Developing Soft and Parallel Programming Skills Using Project - Based Learning

Spring 2019

A Wad Winning Group

Group Members:

Kishan Bhakta
Shilp Patel
Sean Staley
Mindy Tran
Donald Weaver

Planning and Scheduling:

For the third project, we had an easy going with planning meetings, communicating and assigning the task to each remember. For this project we only had two meetings, one to assign tasks and second meeting was organized to discuss the details for project on a video presentation. And to communicate we used online platform such as GitHub and Slack a lot to keep each other updated. Here is the breakdown of our schedule for this project.

Name	Email	Task	Duration (hours)	Dependency	Due Date	Note
Shilp Patel (coordinator)	Spatel255@student.gsu.edu	Report	5	Slack, Table, Report.	3/08/19	Remember to print the report and submit paper to teacher on Monday.
Sean Staley	sstaley4@student.gsu.edu	Foundation of parallel programming skills	3	Lead the meeting for answering the questions in Parallel Skills.	3/08/19	Make sure everyone understands the presentation and explain it efficiently.
Mindy Tran	mtran42@student.gsu.edu	Parallel programming	2.5	Parallel programming lab report.	3/08/19	Take screenshots.
Kishan Bhakta	kbhakta1@student.gsu.edu	Video	.5	Uploading the video to YouTube, video editing, GitHub.	3/08/19	Make sure video is within the time limit and good quality.
Donald Weaver	Dweaver20@student.gsu.edu	ARM assembly program	3	ARM assembly program lab report.	3/08/19	Take screenshots.

Parallel Programming Skills:

Foundation:

Task – A basic set of programming instructions that are carried out by the processor.

Pipelining – The continuous set of instructions fed into the processor, or the arithmetic steps taken by the processor to perform an instruction.

Shared Memory – Memory that can be accessed simultaneously by multiple programs. Avoids redundant copies, provides communication between processors, and passes data between programs.

Communications – Transferring data from one computer to another.

Synchronization – Multiple processors meeting up at a point to determine a sequence of action in the program.

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

Single Instruction, Single Data (SISD): CPU is being fed only one instruction at a time. Each clock cycle is waiting on the CPU to finish the one and only task being fed to it from the data stream.

Single Instruction, Multiple Data (SIMD): Each processing unit is taking on the same task at any given time during a clock cycle. Each processing unit can operate on different data elements however.

Multiple Instruction, Single Data (MISD): A single data stream is fed into multiple processing units that each process the data independently using separate instruction streams.

Multiple Instruction, Multiple Data (MIMD): All processors can work independently with different data streams and different instruction streams either synchronously, asynchronously, deterministically, or non-deterministically.

What are the Parallel Programming Models?

A parallel programming model is an abstraction of parallel computing architecture making it easier to express algorithms and their composite programs. The parallel programming models are process interaction and problem decomposition.
Process interaction includes: shared memory, message passing and implicit interaction.
Problem decomposition includes: task parallelism, data parallelism, and implicit parallelism.

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

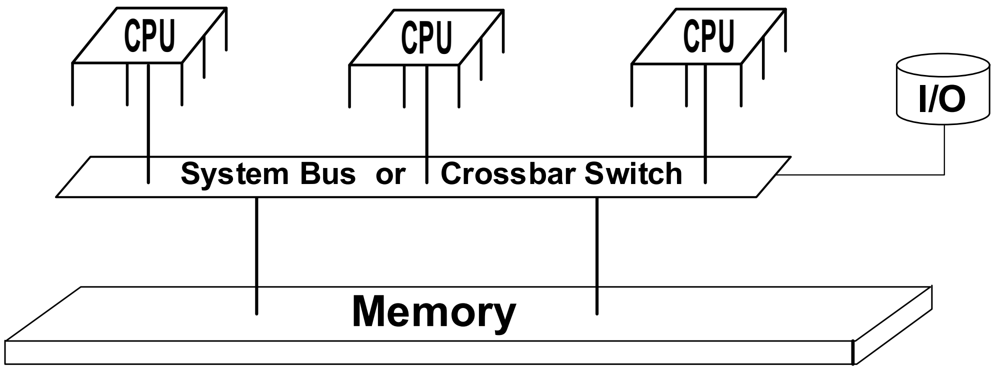
Uniform Memory Access (UMA): All processors uniformly share the physical memory. Access time is independent to which processor makes the request or which memory chip contains the transferred data.

Non-Uniform Memory Access (NUMA): Memory access time depends on memory location relative to the processor. A processor can access its own local memory faster that non local memory, such as memory local to another processor or memory shared between processors.

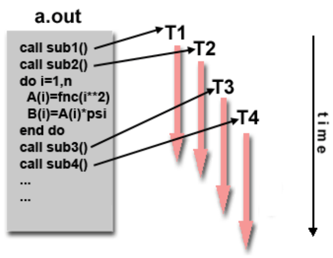
Open MP uses NUMA so that it can split the work up among threads using its Fork-Join model of program execution.

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

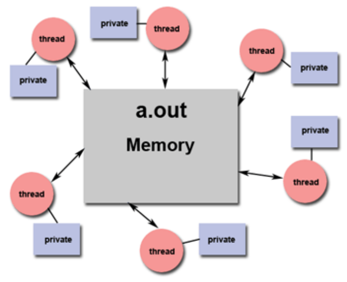
Shared memory is memory that can be accessed at the same time by multiple processors. This is likely to reduce or avoid redundant copies, and provides communication between them. Programs using shared memory can run on a single processor or on multiple shared processors.



By Khazadum, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3465954



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



In the threads model, multiple threads can be executed at the same time by multiple processors.

Threads exist as subsets of a process, where processes in the shared memory model are independent, typically. Processes carry more state information than threads. Processes in shared memory model have separate address space, where threads share address space. Context switching between threads is typically faster than context switching between processes. Each process can be Thought of as being carried out by multiple lighter weight threads in this model.

In the shared memory model each program contains the same access to the shared memory, but in the threads model, each thread contains its own private memory, and also has access to a shared memory.

What is Parallel Programming? (in your own words)

Parallel Programming is the simultaneous execution of multiple processes on different processors that are either working on the same or separate tasks.

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

A system on chip is an electronic circuit board that integrates all of the necessary components in a computer and in other systems. It is comprised of a graphics processor, central processing unit, memory, power management circuits, a USB controller, wireless radios and other features. All of these components are permanently soldered onto the motherboard unlike standard computers where parts are interchangeable. The Raspberry PI system utilizes SoC.

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

A SoC is essentially a working computer, where a CPU requires other hardware to run, such as a GPU and RAM. SoC are typically smaller which makes them ideal for mobile computing for things like phones, tablets and smartwatches. They are more reliable and functional than microcontrollers in the embedded systems market. Mobile computing SoCs usually bundle processors, memories, on-chip cashes, wireless networking and other hardware and firmware, which makes them ideal for phones where space is a top priority, and people aren't going to be customizing things. Apple and Samsung both use versions of SoCs in their phones. The main advantage of using an SoC over a CPU is size. The only real disadvantage is flexibility. Users do not have the possibility of replacing the CPU, GPU or RAM, such as they do in mobile computers. But most people who use smart phones probably never considered doing that to it in the first place.

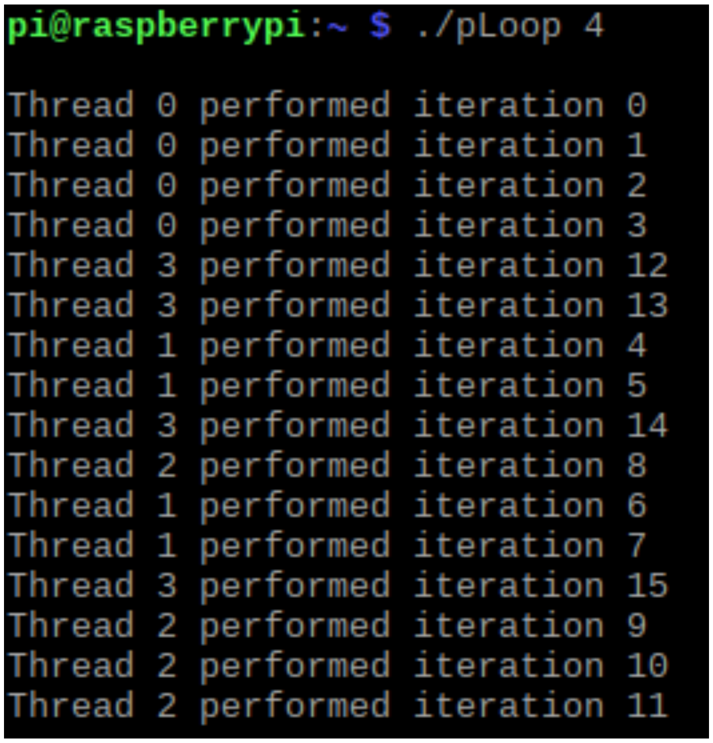
Parallel Programming Basics:

1. Open the terminal and type *nano parallelLoopEqualChunks.c* to open an editor called nano to create a file named *parallelLoopEqualChunks.c.* Copy the following code into the executable program. (Use ctrl+w to save the file and ctrl+x to exit the file)

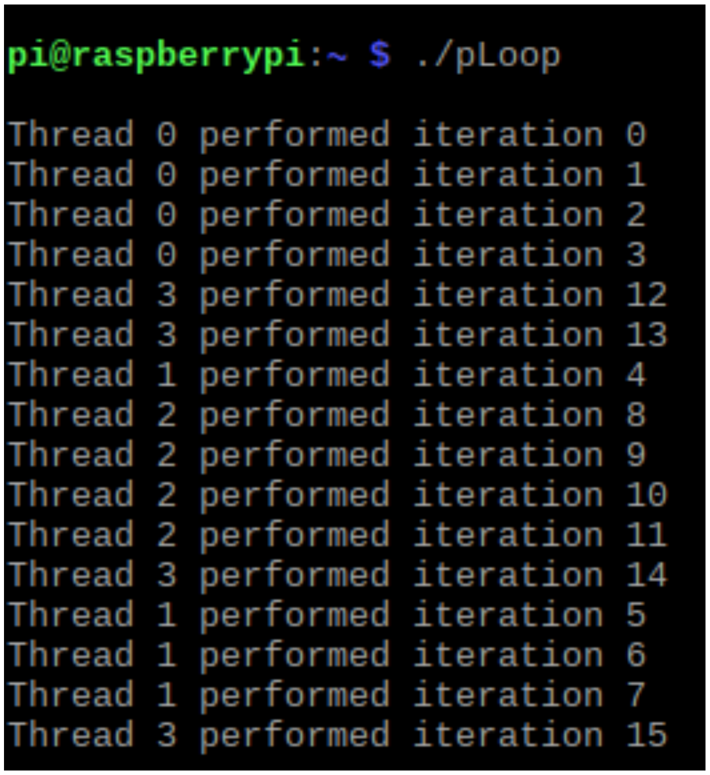
```
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP

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;
}
```

2. Type *gcc parallelLoopEqualChunks.c -o pLoop -fopenmp* to create an executable file named pLoop. Run the program by typing ./pLoop **4**(The 4 is a command-line argument. You can replace 4 with another value) Test with other values. (I used 4, blank, 5, 16, 7, 0, &12).

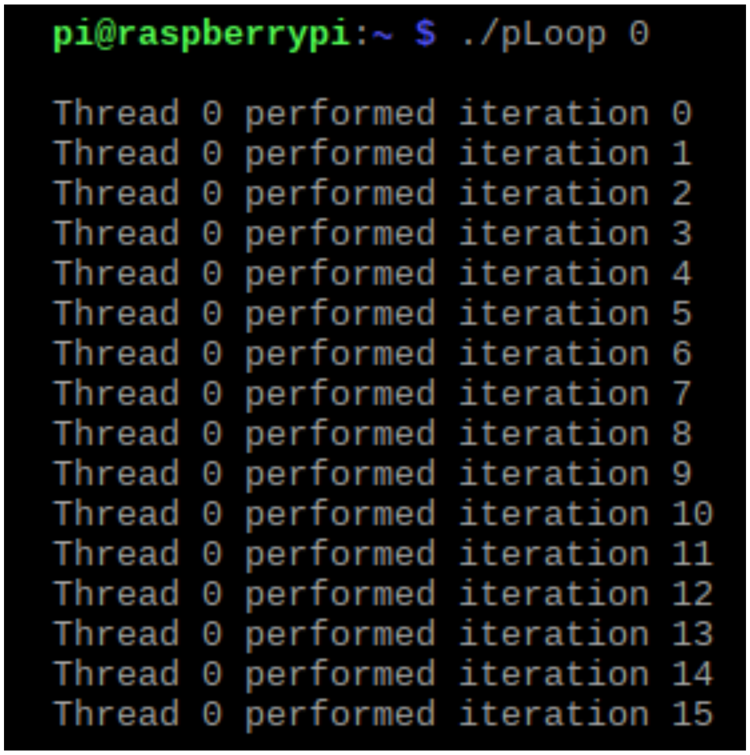
Results for pLoop:



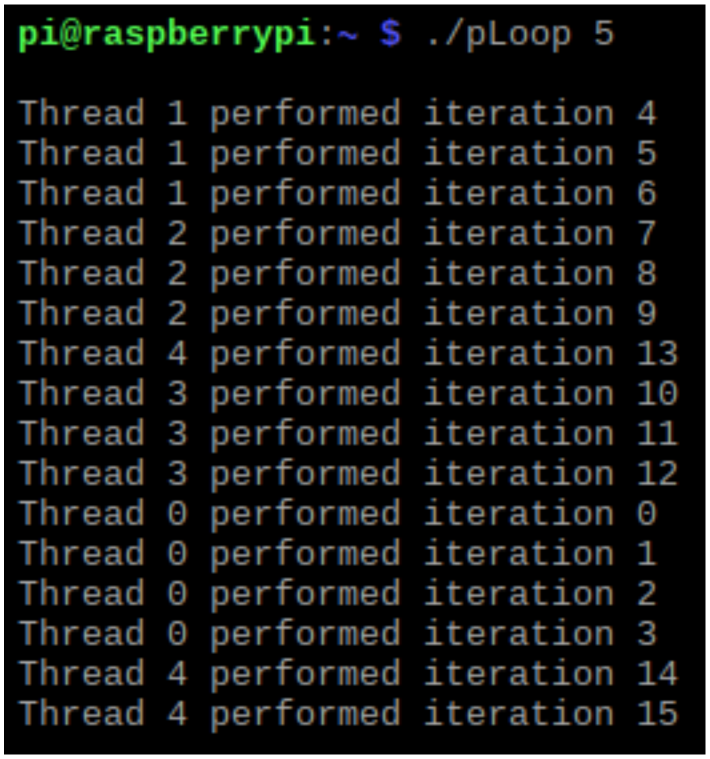
```
pi@raspberrypi:~ $ ./pLoop 4

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



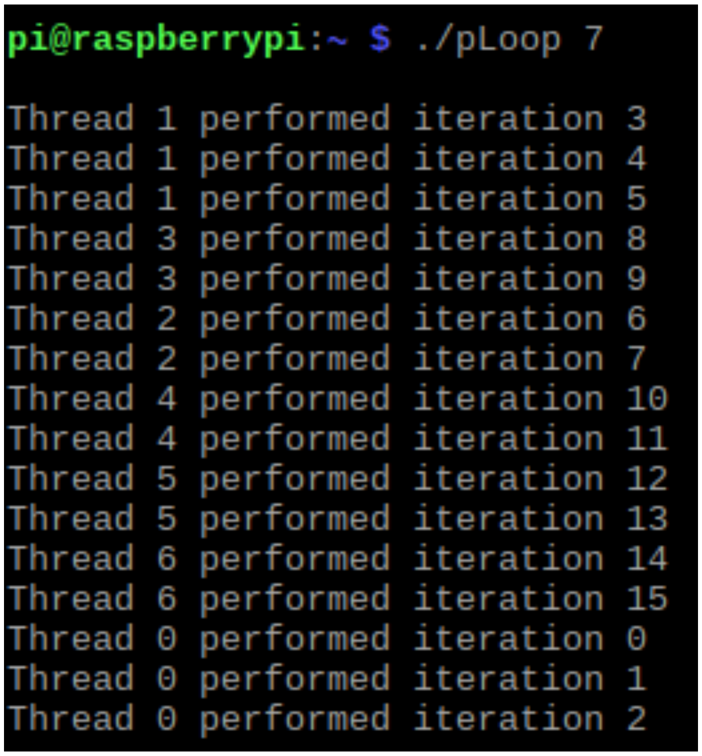
```
pi@raspberrypi:~ $ ./pLoop

