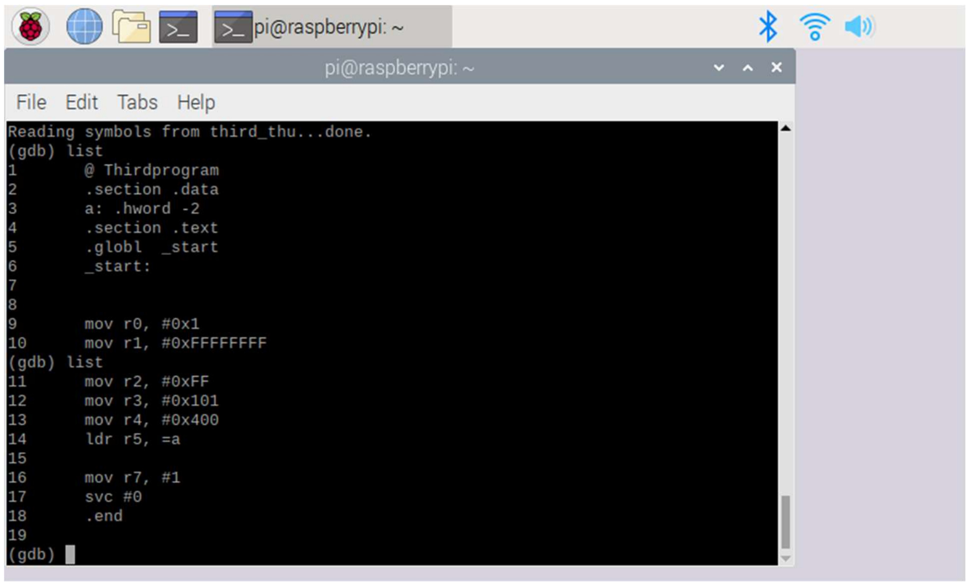
<u>Task 4 - ARM Assembly Programming:</u>

<u>Third.s</u>



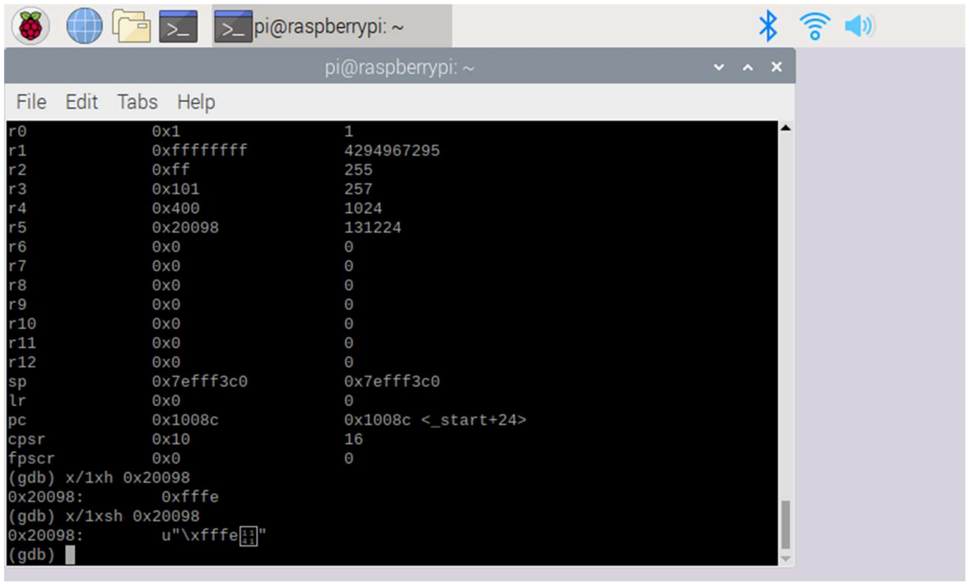
```
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g third.o third.s
Assembler messages:
Error: can't open third.o for reading: No such file or directory
third.s:3: Error: unknown pseudo-op: `.shalfword'
pi@raspberrypi:~ $
```

The error is that there is no such thing as a shalfword memory size in ARM, there is only hword which stand for half word in ARM. Unlike x86 you declare a value is signed or unsigned when you load it into a register



```
Reading symbols from third_thu...done.
(gdb) list
1       @ Thirdprogram
2       .section .data
3       a: .hword -2
4       .section .text
5       .globl  _start
6       _start:
7
8
9       mov r0, #0x1
10      mov r1, #0xFFFFFFFF
(gdb) list
11      mov r2, #0xFF
12      mov r3, #0x101
13      mov r4, #0x400
14      ldr r5, =a
15
16      mov r7, #1
17      svc #0
18      .end
19
(gdb)
```

The code fixed.

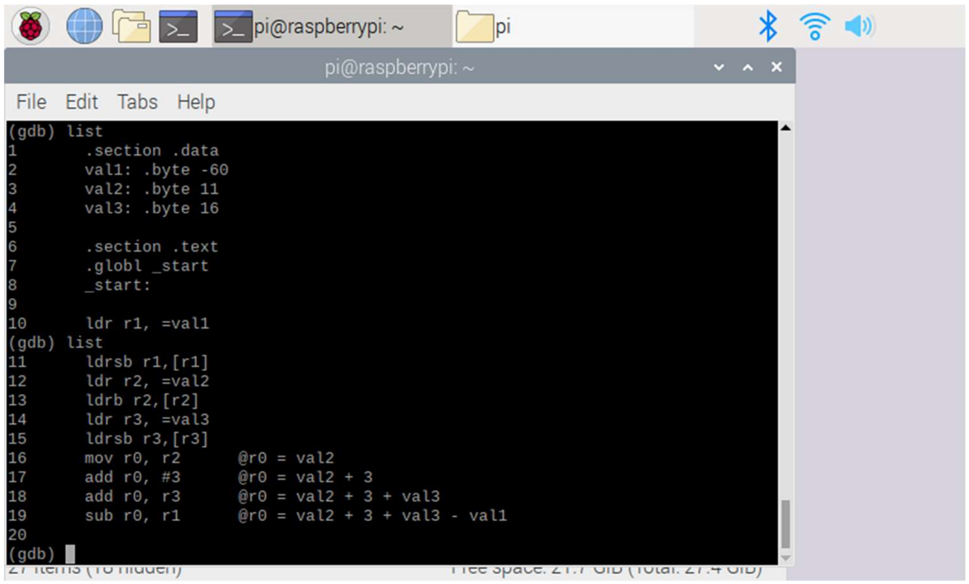


```
r0             0x1                 1
r1             0xffffffff          4294967295
r2             0xff                255
r3             0x101               257
r4             0x400               1024
r5             0x20098             131224
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3c0          0x7efff3c0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/1xh 0x20098
0x20098:        0xfffe
(gdb) x/1xsh 0x20098
0x20098:        u"\xfffe[illegible]"
(gdb)
```

mov r1, #0xFFFFFFFF is a signed value because the most significant byte is F which mean the sign-bit is 1. mov r2, #0xFF is an unsigned value because the most significant byte is not F but instead it is 0

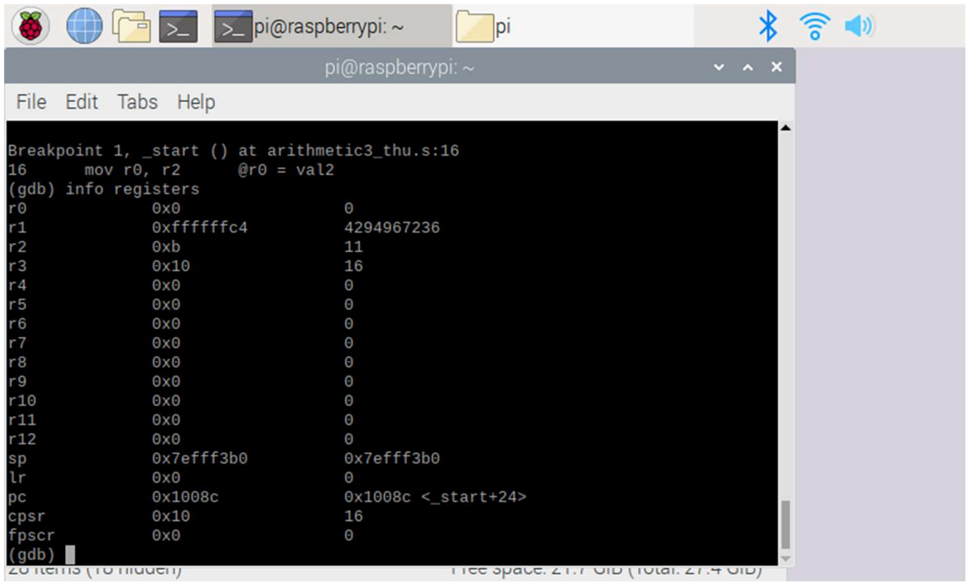
When looked into the memory at address of a in both signed and unsigned format, the values are the same it is fffe, a signed value.

<u>arithmetic3.s</u>



```
(gdb) list
        .section .data
        val1: .byte -60
        val2: .byte 11
        val3: .byte 16

        .section .text
        .globl _start
        _start:

        ldr r1, =val1
(gdb) list
        ldrsb r1,[r1]
        ldr r2, =val2
        ldrb r2,[r2]
        ldr r3, =val3
        ldrsb r3,[r3]
        mov r0, r2      @r0 = val2
        add r0, #3      @r0 = val2 + 3
        add r0, r3      @r0 = val2 + 3 + val3
        sub r0, r1      @r0 = val2 + 3 + val3 - val1

(gdb)
```

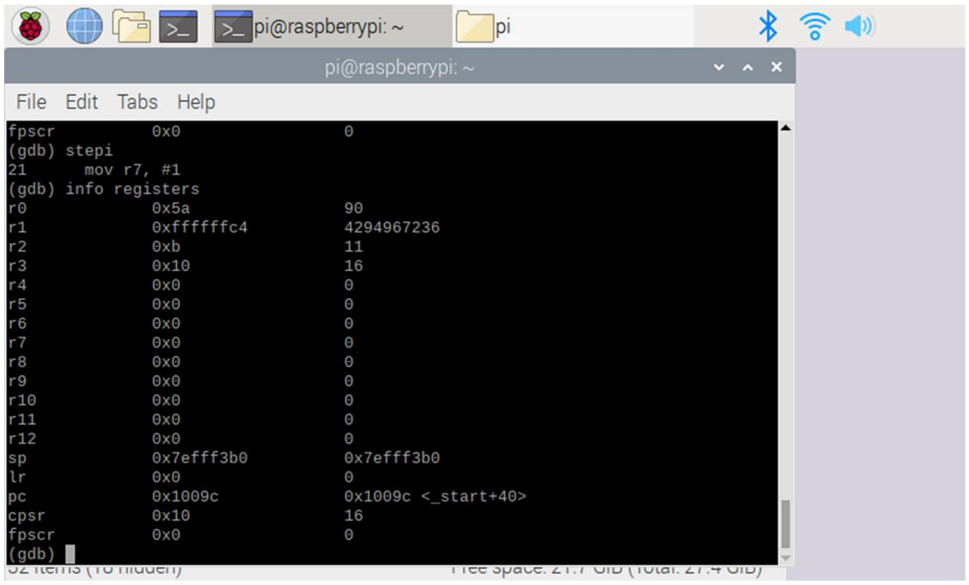
Since val1, val2, and val3 are 8-bit integer memory variables, 8-bit memory size in ARM is byte.



```
Breakpoint 1, _start () at arithmetic3_thu.s:16
16      mov r0, r2      @r0 = val2
(gdb) info registers
r0             0x0                 0
r1             0xffffffc4          4294967236
r2             0xb                 11
r3             0x10                16
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

This is after all the memory variables loaded into registers.

Val1 will be loaded to register r1 as a signed value using ldrsb loader, since it is a negative value. Val2 will be loaded to register r2 as an unsigned value, because the direction explicitly stated that val2 is unsigned. Val3 will be loaded to register r3 as a signed value follow the direction.



```
fpscr          0x0                 0
(gdb) stepi
21      mov r7, #1
(gdb) info registers
r0             0x5a                90
r1             0xffffffc4          4294967236
r2             0xb                 11
r3             0x10                16
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1009c             0x1009c <_start+40>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

This is after the arithmetic execution following Register = val2 + 3 + val3 - val1. The result is stored in register r0. The final result is 11 + 3 + 16 - (-60)= 90 and 5Ah= 90d. The value in r0 is 5a.

Kirby Liu

Task 3 - Parallel Programming Skills:

1. **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
   - **Task**: A program or program-like set of instructions that is executed by a processor. A parallel program can have multiple tasks running on multiple processors.
   - **Pipelining**: The breakdown of tasks into steps that are performed by different processor units, with the inputs streaming through, where the output of a pipeline can be the input of another pipeline interweaved through parallel programing.
   - **Shared Memory**: Describes the computer architecture where all processors can directly address and access a common physical memory.
   - **Communications**: The event of data exchange either through shared memory or over a network that is required when executing parallel tasks.
   - **Synchronization**: The coordination of parallel tasks, often through ensuring that a certain point or logical condition is met before another task can proceed.
2. **Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.**

   Flynn's Taxonomy is based on 4 variables Single/Multiple Instructions and Single/Multiple Data Steam

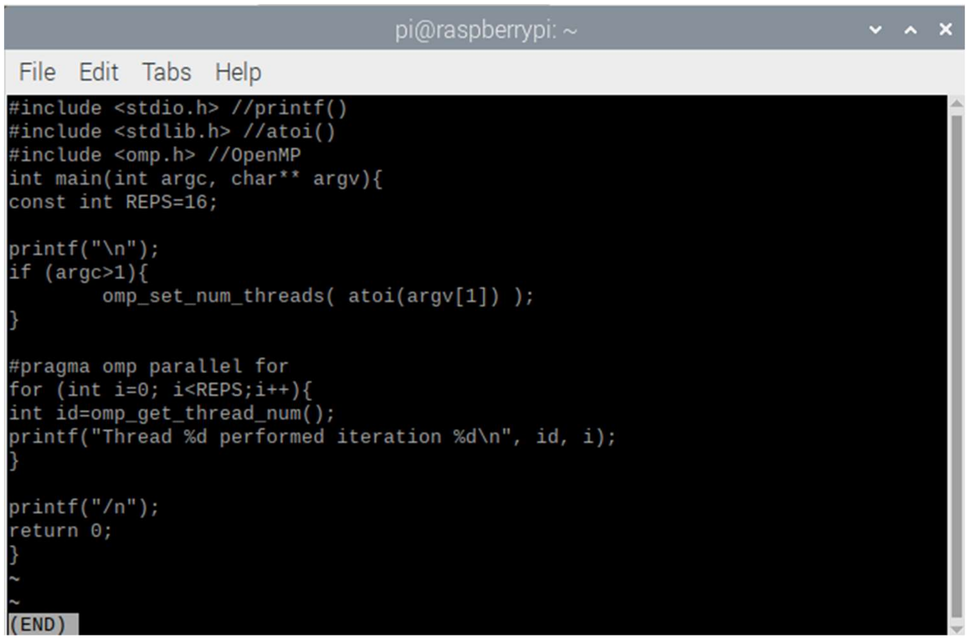
   - **SISD**: Single Instruction Single Data Stream- Serial (Non-Parallel) computer with only one instruction and one data stream being acted on during any one clock cycle
   - **SIMD**: Single Instruction Multiple Data Stream- Parallel computer with only one instruction stream and multiple data stream being processed by multiple processing units
   - **MISD**: Multiple Instruction Single Data Stream- Parallel computer that can have multiple processors operating on multiple instructions independently though these multiple processers are fed a single data stream
   - **MIMD**: Multiple Instruction Multiple Data Stream- A type of parallel computer that can handle multiple instructions and multiple data streams across multiple processors that can be synchronous or asynchronous, deterministic or non-deterministic
3. **What are the Parallel Programming Models?**
   - A parallel programming model is a set of programming abstraction for allowing parallel activities from application to the parallel hardware.
   - Examples: o Shared Memory (without threads) o Threads o Distributed Memory / Message Passing o Data Parallel o Hybrid o Single Program Multiple Data (SPMD) o 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?**

- Uniform Memory Access (UMA): Identical processors with equal access and access time to memory
- Non-Uniform Memory Access (NUMA): Made by physically linking two or more Symmetric Multiprocessor (SMP). One SMP can access another SMP, however not all processors have equal access time to all memories.
- OpenMp is shared memory architecture which UMA and NUMA is bother considered shared architecture.

5. **Compare Shared Memory Model with Threads Model?**
    - Shared Memory Model: A memory model where processes/tasks share a common address space, which they read and write to asynchronously.
    - Threads Model: In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths. . This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
6. **What is Parallel Programming?**
    - Parallel programming is the ability to break down larger tasks and/or computations into smaller instructions/data streams to be processed simultaneously and then assembled back together.
7. **What is system on chip (SoC)? Does Raspberry PI use system on SoC?**
    - A personal computer often has separate but necessary components such as a CPU which that then connects with memory, USB Controller, audio chip, graphic processor, etc to create a functioning computer. A SOC is system-on-a-chip that integrates all those components into a a single silicon chip that integrates the CPU along with GPU, memory, USB controller, powers management circuits to create a working computing device.
    - The Raspberry PI uses a SOC system.
8. **Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components**
    - System on Chip are smaller than systems that have separate CPU, memory, graphic processing chips. The smaller SOC are more suited to mobile, handheld devices. Additionally, the SOC uses much less power allowing for longer battery life.

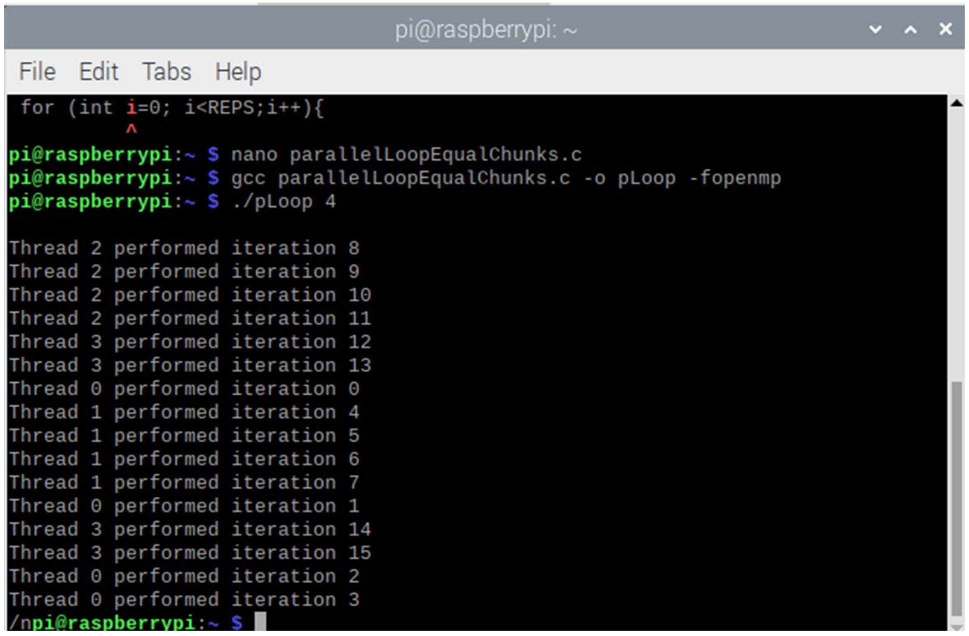
Parallel Programming Basics

ParallelLoopEqualChunks.c-Screenshot



```
#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;
}
~
~
(END)
```

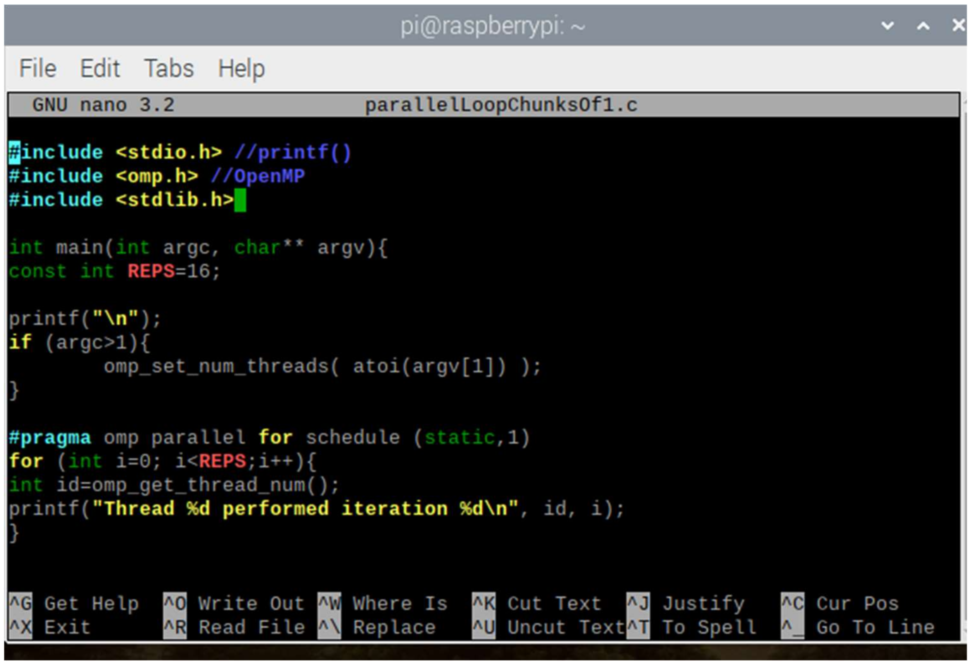
<u>Running ./pLoop 4 of parallelLoopEqualChunks.c- Screenshot</u>



```
 for (int i=0; i<REPS;i++){
          ^
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

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 0 performed iteration 0
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 0 performed iteration 1
Thread 3 performed iteration 14
Thread 3 performed iteration 15
Thread 0 performed iteration 2
Thread 0 performed iteration 3
/npi@raspberrypi:~ $ 
```

<u>Running ./pLoop 4 of parallelLoopEqualChunks.c-</u> Executing the parallelLoopEqualChunks.c program with 4 threads splits the 16 iterations into 4 equal iterations across the 4 Threads. Thread 0 performs iterations 0-3, thread 1 performs iteration 4-7, thread 3 performs iterations 8-11, and thread 4 performs iterations 12-15. If the number of iterations are not evenly divisible by the number of threads, then the iterations are split with every thread having quotient or quotient+1 number of iterations remainder sum of threads having the quotient+1 iterations.

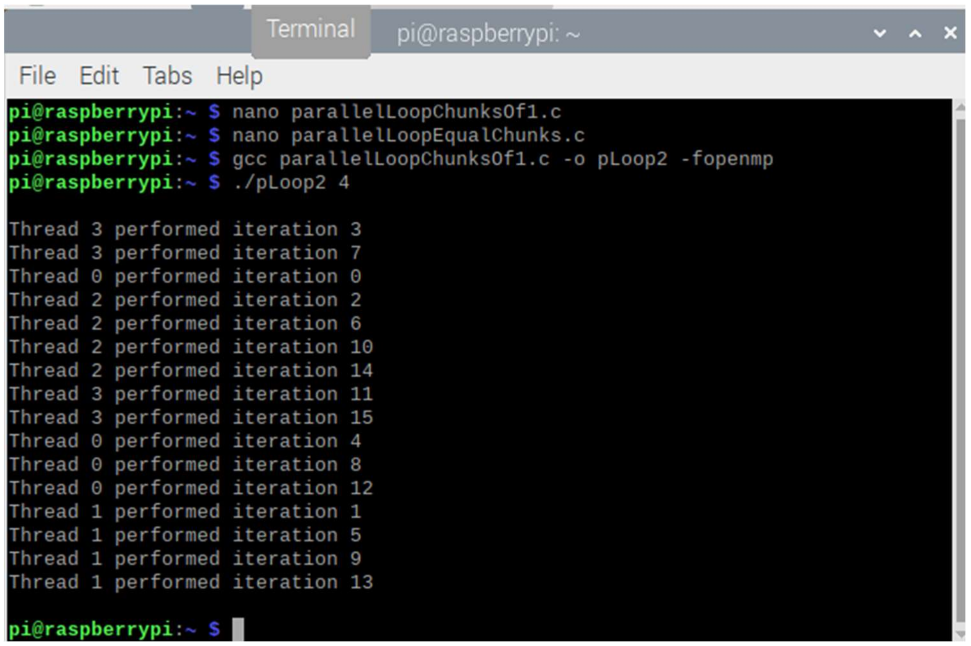
<u>ParallelLoopChunksOf1.c-Screenshot</u>


pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2 parallelLoopChunksOf1.c

```
#include <stdio.h> //printf()
#include <omp.h> //OpenMP
#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);
}
```