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 1 performed iteration 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 14
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 3 performed iteration 15
```



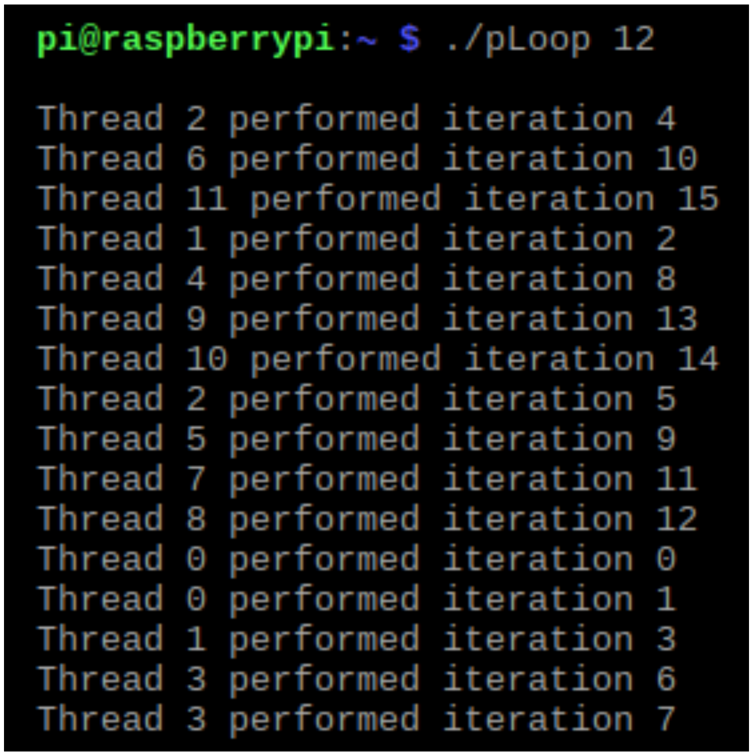
```
pi@raspberrypi:~ $ ./pLoop 0

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



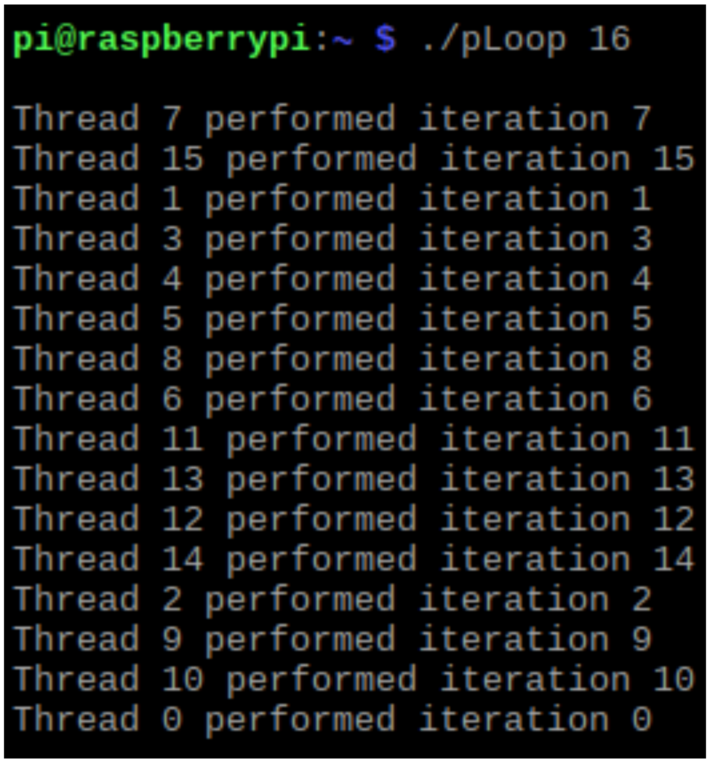
```
pi@raspberrypi:~ $ ./pLoop 5

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



```
pi@raspberrypi:~ $ ./pLoop 7

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



```
pi@raspberrypi:~ $ ./pLoop 12

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



```
pi@raspberrypi:~ $ ./pLoop 16

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

Observation: When you don't have a command line argument (leaving it blank) it will default to how many cores the CPU have. In this case it will be 4 because the Raspberry Pi have 4-cores.) Also, when the number of iterations can not be evenly divisible, some threads will have more iterations than others.

This type of code is usually used when the memory is stored consecutively (such as an array).

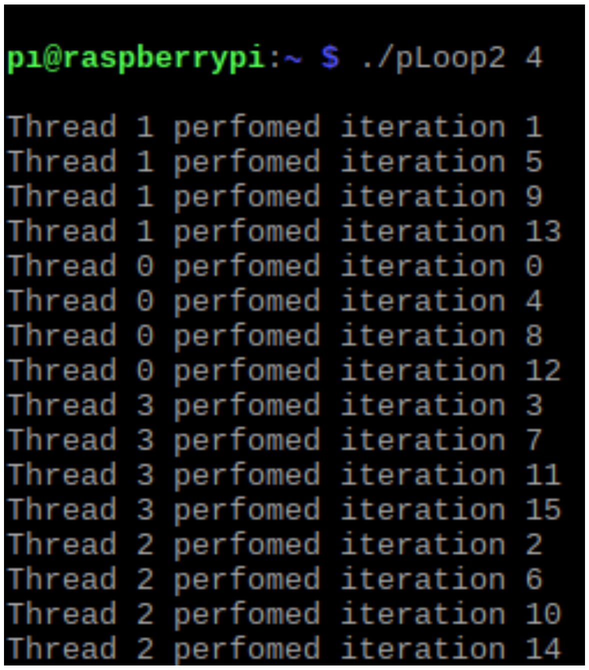
3. Type *nano parallelLoopEqualChunksOf1.c* to open up an editor called to create a file named *parallelLoopEqualChunksOf1.c.* Copy the following code into the executable program.

```
#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---\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;
 }
```

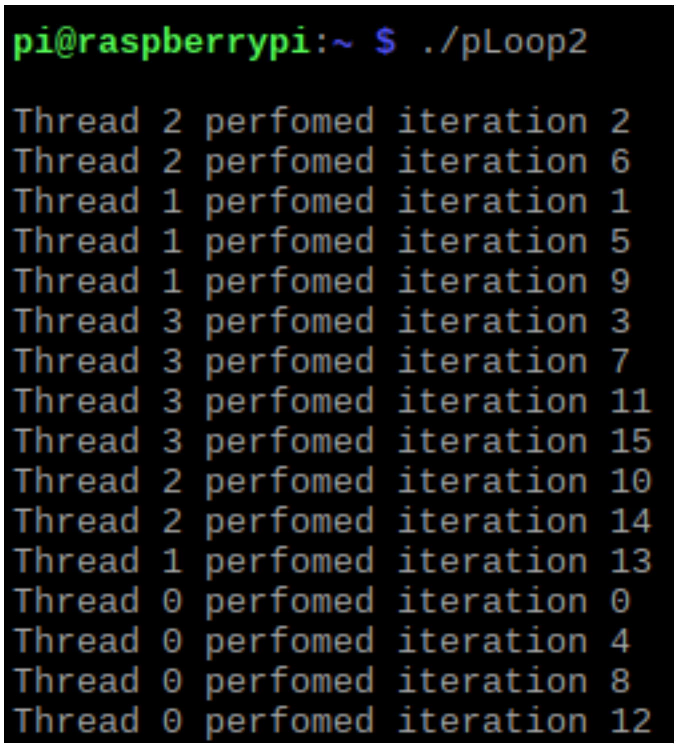
For this method, the threads get the next available iteration instead of completing all the iterations and moving to the next thread. There is a static variable so the work can still be divided evenly. The iterations will not be consecutive though. The schedule clause is there so each thread does one iteration of the loop in a regular pattern.

4.Type *gcc parallelLoopEqualChunksOf1.c -o pLoop2 -fopenmp* to create a executable file named pLoop. Run the program by typing ./pLoop2 **4** Test with other values. (I used 4, blank, 5, 16, 7, 0, &12).