^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

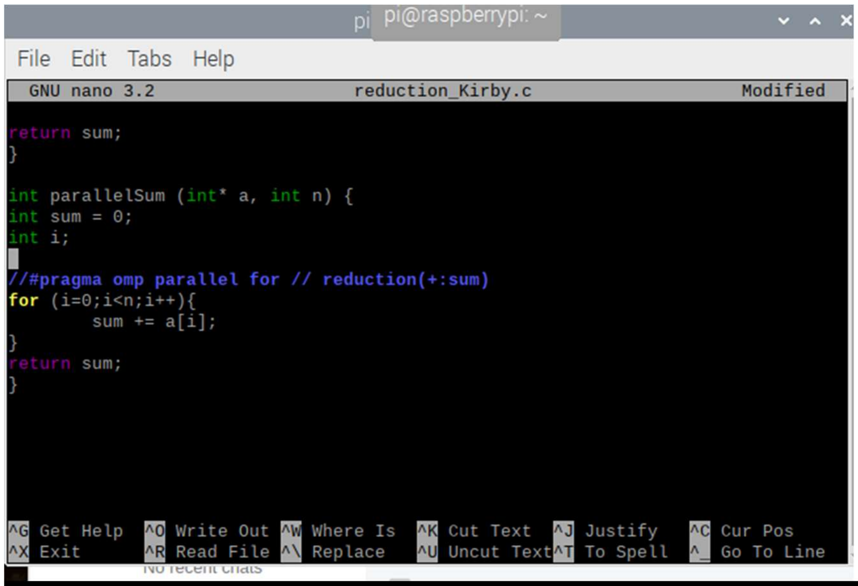
Running ./pLoop2 4 parallelLoopChunksOf1.c- Screenshot



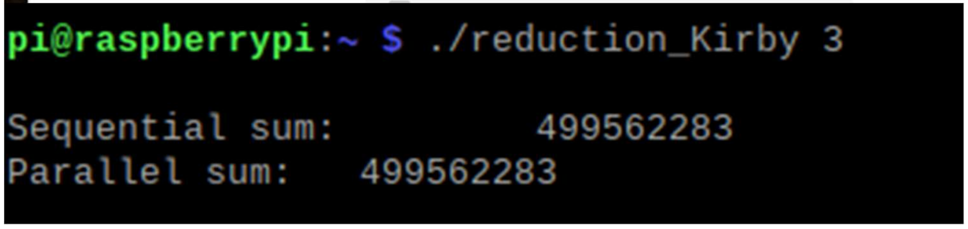
```
pi@raspberrypi:~ $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 0 performed iteration 0
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 3 performed iteration 11
Thread 3 performed iteration 15
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:~ $
```

Running ./pLoop2 4 parallelLoopChunksOf1.c- When we run the parallelLoopChunksOf1.c we see that the iterations are cycled through the Threads in 1 cycles instead of as blocks of iterations. So we see that iteration 0 is performed by thread 0, iteration 1 by thread 1, iteration 2 by thread 2, iteration 3 by thread 3 and since we specified 4 forks, iteration 4 is handled by thread 0 and so on until the iterations are completed. Line 10, #pragma omp parallel for schedule(static,1), the "1" sets the iteration blocks to size 1. If set #pragma omp parallel for schedule(static,2), then the iteration block will be set to size 2.

Reduction.c //#pragma omp parallel for // reduction(+:sum)- Screenshot



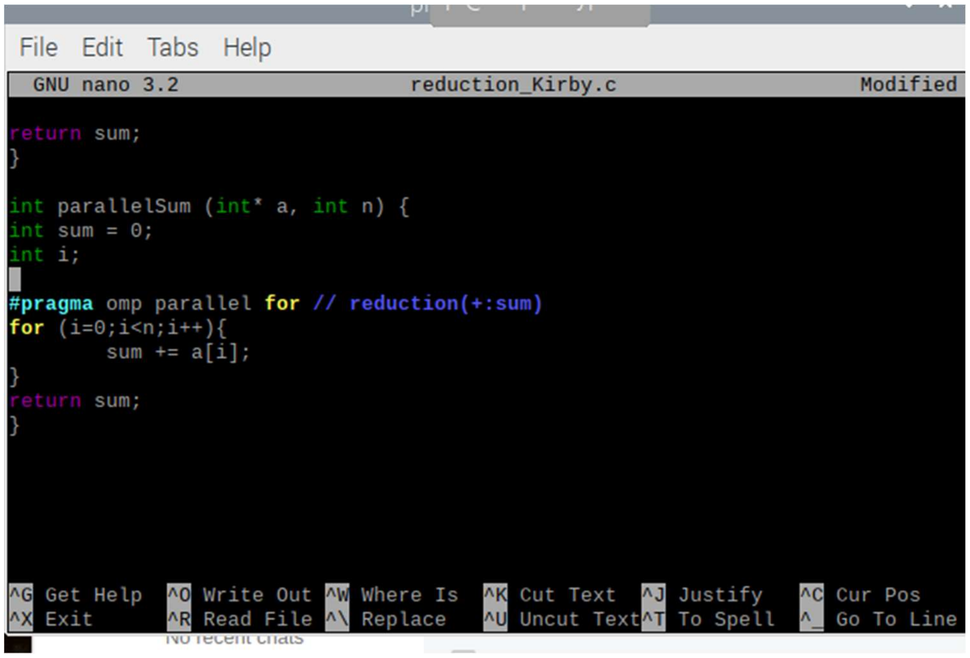
```
GNU nano 3.2                reduction_Kirby.c                 Modified

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;
}

^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
```



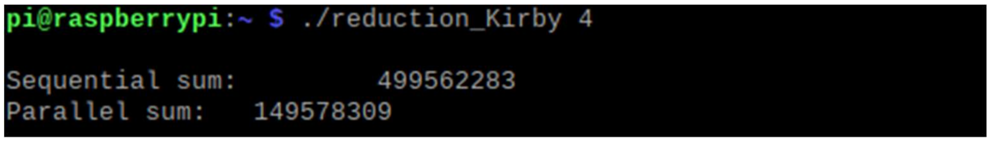
```
pi@raspberrypi:~ $ ./reduction_Kirby 3

Sequential sum:         499562283
Parallel sum:   499562283
```

<u>Reduction.c //#pragma omp parallel for // reduction(+:sum)-</u> When the line 39 code is ran with commenting the //#pragma omp parallel for // reduction(+:sum). The process is not forked so the "sequential" and "parallel" sums are ran the same way, with both being sequential so the difference in process and sum is not existant.

<u>Reduction.c #pragma omp parallel for // reduction(+:sum)- Screenshot</u>



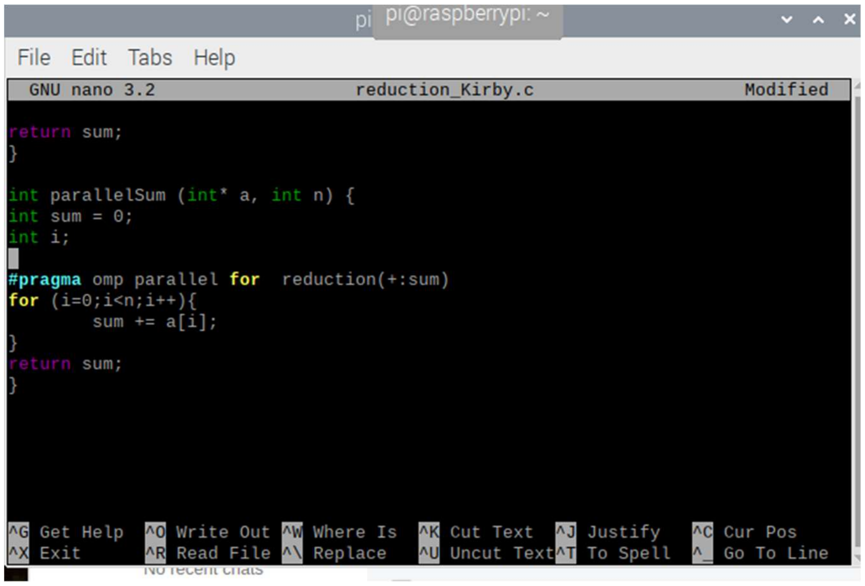
```
GNU nano 3.2                reduction_Kirby.c                 Modified

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;
}

^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
```



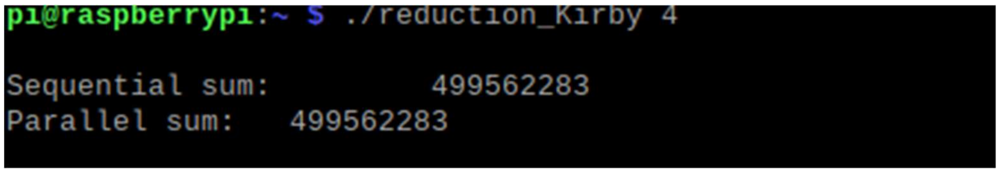
```
pi@raspberrypi:~ $ ./reduction_Kirby 4

Sequential sum:         499562283
Parallel sum:   149578309
```

<u>Reduction.c #pragma omp parallel for // reduction(+:sum)-</u> When the line 39 code is ran with commenting the #pragma omp parallel for // reduction(+:sum). The process is forked however the sums are not added up so the end result of "149578309" is the sum of just one of the thread and doesn't include the total sum of the multiple threads.

<u>Reduction.c //#pragma omp parallel for // reduction(+:sum)- Screenshot</u>



```
  GNU nano 3.2                  reduction_Kirby.c                    Modified

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;
}

^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
```

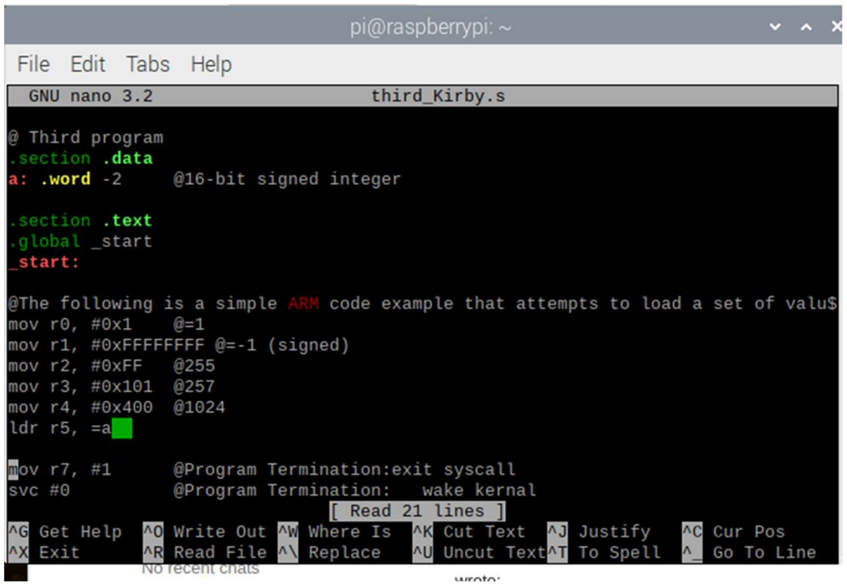


```
pi@raspberrypi:~ $ ./reduction_Kirby 4

Sequential sum:         499562283
Parallel sum:   499562283
```

<u>Reduction.c #pragma omp parallel for reduction(+:sum)-</u> When the line 39 code is ran with commenting the #pragma omp parallel for reduction(+:sum). The process is forked with the sums of the threads being added together so that the parallel sums is equal to the sequential sum.

<u>ARM Assembly Task</u>

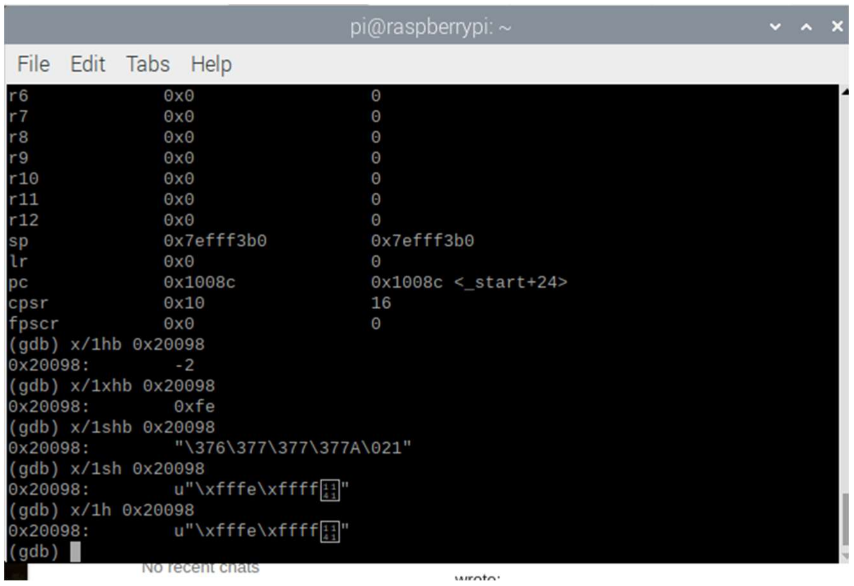
<u>Reduction.c-Screenshot</u>



```
 GNU nano 3.2                      third_Kirby.s

@ Third program
.section .data
a: .word -2     @16-bit signed integer

.section .text
.global _start
_start:

@The following is a simple ARM code example that attempts to load a set of valu$
mov r0, #0x1    @=1
mov r1, #0xFFFFFFFF @=-1 (signed)
mov r2, #0xFF   @255
mov r3, #0x101  @257
mov r4, #0x400  @1024
ldr r5, =a

mov r7, #1      @Program Termination:exit syscall
svc #0          @Program Termination:   wake kernal
                                [ 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
```

Reduction.c-Screenshot- in ARM Assembly the variable should be declared with .word instead of the shelfword as the ARM Assembly is default signed as there is no differentiation for signed or unsigned variable declaration. Additionally, in ARM 16-bit needs to be declared as a .word.