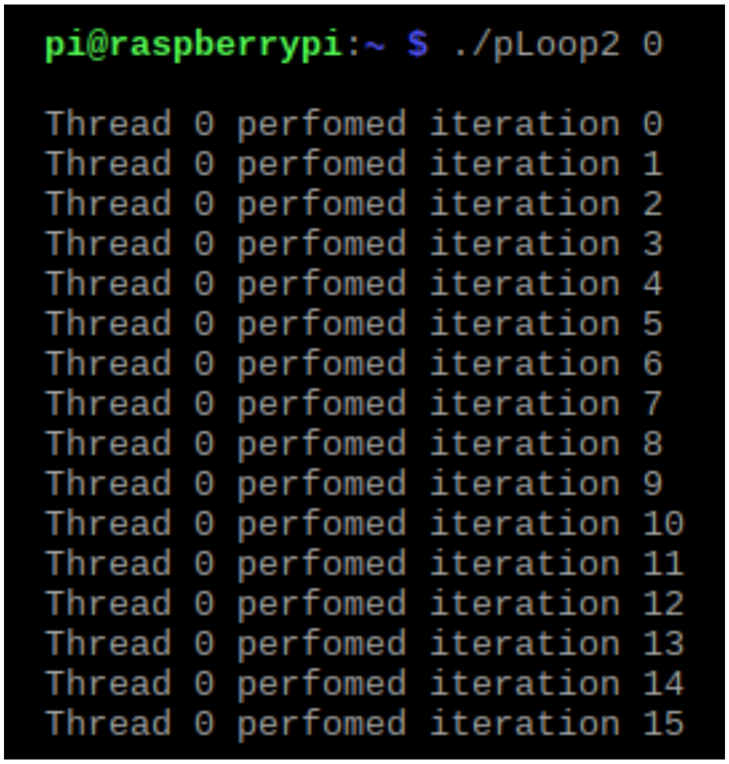
Results for pLoop2:



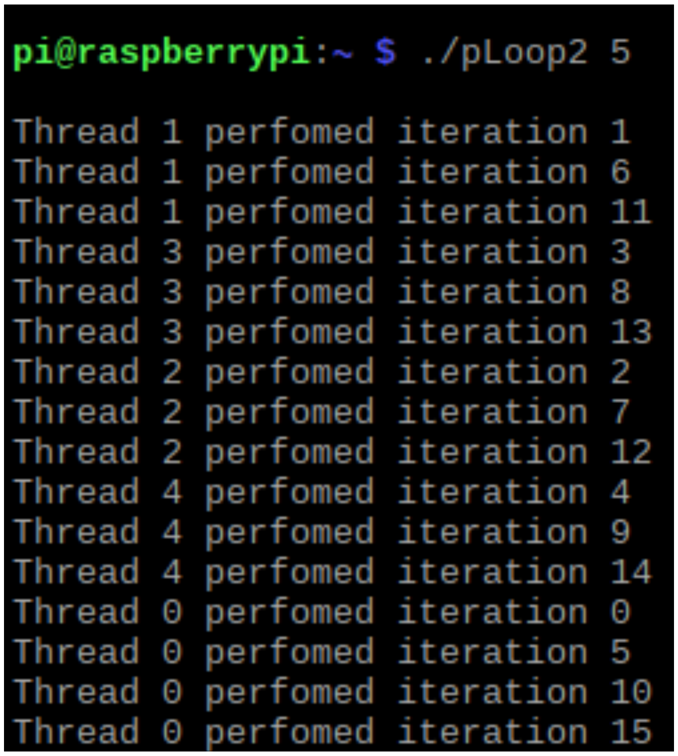
```
pi@raspberrypi:~ $ ./pLoop2 4

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



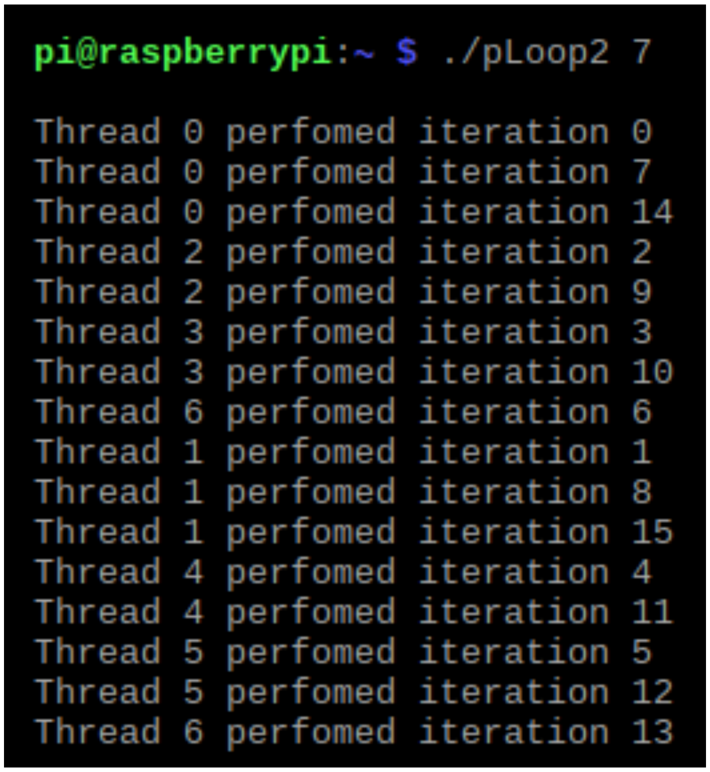
```
pi@raspberrypi:~ $ ./pLoop2

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



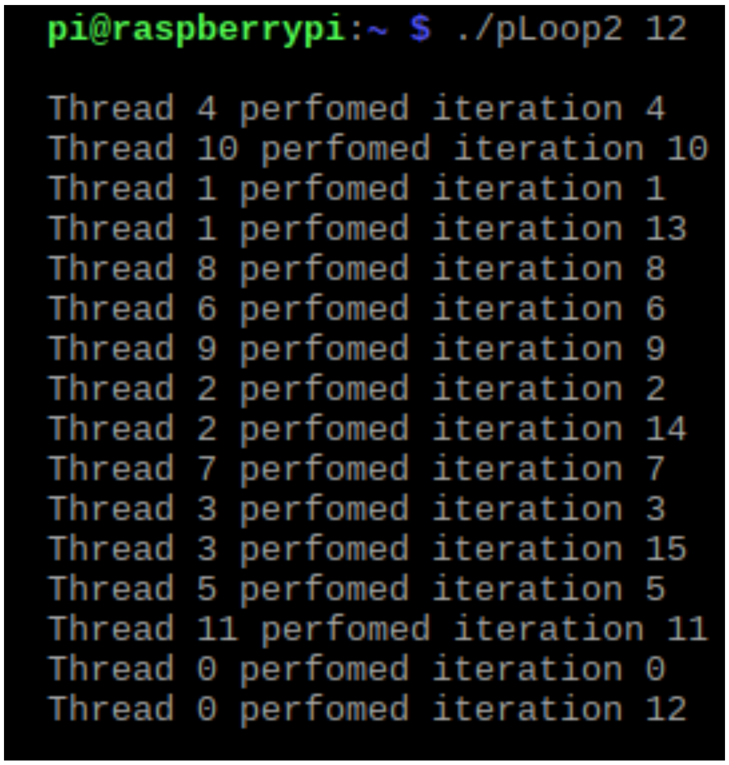
```
pi@raspberrypi:~ $ ./pLoop2 0

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



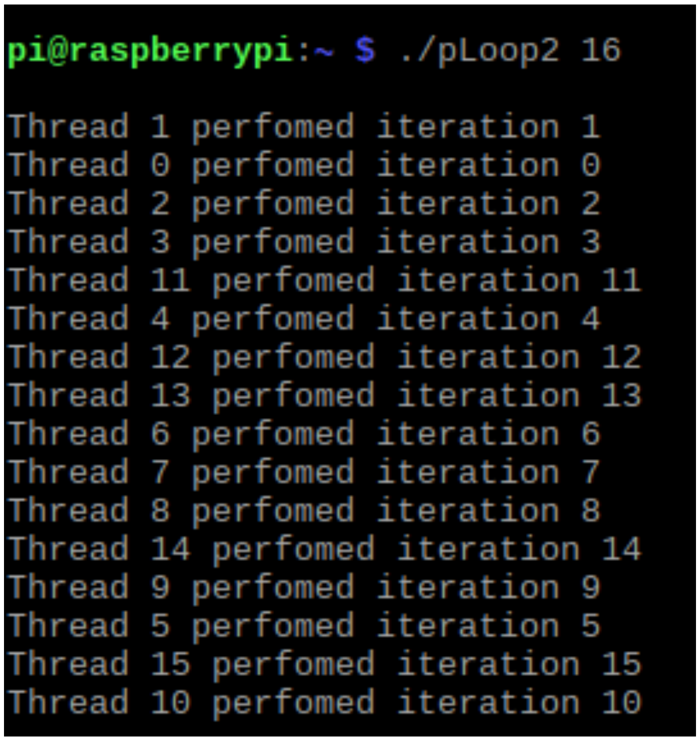
```
pi@raspberrypi:~ $ ./pLoop2 5

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



```
pi@raspberrypi:~ $ ./pLoop2 7

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