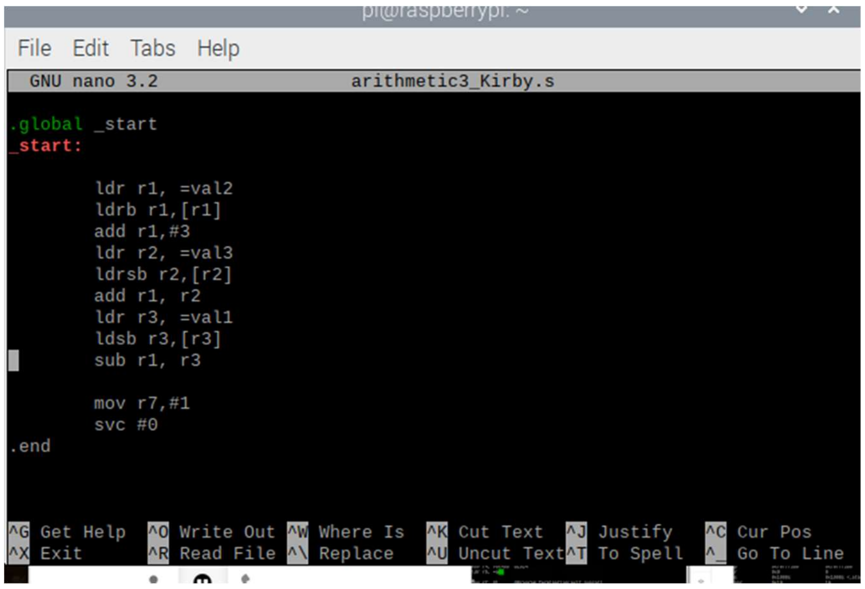
Reduction.c debugger- Screenshot



```
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/1hb 0x20098
0x20098:        -2
(gdb) x/1xhb 0x20098
0x20098:        0xfe
(gdb) x/1shb 0x20098
0x20098:        "\376\377\377\377A\021"
(gdb) x/1sh 0x20098
0x20098:        u"\xfffe\xffff"
(gdb) x/1h 0x20098
0x20098:        u"\xfffe\xffff"
(gdb)
```

Reduction.c debugger- When viewing the memory of where variable a is stored, you can see that the decimal value stored in a is –2. When you run it as a hex, the value is -2 in hex after 2's complement operation.

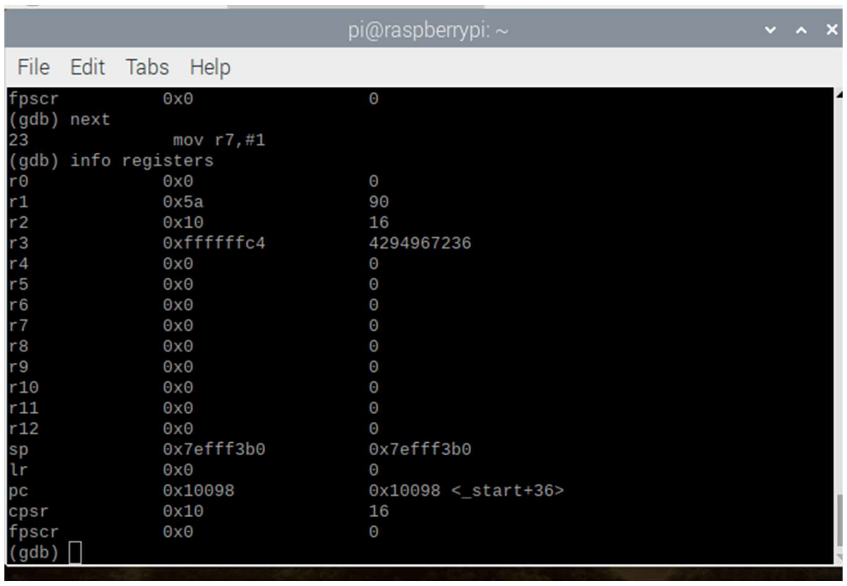
Arithmetic3.s-Screenshot



```
  GNU nano 3.2                   arithmetic3_Kirby.s

.global _start
_start:

        ldr r1, =val2
        ldrb r1,[r1]
        add r1,#3
        ldr r2, =val3
        ldrsb r2,[r2]
        add r1, r2
        ldr r3, =val1
        ldsb r3,[r3]
        sub r1, r3

        mov r7,#1
        svc #0
.end

^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
```

<u>Arithmetic3.s-</u> The following is how I programmed the exercise of Arithmetic3 code and shows the importance of correct loading of signed or unsigned using ldrsb or ldrb of the variables into the registers.

<u>Debugging Arithmetic3.s-Screenshot</u>



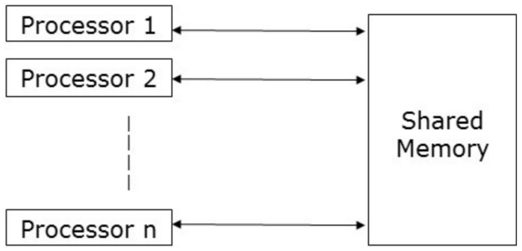
```
fpscr          0x0                 0
(gdb) next
23              mov r7,#1
(gdb) info registers
r0             0x0                 0
r1             0x5a                90
r2             0x10                16
r3             0xffffffc4          4294967236
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10098             0x10098 <_start+36>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

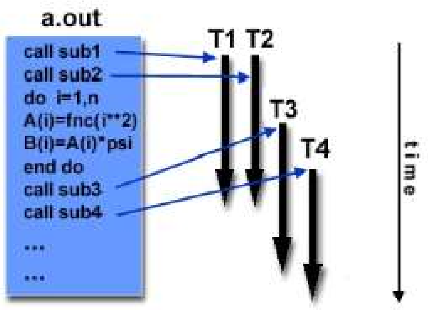
<u>Debugging Arithmetic3.s-</u> When running the debugger, the info register indicates the total of the vall2+3+val3-val1 stored in register r1 is 90. You see where val1,-60, is stored as a hex ffffffc4, which the 2 complement will result in –60. The info register of the register value won't show it as a negative or positive decimal, but when the register r1 shows 90, it indicates the correct addition/subtraction.

Humaira Ridi

Parallel Programming Foundation

1. Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)
    - A **task** is either a program or similar to a program that a processor can execute.
    - **Pipelining** is a type of parallel computing that breaks a task down into multiple steps which are then executed by different processor units.
    - **Shared memory**, in regards to hardware, is when memory is directly accessible to all processors. Regarding programming, however, shared memory is when tasks can access the address of logical memory locations, no matter where the physical memory is located.
    - **Communications** is the term that defines the exchange of data between parallel tasks.
    - **Synchronization** refers to the organization of parallel tasks.
2. Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.
    - **Single Instruction, Single Data (SISD)** involves a sole instruction performed by the CPU during a single clock cycle while a single data is used as input.
    - **Single Instruction, Multiple Data (SIMD)** is a parallel type of computer where the same instruction is executed by all the processing units during a clock cycle and each processing unit is able to function on different data elements.
    - **Multiple Instruction, Single Data (MISD)** is where separate instruction streams allow for independent processing unit operation and each individual data stream is being loaded into multiple processing units.
    - **Multiple Instruction, Multiple Data (MIMD)** is a parallel type of computer that allows all processors to execute a different instruction stream and data stream.
3. 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
4. List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?
    - Uniform Memory Access (UMA) refers to the equal access and access types. OpenMP uses UMA.
    - Non-Uniform Memory Access (NUMA) does not allow for equal access time to memories for all processors.
5. Compare Shared Memory Model with Threads Model? (in your own words and show pictures)

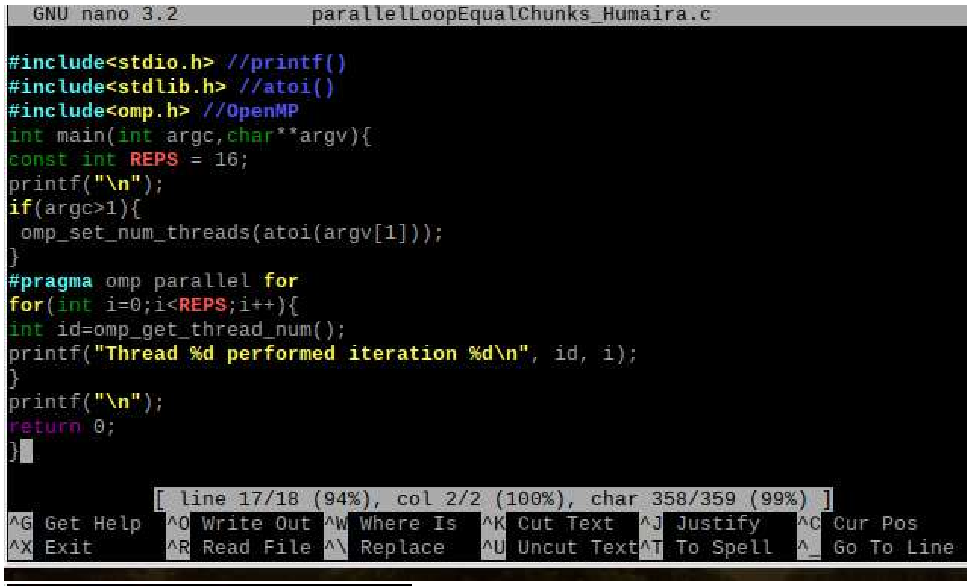
- In a **Shared Memory Model**, the simplest parallel programing model, processes are located in the same address in the memory that they can read and write to at different times.
- 

- In a **Threads Model**, one process that is considered the main program possesses multiple execution paths. This main program creates threads that are executed by the thread simultaneously.
- 

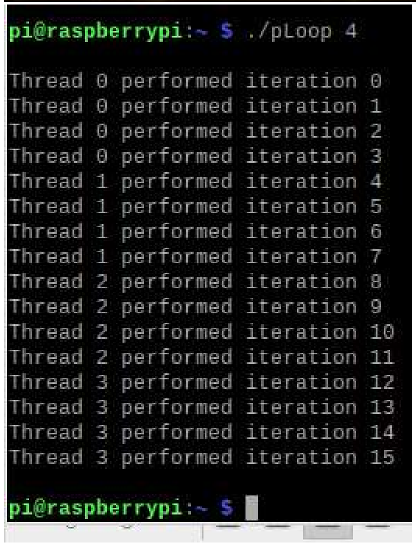
6. What is Parallel Programming? (in your own words)
    - Parallel programming is the solving of a problem using more than one resources. The problem is separated into different parts to be solved simultaneously.
7. What is system on chip (SoC)? Does Raspberry PI use system on SoC?
    - An **SoC** combines components of a computer into one silicon chip; Yes.
8. Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.
    - The size of an SoC is its greatest advantage. It is a lot more efficient than a CPU despite only being slightly larger than one.

Humaira Ridi - Parallel Programming Basics

1. **ParallelLoopEqualChunks.c**



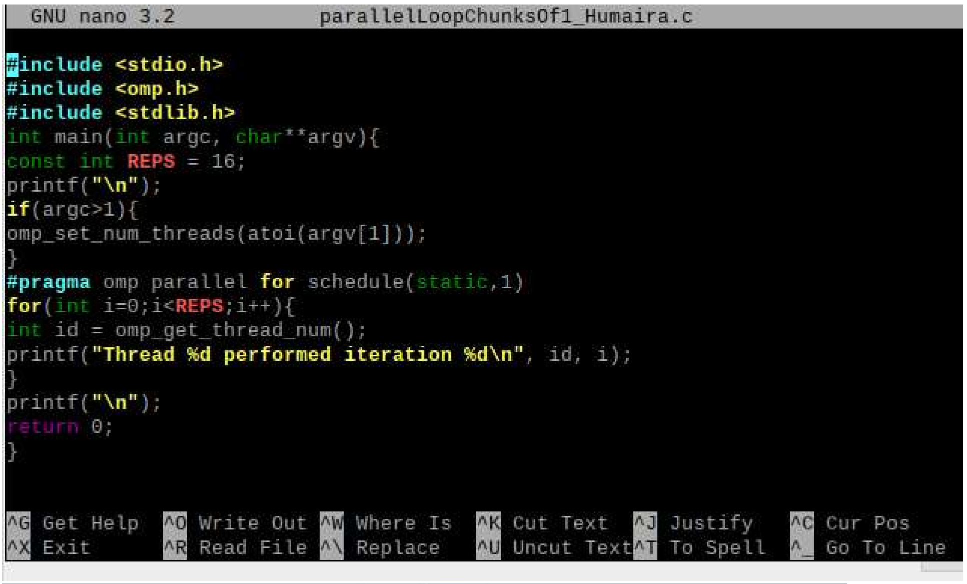
```
  GNU nano 3.2              parallelLoopEqualChunks_Humaira.c

#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;
}

            [ line 17/18 (94%), col 2/2 (100%), char 358/359 (99%) ]
^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:~ $ ./pLoop 4

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
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

pi@raspberrypi:~ $
```

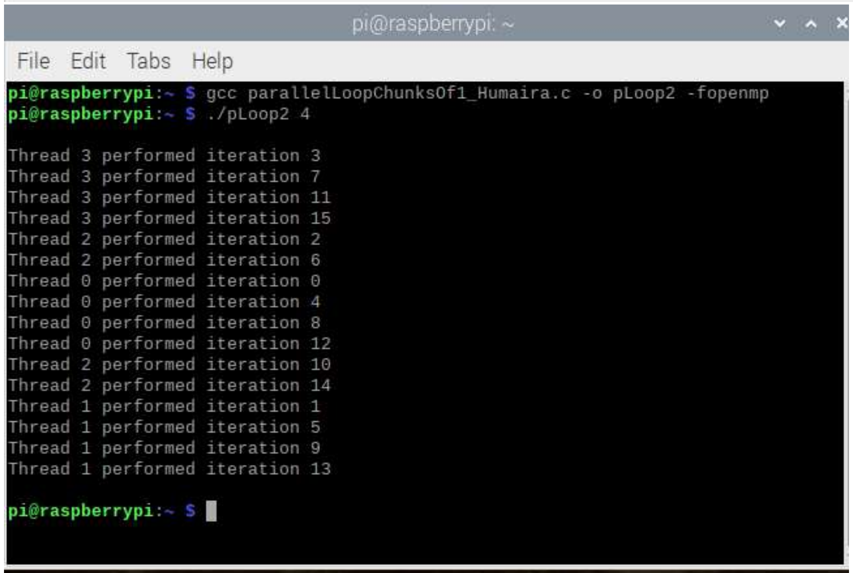
In ParallelLoopEqualChunks.c, we are running a loop in OpenMP with the omp parallel for pragma. There are 16 iterations with 4 threads to fork. When replacing 4 with other values not divisible by 16, thread 0 will ultimately have the most iterations. This division of work into different iterations of a loop is defined as working on chunks of the work.

2. **ParallelLoopChunksOf1.c**

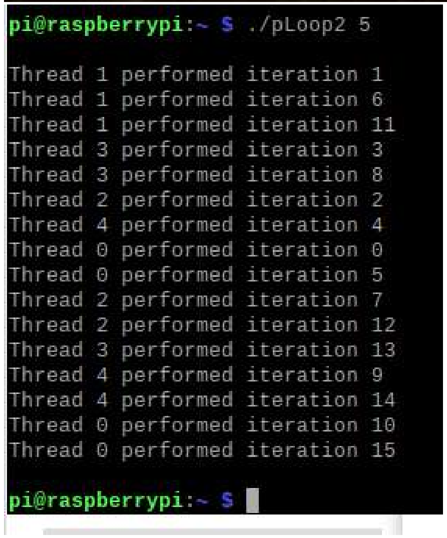


```
GNU nano 3.2                parallelLoopChunksOf1_Humaira.c

#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++){
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n", id, i);
}
printf("\n");
return 0;
}
```

^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



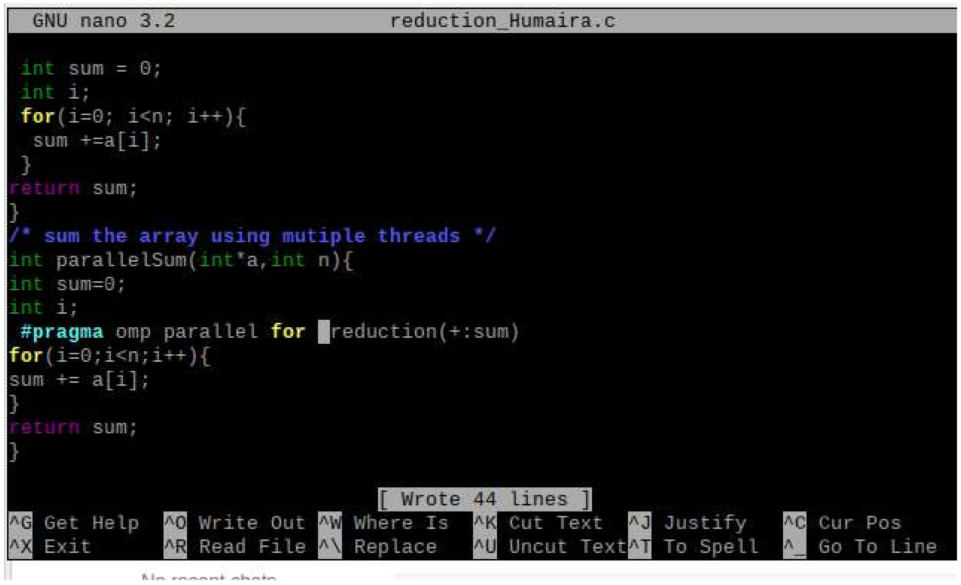
```
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1_Humaira.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4

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 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 2 performed iteration 10
Thread 2 performed iteration 14
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13

pi@raspberrypi:~ $
```



```
pi@raspberrypi:~ $ ./pLoop2 5

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

pi@raspberrypi:~ $
```

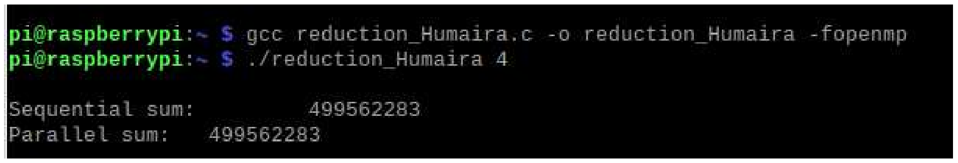
ParallelLoopChunksOf1.c is another way to have each thread perform however many iterations of a loop, where one iteration is tied to one thread, the next iteration would be tied to the next thread, and it would go on until the end of the loop. In this case, threads do not have consecutive iterations.

3. **Reduction.c**



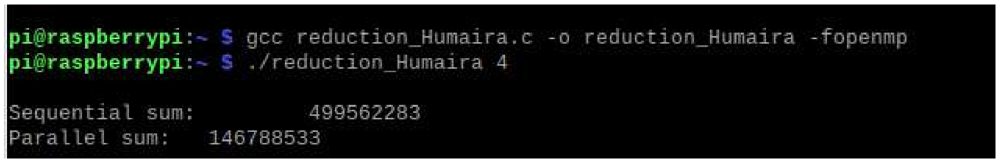
```
  GNU nano 3.2                    reduction_Humaira.c

 int sum = 0;
 int i;
 for(i=0; i<n; i++){
  sum +=a[i];
 }
return sum;
}
/* sum the array using mutiple threads */
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;
}

                               [ Wrote 44 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
```

In reduction.c, the sum of an array of randomly assigned integers is being printed. When “#pragma omp parallel for reduction(+:sum)” in line 39 was originally commented out, the program was not being run in parallel with threads. This was the result:



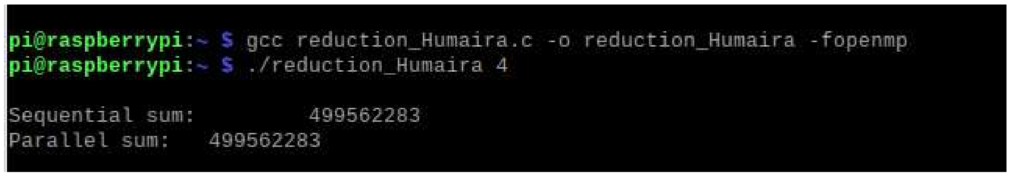
```
pi@raspberrypi:~ $ gcc reduction_Humaira.c -o reduction_Humaira -fopenmp
pi@raspberrypi:~ $ ./reduction_Humaira 4

Sequential sum:         499562283
Parallel sum:   499562283
```

When only “reduction(+:sum)” was commented out from line 39, the sequentialSum result differed from the parallelSum result:



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

Sequential sum:         499562283
Parallel sum:   146788533
```

After removing all // from line 39, allowing for the code to run in parallel, the results of sequentialSum and parallelSum were the same once again:

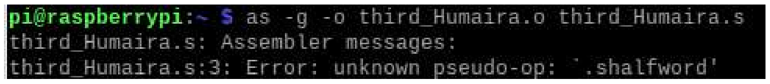


```
pi@raspberrypi:~ $ gcc reduction_Humaira.c -o reduction_Humaira -fopenmp
pi@raspberrypi:~ $ ./reduction_Humaira 4

Sequential sum:         499562283
Parallel sum:   499562283
```

<u>**Humaira Ridi – ARM Assembly**</u>

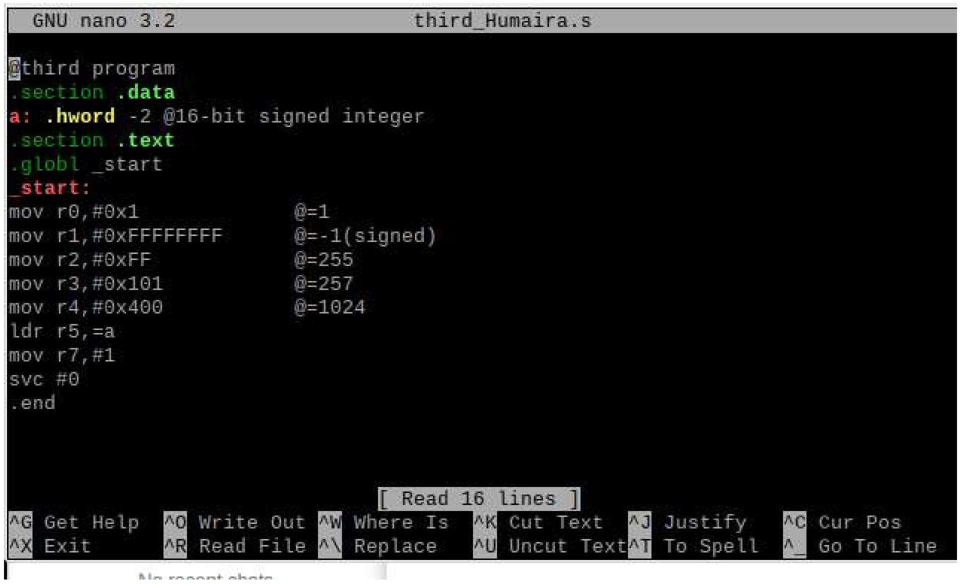
Part 1:

When we attempted to run the original code we were given, we received an error message claiming the memory size that we used was not valid:



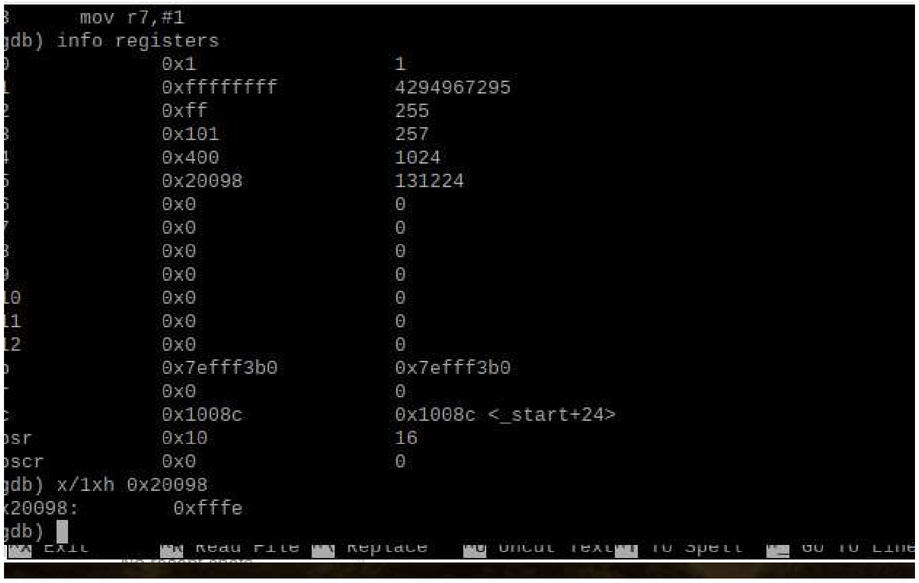
```
pi@raspberrypi:~ $ as -g -o third_Humaira.o third_Humaira.s
third_Humaira.s: Assembler messages:
third_Humaira.s:3: Error: unknown pseudo-op: `.shalfword'
```

To fix this error, .shalfword was changed to .hword to fit 16-bit integers:



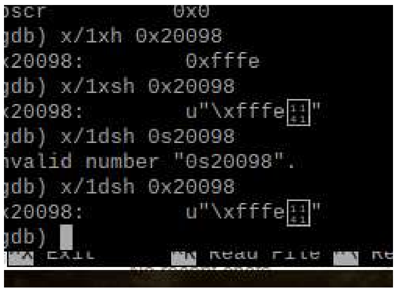
```
  GNU nano 3.2                         third_Humaira.s

@third program
.section .data
a: .hword -2 @16-bit signed integer
.section .text
.globl _start
_start:
mov r0,#0x1             @=1
mov r1,#0xFFFFFFFF      @=-1(signed)
mov r2,#0xFF            @=255
mov r3,#0x101           @=257
mov r4,#0x400           @=1024
ldr r5,=a
mov r7,#1
svc #0
.end


                              [ Read 16 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
```

After displaying the registers, the address and the value stored were both visible.



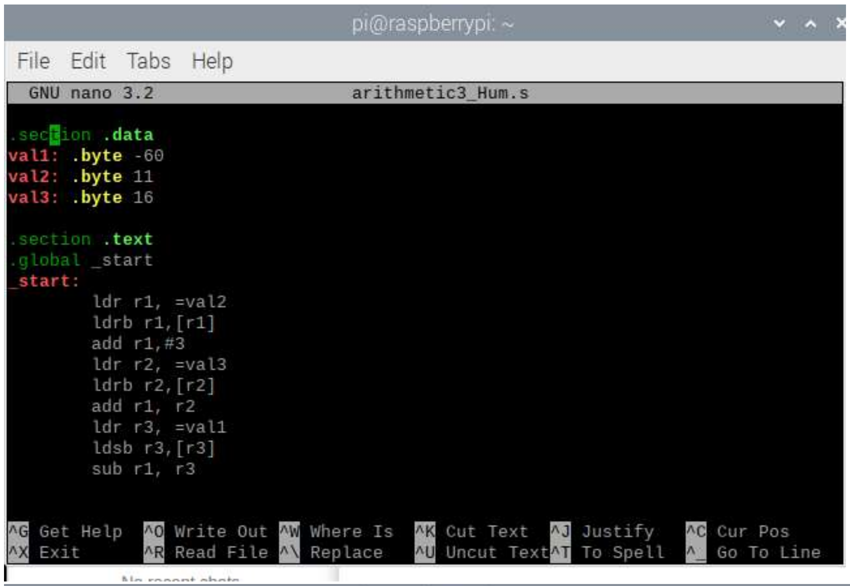
```
3      mov r7,#1
gdb) info registers
0              0x1                 1
1              0xffffffff          4294967295
2              0xff                255
3              0x101               257
4              0x400               1024
5              0x20098             131224
6              0x0                 0
7              0x0                 0
8              0x0                 0
9              0x0                 0
10             0x0                 0
11             0x0                 0
12             0x0                 0
p              0x7efff3b0          0x7efff3b0
r              0x0                 0
c              0x1008c             0x1008c <_start+24>
psr            0x10                16
pscr           0x0                 0
gdb) x/1xh 0x20098
x20098:        0xfffe
gdb)
```

To examine the memory at r5, I used the command x/1xsh 0x20098.

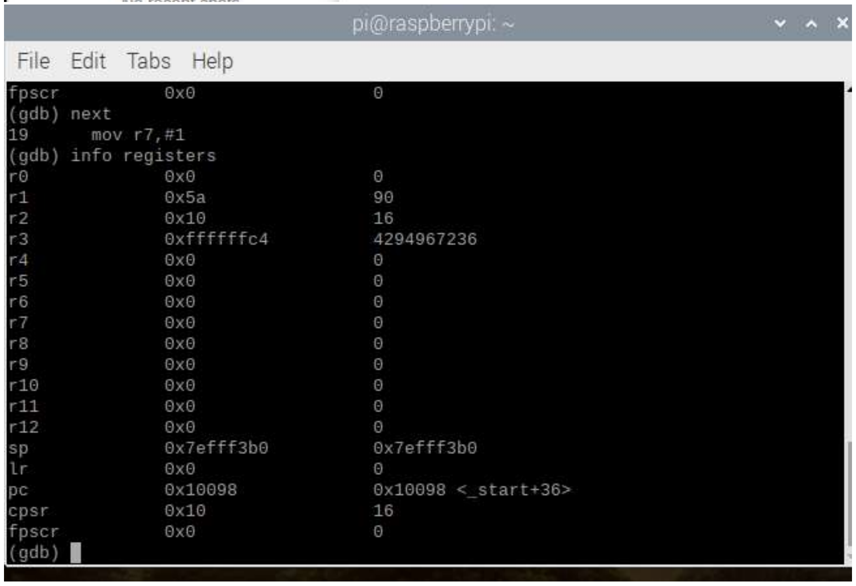


```
pscr           0x0
gdb) x/1xh 0x20098
x20098:        0xfffe
gdb) x/1xsh 0x20098
x20098:        u"\xfffe"
gdb) x/1dsh 0s20098
nvalid number "0s20098".
gdb) x/1dsh 0x20098
x20098:        u"\xfffe"
gdb) 
```

Part 2:



```
  GNU nano 3.2                   arithmetic3_Hum.s

.section .data
val1: .byte -60
val2: .byte 11
val3: .byte 16

.section .text
.global _start
_start:
        ldr r1, =val2
        ldrb r1,[r1]
        add r1,#3
        ldr r2, =val3
        ldrb r2,[r2]
        add r1, r2
        ldr r3, =val1
        ldsb r3,[r3]
        sub r1, r3
```



```
fpscr          0x0                 0
(gdb) next
19        mov r7,#1
(gdb) info registers
r0             0x0                 0
r1             0x5a                90
r2             0x10                16
r3             0xffffffc4          4294967236
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10098             0x10098 <_start+36>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) 
```

For arithmetic3.s, we were asked to find the value of val2 + 3 + val3 - val1 and store it in a register, where val1 = -60, val2 = 11, val3 = 16. Since we were dealing with Bytes, we had to use ldrb to load unsigned bytes into a register and ldrsb to load signed bytes. After storing the result in register r1, the value was 90.

Dhruv Parikh

Parallel Programming Foundation

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

- Task is a basic unit of programming that an operating system control. Depending on how the operating system defines a task in its design, this unit of programming may be an entire program or each successive invocation of a program.
- Pipelining is the continuous and somewhat overlapped movement of instruction to the processor or in the arithmetic steps taken by processor to perform an instruction.'
- Shared memory is a memory that can be accessed many programs with an intent to provide communication among them or avoid redundant copies.
- Communications can be the data exchange which is needed by parallel tasks. It can be done in several ways such as through memory bus or by a network.
- Synchronization -

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

- Single Instruction, Single Data (SISD): 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 unit execute the same instruction at any given clock cycles. Each processing unit operate on different data element.
- Multiple Instruction, Single Data (MISD): Each processing unit operates on the data independently via separate instruction streams. A single data stream is fed into multiple processing units.
- Multiple Instruction, Multiple Data (MIMD): Every processor may be executing a different instruction stream. Every processor may be working with a different data stream.

3. **What are the Parallel Programming Models?**

- A Parallel Programming Models is a set of program abstractions for fitting parallel activities from the application to the underlying parallel hardware.

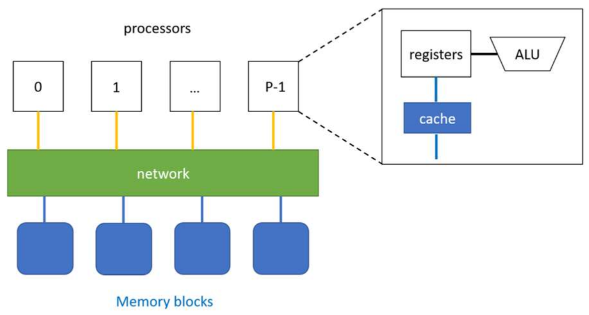
4. **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**

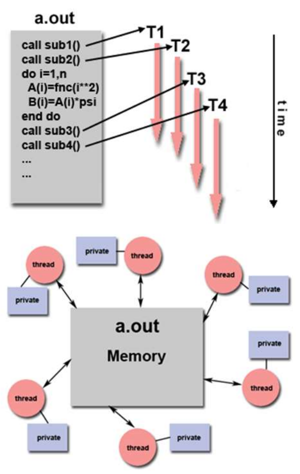
- Uniform Memory Access (UMA):
    - Most represented today by Symmetric Multiprocessor (SMP) machines

- Identical processors
    - Equal access and access time to memory
    - Sometimes call CC - UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- Non – Uniform memory Access (NUMA):
    - Often made by physically linking two or more SMPs
    - One SMP can directly access memory
    - Not all processors have equal access time to all memories
    - Memory access across link is slower
    - If cache coherency is maintained, then may alo be called CC-NUMA- Cache Coherent NUMA
- OpenMp is a library for parallel programming in the SMP model. When programming with OpenMp all threads share memory and data. OpenMP supports C, C++ and Fortran. The OpenMP functions are included in a header file called omp.