```
pi@raspberrypi:~ $ ./pLoop2 12

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



```
pi@raspberrypi:~ $ ./pLoop2 16

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

Observation: Thanks to the static, the work is divided evenly as the first snippet of code.

5.Type *nano reduction.c* to open up an editor to create file named *reduction.c.* Copy the following code into the executable program.

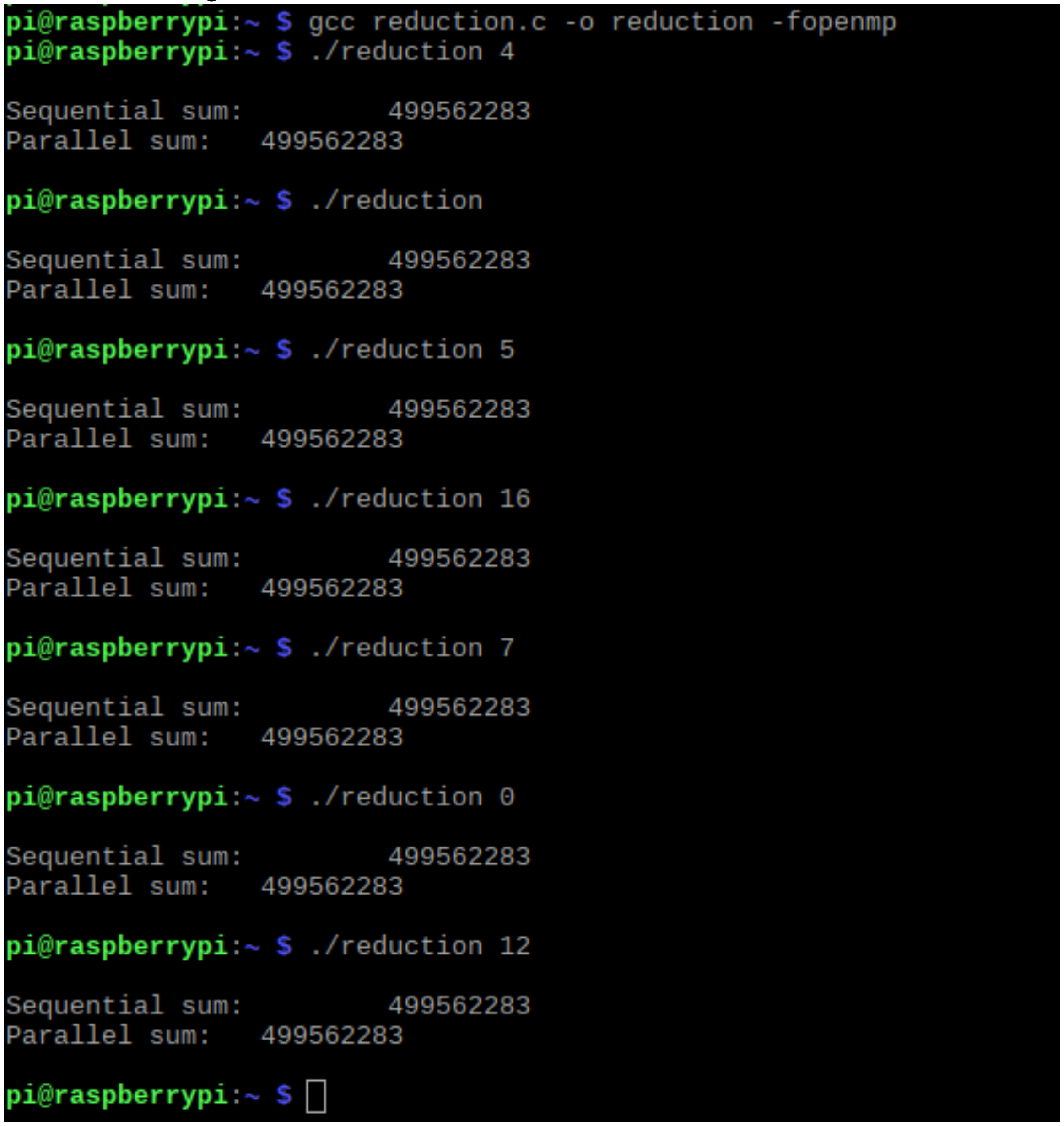
```
#include <stdio.h> // printf()
#include <omp.h> // OpenMP
#include <stdlib.h> // rand()

 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;
 }

 /* 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;
 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 for // reduction(+:sum)
for (i = 0; i < n; i++) {
sum += a[i];
}
return sum;
}
```

The parallels must communicate to keep the sum updates with each other (stay the same). They use a special cause called reduction (+:sum in the code snippet above). The addition sign points out that the sum is computed by adding all of the values together in the loop.

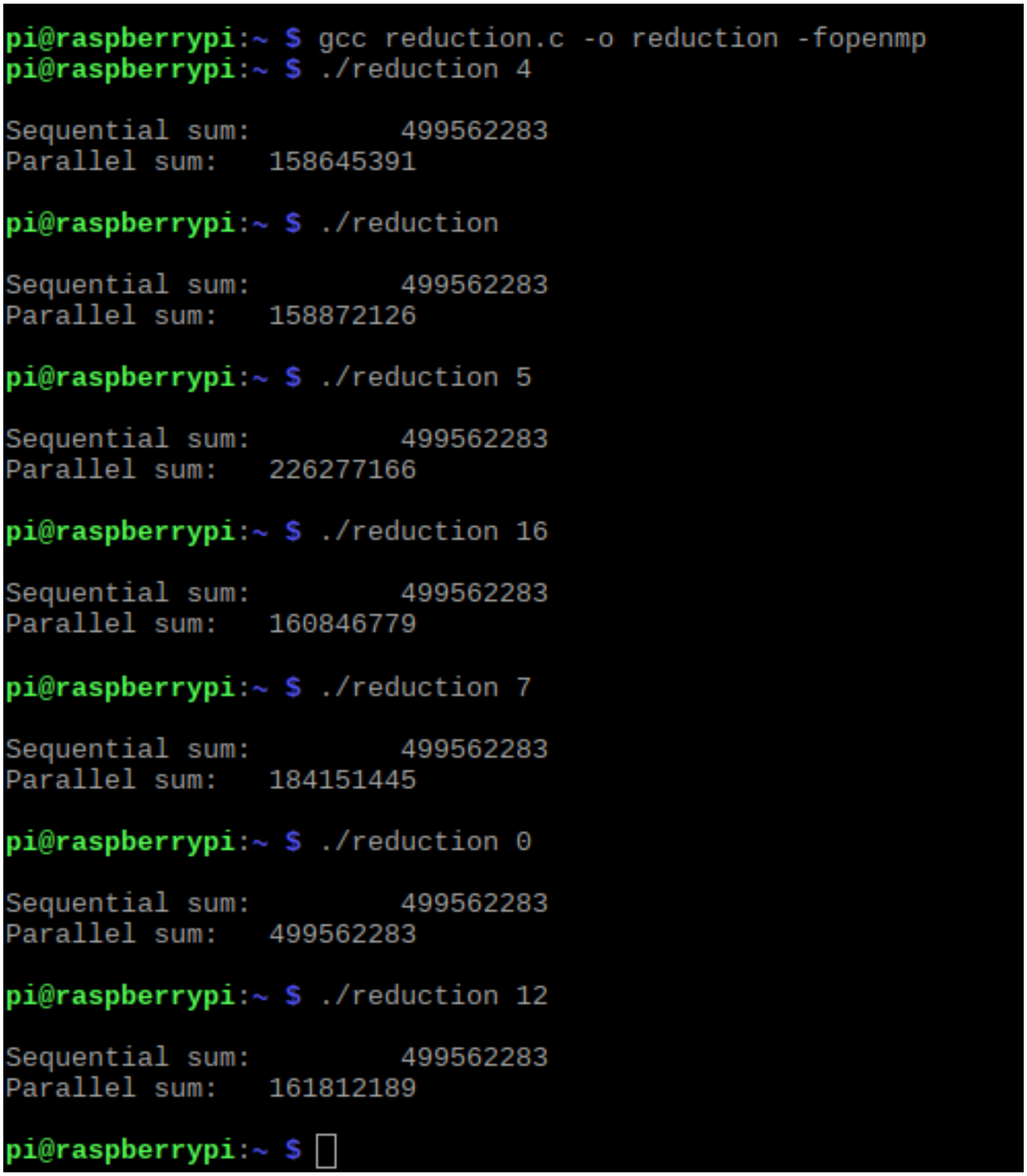
6. Type *gcc reduction.c -o reduction -fopenmp* to create a executable file named reduction. Run the program by typing ./reduction **4** Test with other values. (I used 4, blank, 5, 16, 7, 0, &12).

Results of the original code of reduction:



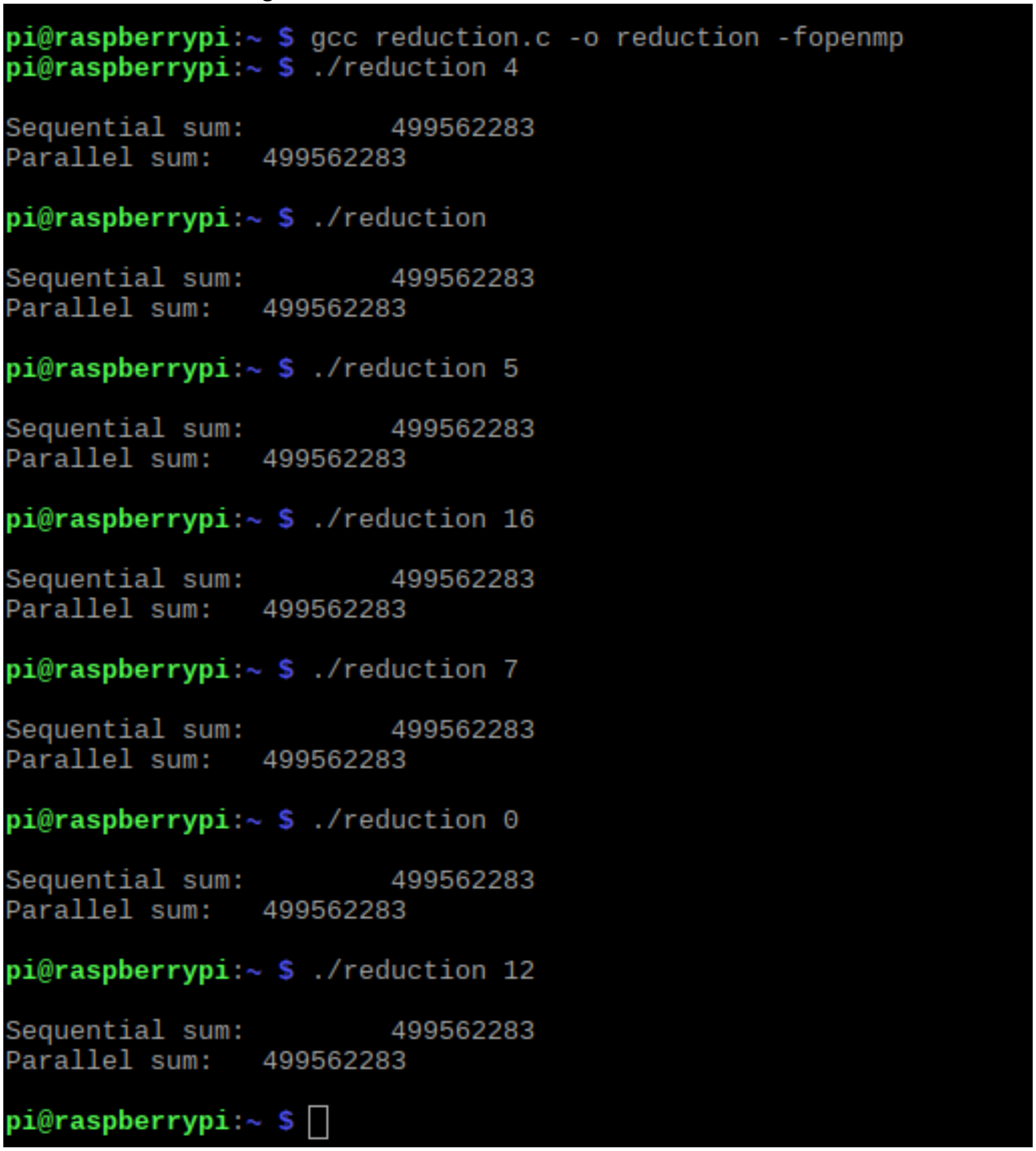
```
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 16