5. Compare Shared Memory Model with Threads Model?

- Shared memory model is the memory that can simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. Windows operating system use shared memory.
- Threads Model is the unit of execution within a process. A process can have anywhere from just one thread to many threads.





```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```

–

6. What is Parallel Programming?

- Parallel Programming model is an abstraction of parallel computing architecture, with which it is convenient to express algorithms and their composition in programs.

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

- An SoC, or system-on-a-chip to give its full name, integrates almost all of these components into a single silicon chip.

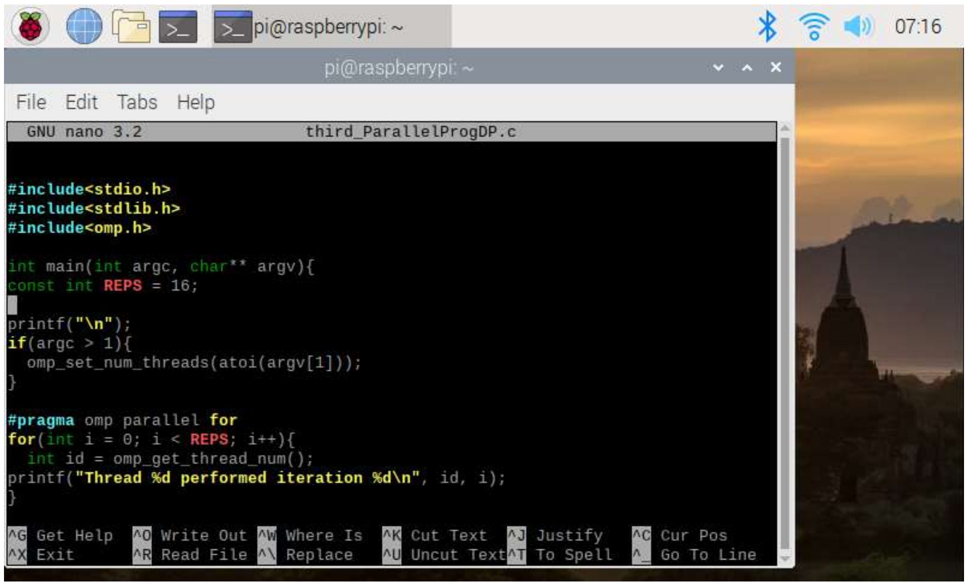
- Several generations of Raspberry Pis have been released. All models feature a Broadcom system on a chip (SoC) with an integrated ARM-compatible central processing unit (CPU) and on-chip graphics processing unit (GPU).

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

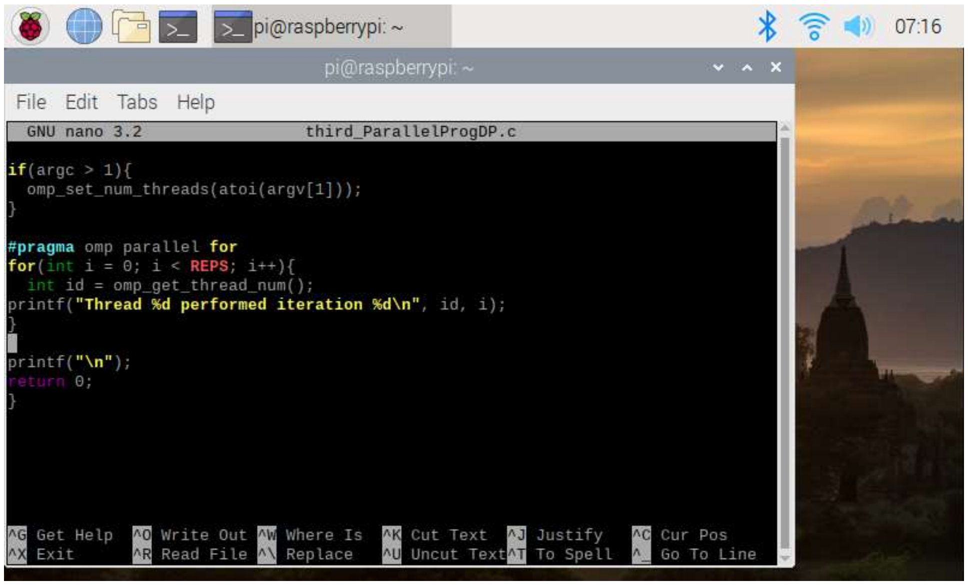
- The number one advantage of an SoC is its size: An SoC is only a little bit larger than a CPU, and yet it contains a lot more functionality. If you use a CPU, it's very hard to make a computer that's smaller than 10cm (4 inches) squared, purely because of the number of individual chips that you need to squeeze in. Using SoCs, we can put complete computers in smartphones and tablets, and still have plenty of space for Batteries.

<u>Parallel Programming Task: -</u>

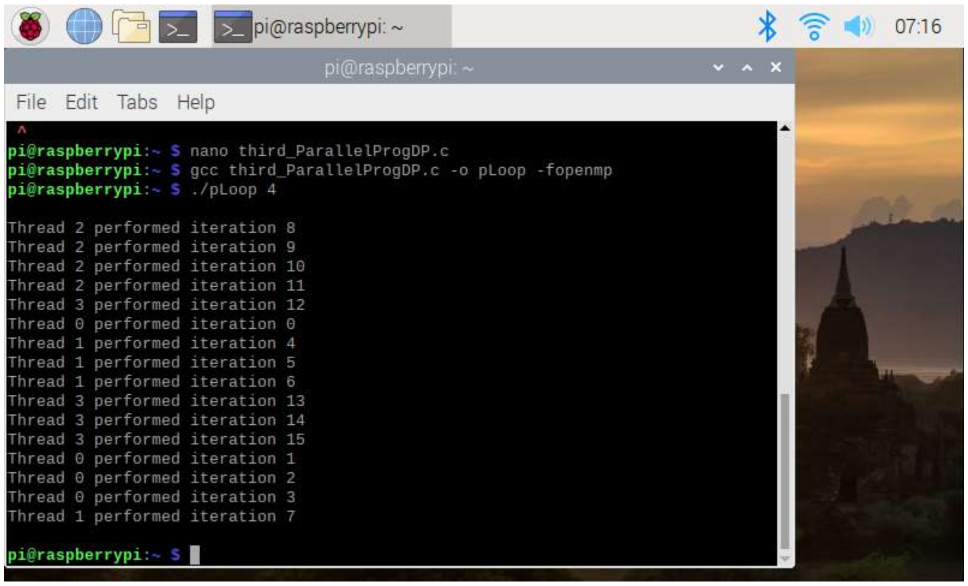
<u>ParallelLoopChunksOfl.c</u>



```
  GNU nano 3.2                    third_ParallelProgDP.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);
}

^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
```



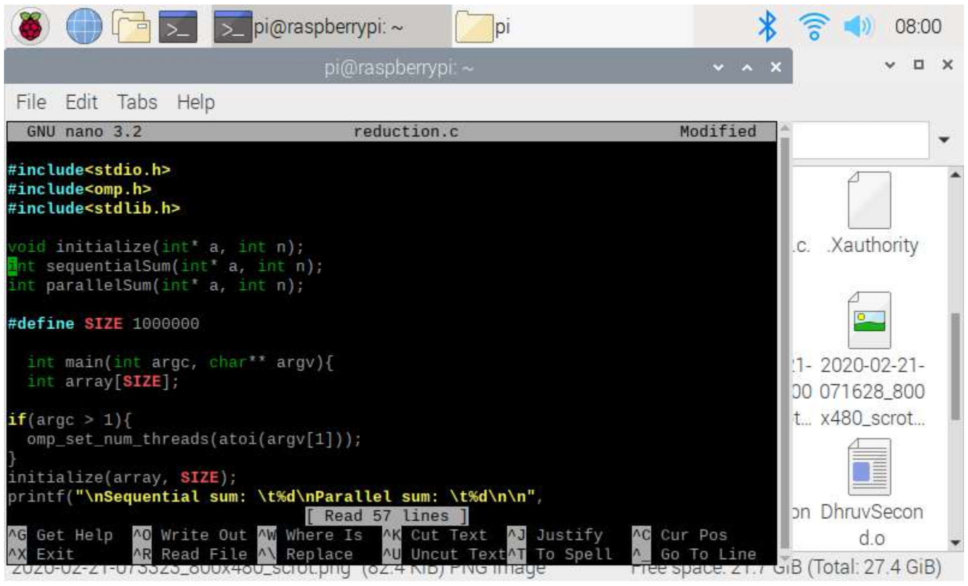
```
  GNU nano 3.2                    third_ParallelProgDP.c

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;
}

^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
```



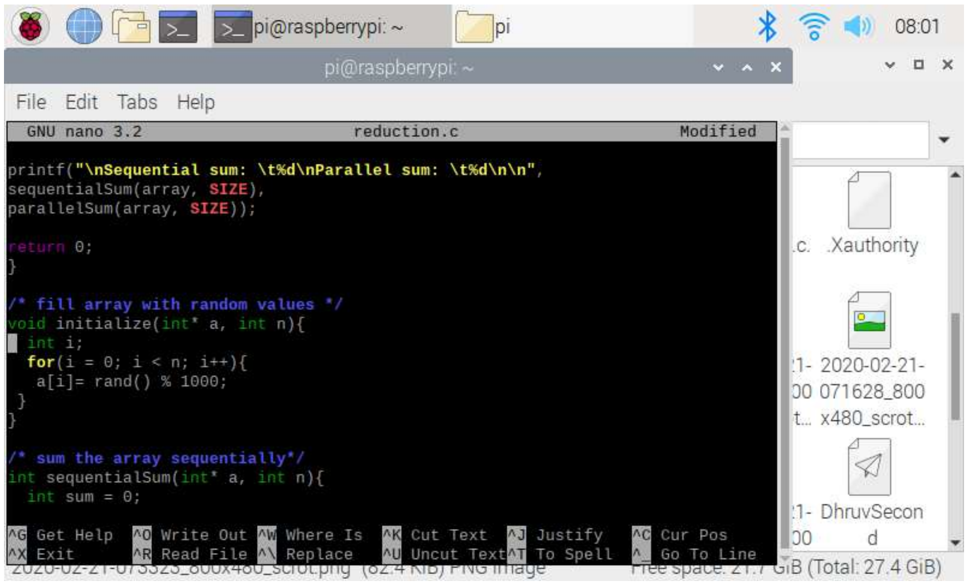
```
pi@raspberrypi:~ $ nano third_ParallelProgDP.c
pi@raspberrypi:~ $ gcc third_ParallelProgDP.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

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 0 performed iteration 0
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 3 performed iteration 15
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 1 performed iteration 7

pi@raspberrypi:~ $
```

In this code you can see that the threads are not in order and they performed the iteration in scrambled order. Each thread is performed four times with different iterations.

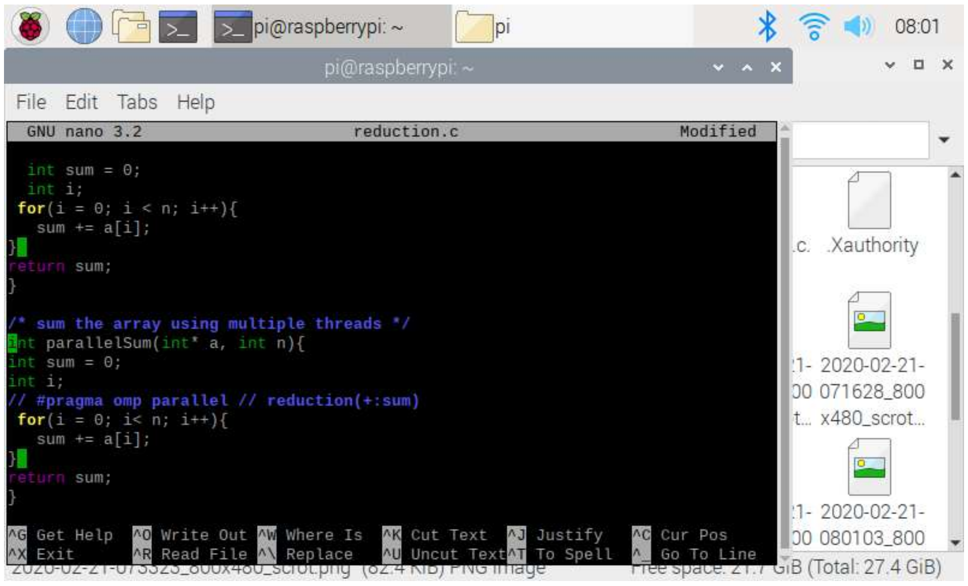
<u>Reduction.c</u>



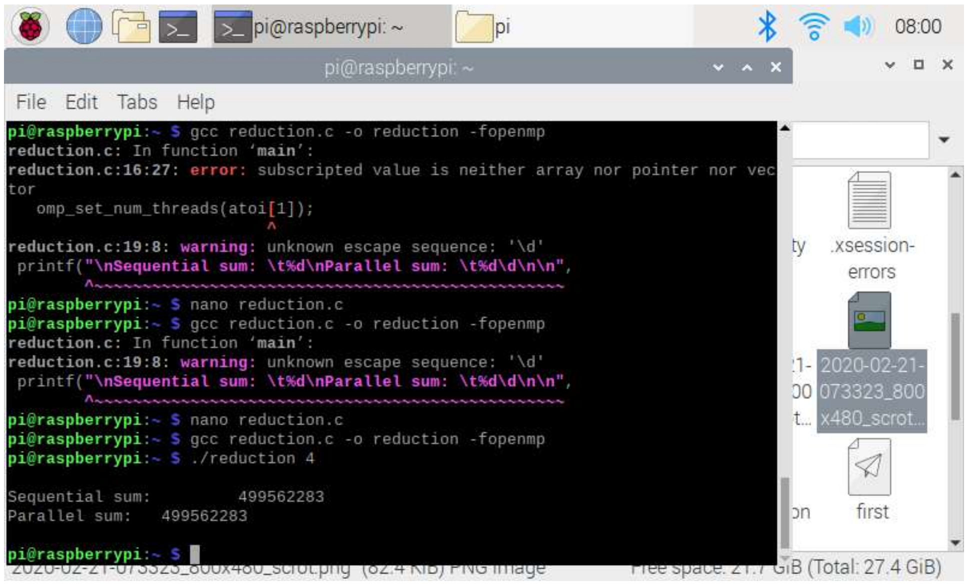
```
  GNU nano 3.2                     reduction.c                     Modified

#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 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",
                              [ Read 57 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
```



```
  GNU nano 3.2                     reduction.c                     Modified

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;
  for(i = 0; i < n; i++){
   a[i]= rand() % 1000;
 }
}

/* sum the array sequentially*/
int sequentialSum(int* a, int n){
  int sum = 0;

^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
```



```
  GNU nano 3.2                   reduction.c                    Modified

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

/* sum the array using multiple threads */
int parallelSum(int* a, int n){
int sum = 0;
int i;
// #pragma omp parallel // reduction(+:sum)
 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
```

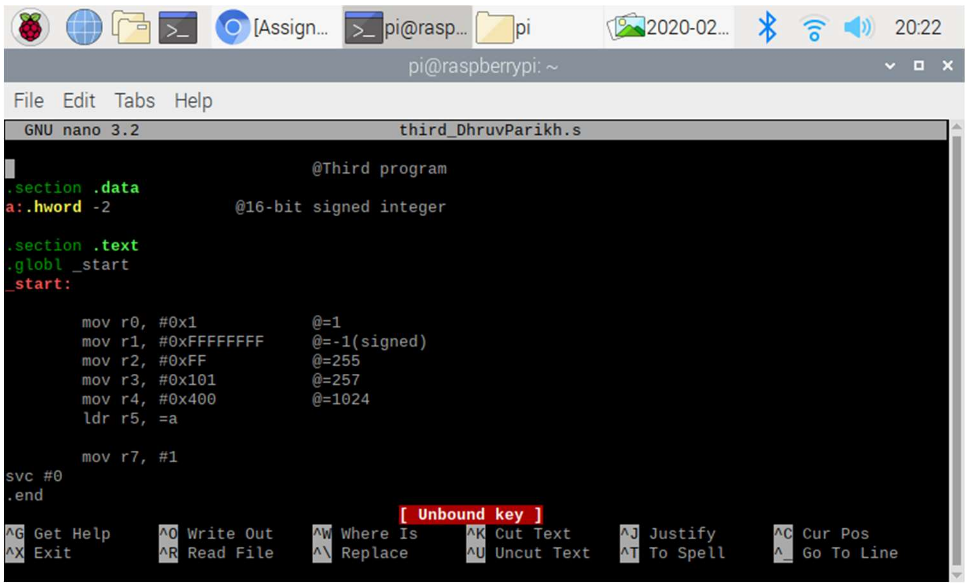


```
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
reduction.c: In function ‘main’:
reduction.c:16:27: error: subscripted value is neither array nor pointer nor vec
tor
   omp_set_num_threads(atoi[1]);
                           ^
reduction.c:19:8: warning: unknown escape sequence: '\d'
 printf("\nSequential sum: \t%d\nParallel sum: \t%d\d\n\n",
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
reduction.c: In function ‘main’:
reduction.c:19:8: warning: unknown escape sequence: '\d'
 printf("\nSequential sum: \t%d\nParallel sum: \t%d\d\n\n",
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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:~ $ 
```

The Sequential sum and Parallel sum are not same because we did not remove the comment and the code is not forked.

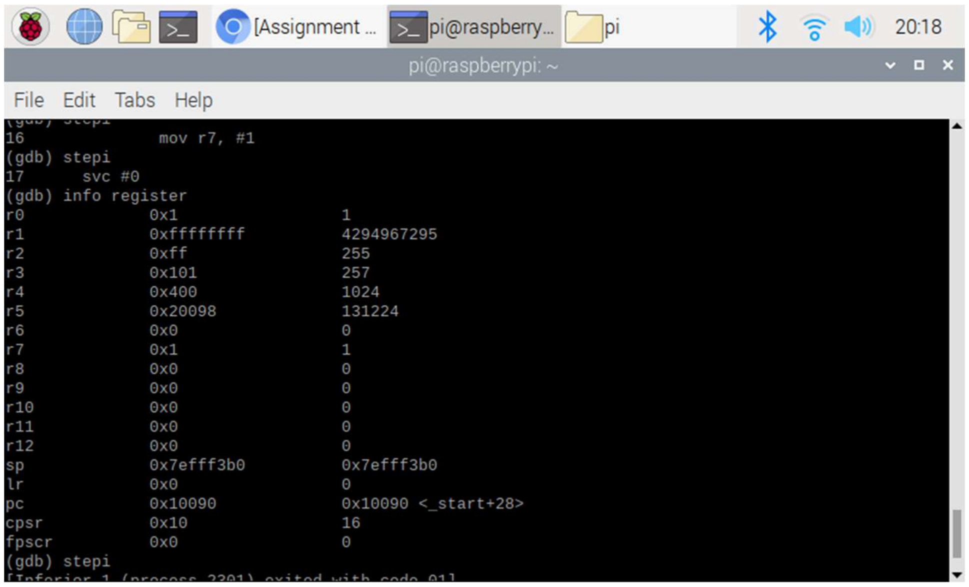
When the Sequential sum and Parallel sum are the same that means we have removed the commented part.

<u>Third.s</u>



```
  GNU nano 3.2                                    third_DhruvParikh.s

                                        @Third program
.section .data
a:.hword -2                   @16-bit signed integer

.section .text
.globl _start
_start:

          mov r0, #0x1          @=1
          mov r1, #0xFFFFFFFF   @=-1(signed)
          mov r2, #0xFF         @=255
          mov r3, #0x101        @=257
          mov r4, #0x400        @=1024
          ldr r5, =a

          mov r7, #1
svc #0
.end
                                     [ Unbound key ]
^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
```

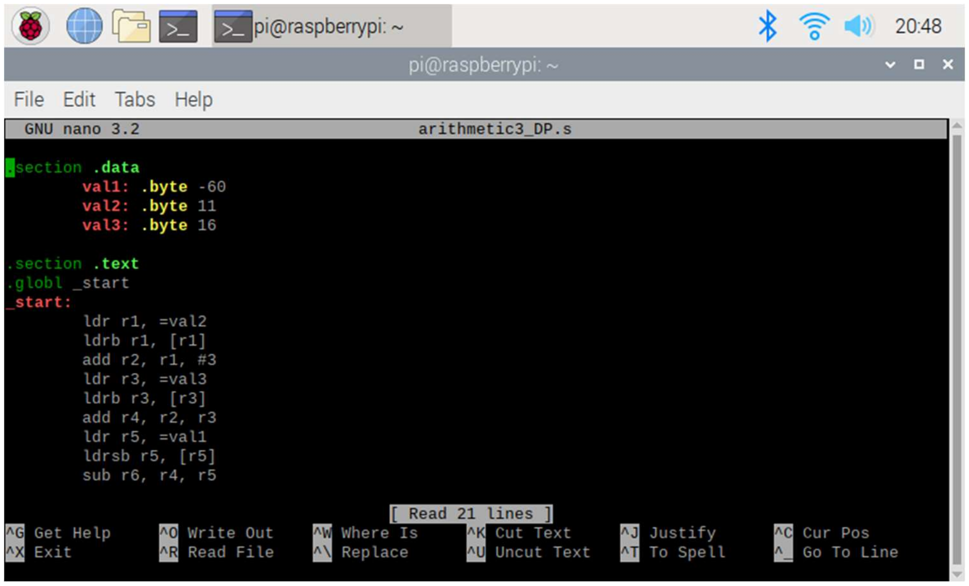
The error is there because there is “shalfword” doesn’t mean anything in ARM Assembly. So we have to replace that word with “hword”(half word). We do not have to declare a value signed or unsigned when we load into register.



```
16              mov r7, #1
(gdb) stepi
17          svc #0
(gdb) info register
r0             0x1                 1
r1             0xffffffff          4294967295
r2             0xff                255
r3             0x101               257
r4             0x400               1024
r5             0x20098             131224
r6             0x0                 0
r7             0x1                 1
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10090             0x10090 <_start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
```

In r1 when you see “0xffffffff” it means the value id a signed value because the most significant bit is “f”. If the most significant bit is “0” than the value is unsigned value.

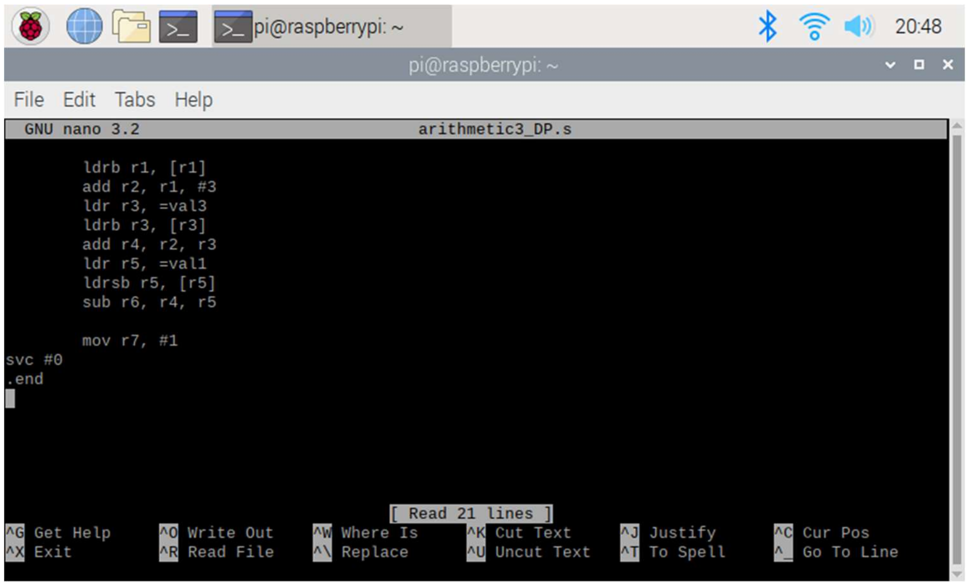
<u>Arithmetic.s</u>



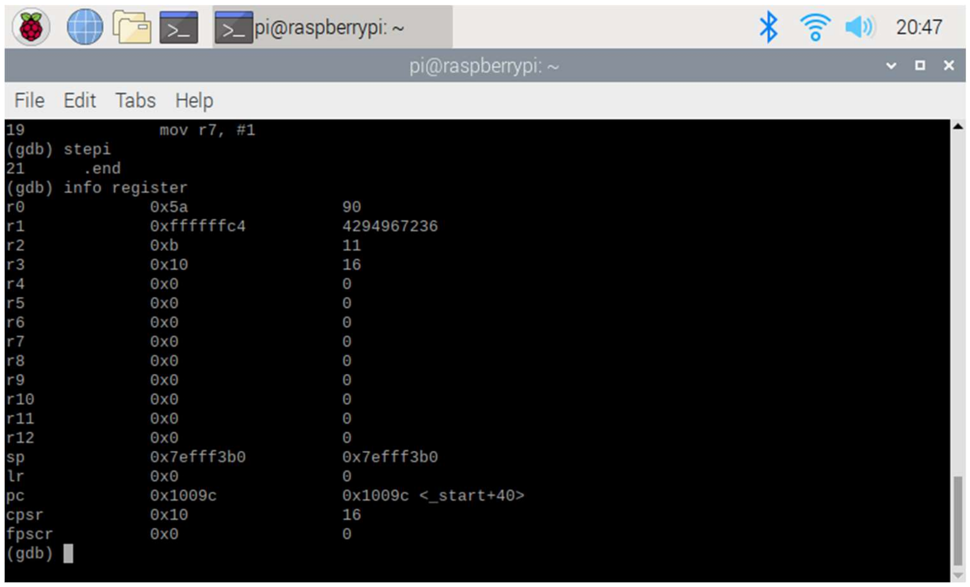
```
  GNU nano 3.2                                  arithmetic3_DP.s

.section .data
        val1: .byte -60
        val2: .byte 11
        val3: .byte 16

.section .text
.globl _start
_start:
        ldr r1, =val2
        ldrb r1, [r1]
        add r2, r1, #3
        ldr r3, =val3
        ldrb r3, [r3]
        add r4, r2, r3
        ldr r5, =val1
        ldrsb r5, [r5]
        sub r6, r4, r5

                                       [ 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
```

We store val1, val2, and val3 in “byte” because they are 8-bit integers.

We have loaded val2 in r1, val3 in r3, and val1 in r5. As val1 is –60 we must load the value using “ldrsb”. And val2 and val3 will be loaded normally as they are unsigned and signed values respectively.



```
  GNU nano 3.2                                  arithmetic3_DP.s

        ldrb r1, [r1]
        add r2, r1, #3
        ldr r3, =val3
        ldrb r3, [r3]
        add r4, r2, r3
        ldr r5, =val1
        ldrsb r5, [r5]
        sub r6, r4, r5

        mov r7, #1
svc #0
.end


                                       [ 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
```



```
19              mov r7, #1
(gdb) stepi
21      .end
(gdb) info register
r0             0x5a                90
r1             0xffffffc4          4294967236
r2             0xb                 11
r3             0x10                16
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x1009c             0x1009c <_start+40>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

This are registers for the result of register = val2 + 3 + val3 – val1.

11 + 3 + 16 - (-60) = 90.

Task 5

Github: https://github.com/Team-0101/CSC3210-Team0101

Slack: https://app.slack.com/client/TT7TS0KN2/CUDK36RKN

Video (Youtube link): https://youtu.be/1Ab5EyVle9s