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 7

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 0

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 12

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ 
```

Results of uncommented the first comment code for reduction:



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

Sequential sum:         499562283
Parallel sum:   158645391

pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:   158872126

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parallel sum:   226277166

pi@raspberrypi:~ $ ./reduction 16

Sequential sum:         499562283
Parallel sum:   160846779

pi@raspberrypi:~ $ ./reduction 7

Sequential sum:         499562283
Parallel sum:   184151445

pi@raspberrypi:~ $ ./reduction 0

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 12

Sequential sum:         499562283
Parallel sum:   161812189

pi@raspberrypi:~ $ 
```

Results of uncommenting both comments for reduction:



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

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 5

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 16

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 7

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 0

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 12

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ 
```

Observation 1: The parallel sum & the sequential sum are the same.

Observation 2: When you get rid of the first comment, the parallel sum is less than the sequential sum.

Observation 3: When you get rid both comments, the sums become the same again.

Do you have any ideas why the parallel for pragma without the reduction clause did not produce the correct answer?

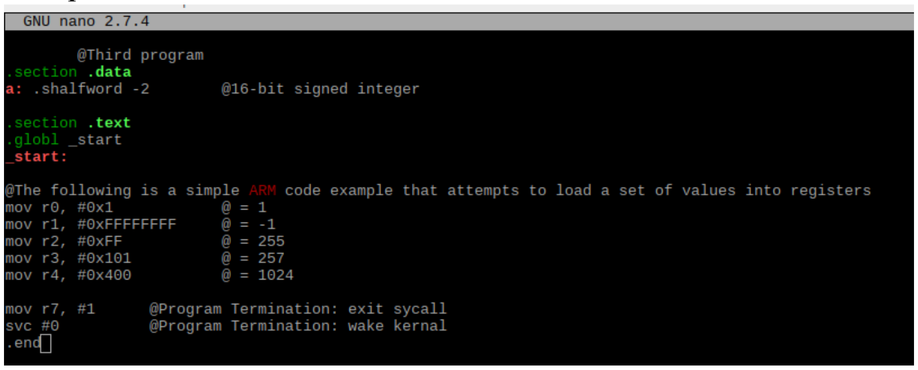
Since the reduction clause was commented out, the parallels had no way to communicate with each other. The sum is dependent on what the threads are doing.

ARM Assembly Programming:
By: Donald Weaver

The goal of this assignment was to practice data movement in ARM, learn how data is stored in memory and registers, as well as to practice and get comfortable using the GDB.

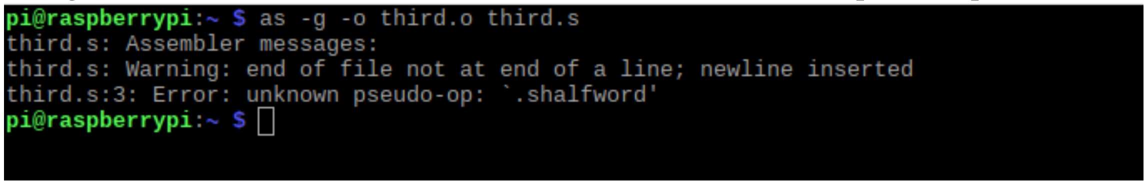
Part 1:

The first command is nano. By executing the command "nano third.s" we open a text editor and create a file named "third.s." Then type a simple program to demonstrate data manipulation and data transfer:



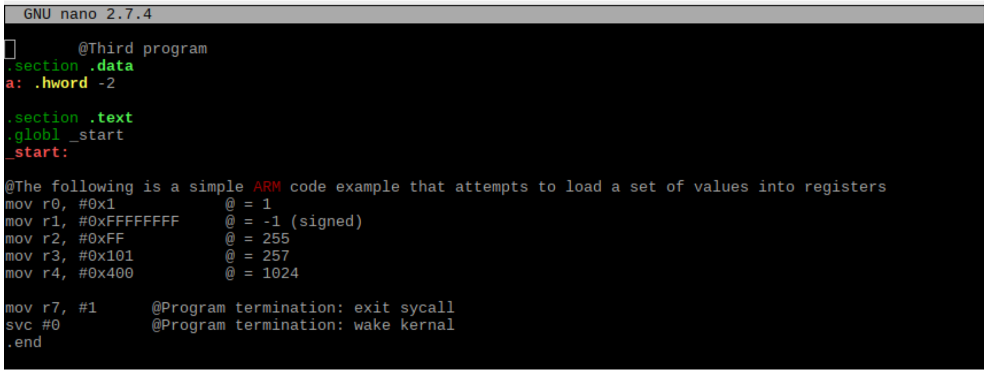
```
 GNU nano 2.7.4

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

.section .text
.globl _start
_start:

@The following is a simple ARM code example that attempts to load a set of values into registers
mov r0, #0x1            @ = 1
mov r1, #0xFFFFFFFF     @ = -1
mov r2, #0xFF           @ = 255
mov r3, #0x101          @ = 257
mov r4, #0x400          @ = 1024

mov r7, #1      @Program Termination: exit sycall
svc #0          @Program Termination: wake kernal
.end
```

Using the "as" command to assemble reveals there is an unknown pseudo op error



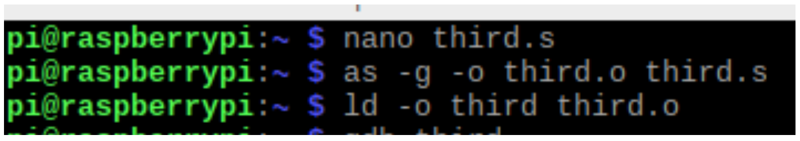
```
pi@raspberrypi:~ $ as -g -o third.o third.s
third.s: Assembler messages:
third.s: Warning: end of file not at end of a line; newline inserted
third.s:3: Error: unknown pseudo-op: `.shalfword'
pi@raspberrypi:~ $
```

ARM does not recognize "shalfword" sp to remedy this we need to use "hword" and by assigning a signed value to it the program will recognize it as a signed variable.



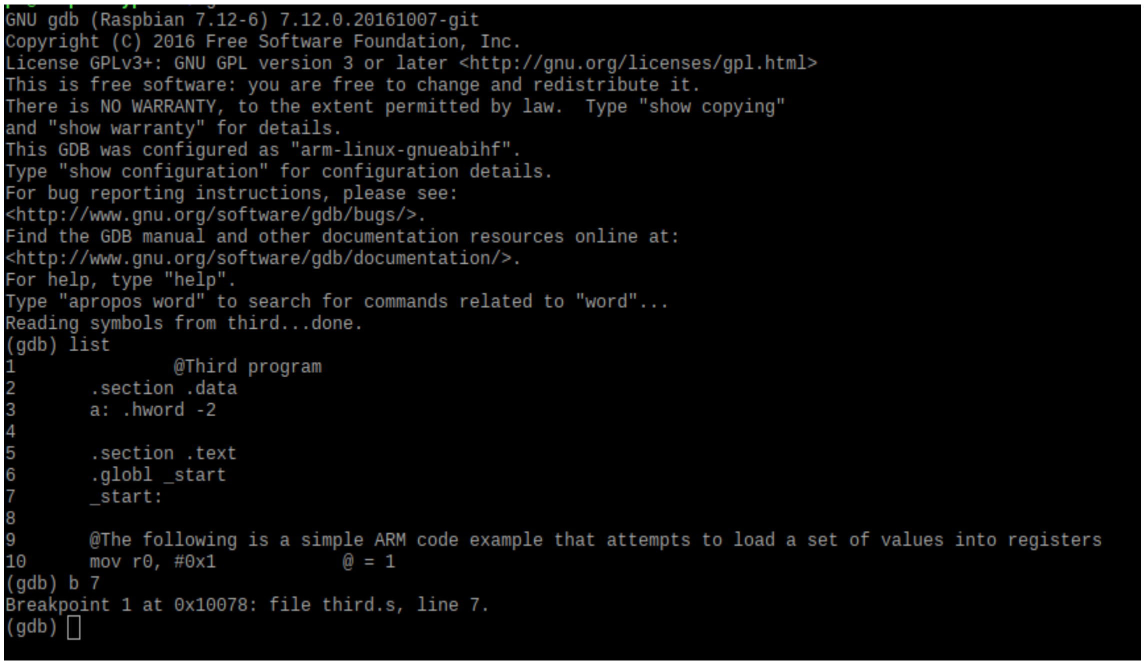
```
 GNU nano 2.7.4

        @Third program
.section .data
a: .hword -2

.section .text
.globl _start
_start:

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

mov r7, #1      @Program termination: exit sycall
svc #0          @Program termination: wake kernal
.end
```

Now "as -g -o third.o third.s" will assemble our program under the name third.o. And the command "ld -o third third.o" will create an executable named third.



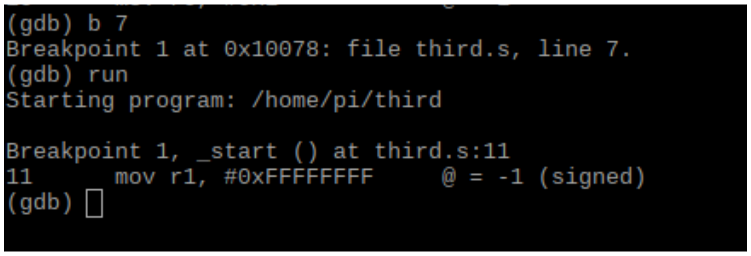
```
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
pi@raspberrypi:~ $ ld -o third third.o
```

To open the GDB simple execute the command “gdb third” and to display the lines type “list” to display the first 10 lines.



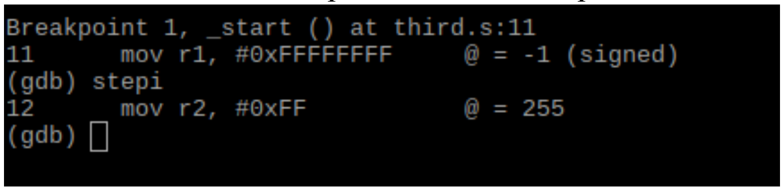
```
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from third...done.
(gdb) list
1               @Third program
2       .section .data
3       a: .hword -2
4
5       .section .text
6       .globl _start
7       _start:
8
9       @The following is a simple ARM code example that attempts to load a set of values into registers
10      mov r0, #0x1            @ = 1
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 7.
(gdb)
```

We can set a breakpoint by using the command “b” followed by the line we want to put the break point, in this case “b 7” Then we can type run to start the debugging process.



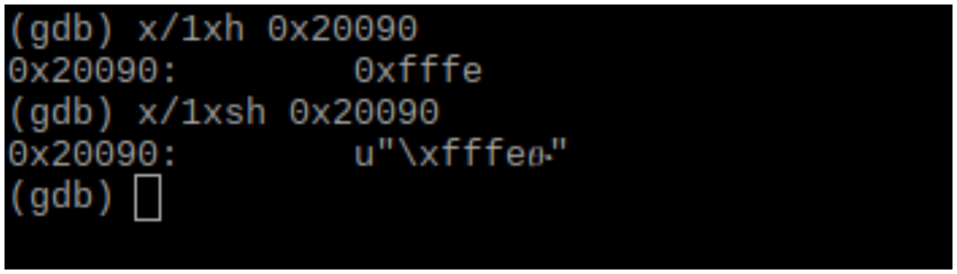
```
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 7.
(gdb) run
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:11
11      mov r1, #0xFFFFFFFF     @ = -1 (signed)
(gdb)
```

Next, we can use the “stepi” command to step over one command.



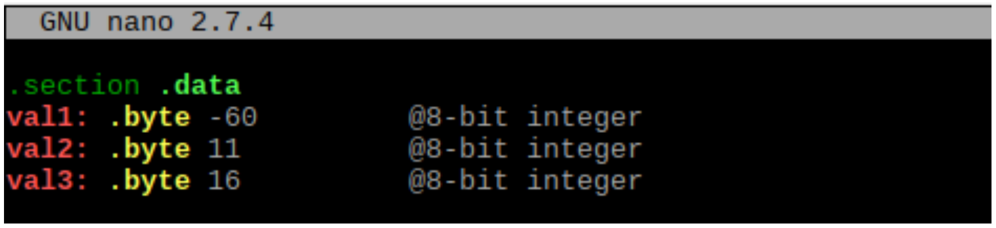
```
Breakpoint 1, _start () at third.s:11
11      mov r1, #0xFFFFFFFF     @ = -1 (signed)
(gdb) stepi
12      mov r2, #0xFF           @ = 255
(gdb)
```

Finally we can use the “x” command to display data values stored in certain addresses by typing “x/ FMT Address” we can display the value contained in any address. In this case we want to know what our “hword” variable contains. So we type “x/1xh 0x20090” 20090 being the the address of our variable ‘a’. With this we get the desired output FFFEh, or -2d The ‘h’ in 1xh denotes the data type, halfword. But if we use sh to denote signed halfword we get a jumbled mess.



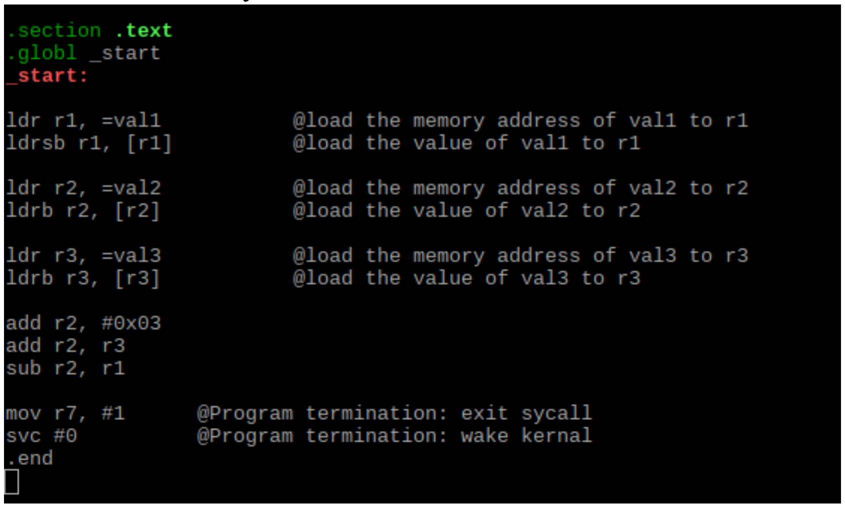
```
(gdb) x/1xh 0x20090
0x20090:        0xfffe
(gdb) x/1xsh 0x20090
0x20090:        u"\xfffe"
(gdb) 
```

Part 2:
Now we are tasked to write a code do calculate the following expression
Register = val2 + 3 + val3 - val1
To begin we need to create memory variable for each of the three val's



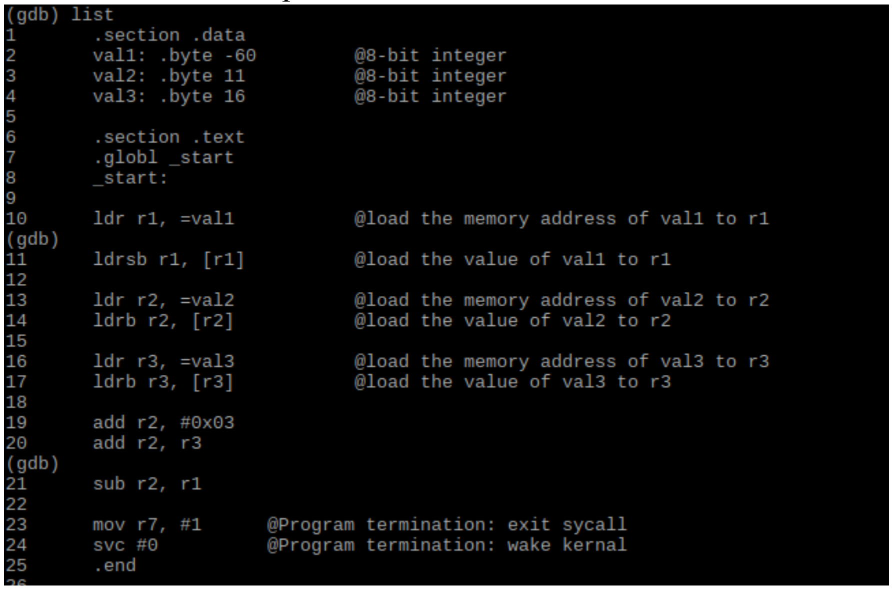
```
  GNU nano 2.7.4

.section .data
val1: .byte -60         @8-bit integer
val2: .byte 11          @8-bit integer
val3: .byte 16          @8-bit integer
```

Next we can begin by writing our code. We need to remember that in ARM we can't transfer data from memory to memory or memory to register, only register to register. So we must use "ldr" to load values into register before adding and subtracting them from each other. The second thing to remember is the data types. For example, when we load val1 which is a signed byte we can't use "ldr", because "ldr" assumes a word data type. So we must use "ldrsb". And for the other two bytes we must use "ldrb"



```
.section .text
.globl _start
_start:

ldr r1, =val1           @load the memory address of val1 to r1
ldrsb r1, [r1]          @load the value of val1 to r1

ldr r2, =val2           @load the memory address of val2 to r2
ldrb r2, [r2]           @load the value of val2 to r2

ldr r3, =val3           @load the memory address of val3 to r3
ldrb r3, [r3]           @load the value of val3 to r3

add r2, #0x03
add r2, r3
sub r2, r1

mov r7, #1      @Program termination: exit sycall
svc #0          @Program termination: wake kernal
.end
```

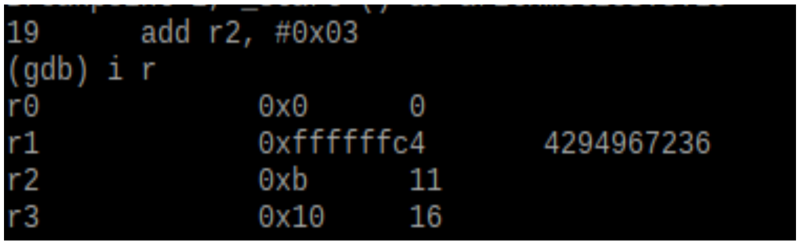
Now that the code is written we can assemble it and use the linked to create an executable so we can go right into debugging. This time once we are in the GDB we can use list again to display the first 10 lines then press enter until the entire code is visible.



```
(gdb) list
1       .section .data
2       val1: .byte -60         @8-bit integer
3       val2: .byte 11          @8-bit integer
4       val3: .byte 16          @8-bit integer
5
6       .section .text
7       .globl _start
8       _start:
9
10      ldr r1, =val1           @load the memory address of val1 to r1
(gdb)
11      ldrsb r1, [r1]          @load the value of val1 to r1
12
13      ldr r2, =val2           @load the memory address of val2 to r2
14      ldrb r2, [r2]           @load the value of val2 to r2
15
16      ldr r3, =val3           @load the memory address of val3 to r3
17      ldrb r3, [r3]           @load the value of val3 to r3
18
19      add r2, #0x03
20      add r2, r3
(gdb)
21      sub r2, r1
22
23      mov r7, #1      @Program termination: exit sycall
24      svc #0          @Program termination: wake kernal
25      .end
```

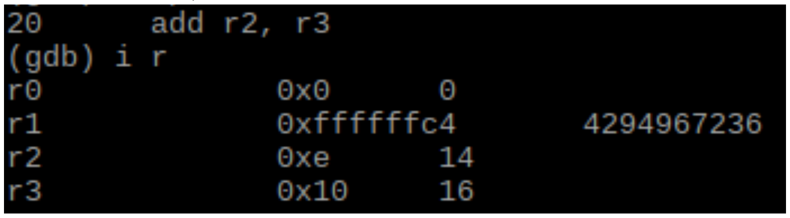
Now we set a breakpoint at line 19 and then we can run the program. To display the value of the registers we can use the command “i r”

Initial values



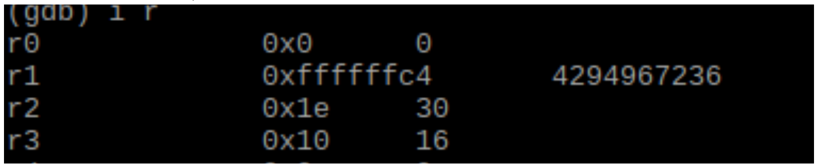
```
19      add r2, #0x03
(gdb) i r
r0             0x0      0
r1             0xffffffc4       4294967236
r2             0xb      11
r3             0x10     16
```

After “add r2, #0x03”



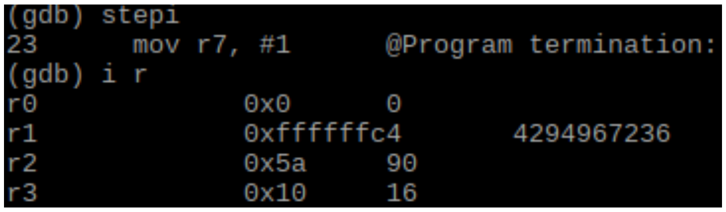
```
20      add r2, r3
(gdb) i r
r0             0x0      0
r1             0xffffffc4       4294967236
r2             0xe      14
r3             0x10     16
```

After “add r2, r3”



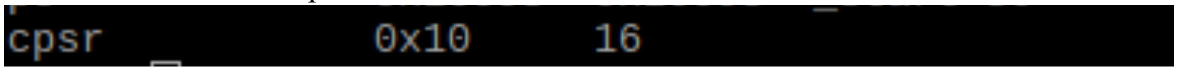
```
(gdb) i r
r0             0x0      0
r1             0xffffffc4       4294967236
r2             0x1e     30
r3             0x10     16
```

After “sub r2, r1”



```
(gdb) stepi
23      mov r7, #1      @Program termination:
(gdb) i r
r0             0x0      0
r1             0xffffffc4       4294967236
r2             0x5a     90
r3             0x10     16
```

We stored the final value in r2 which you can see comes out to be 90d. To show the signed flag we need to look at the cpsr value which remained at a constant 10h.



```
cpsr           0x10     16
```

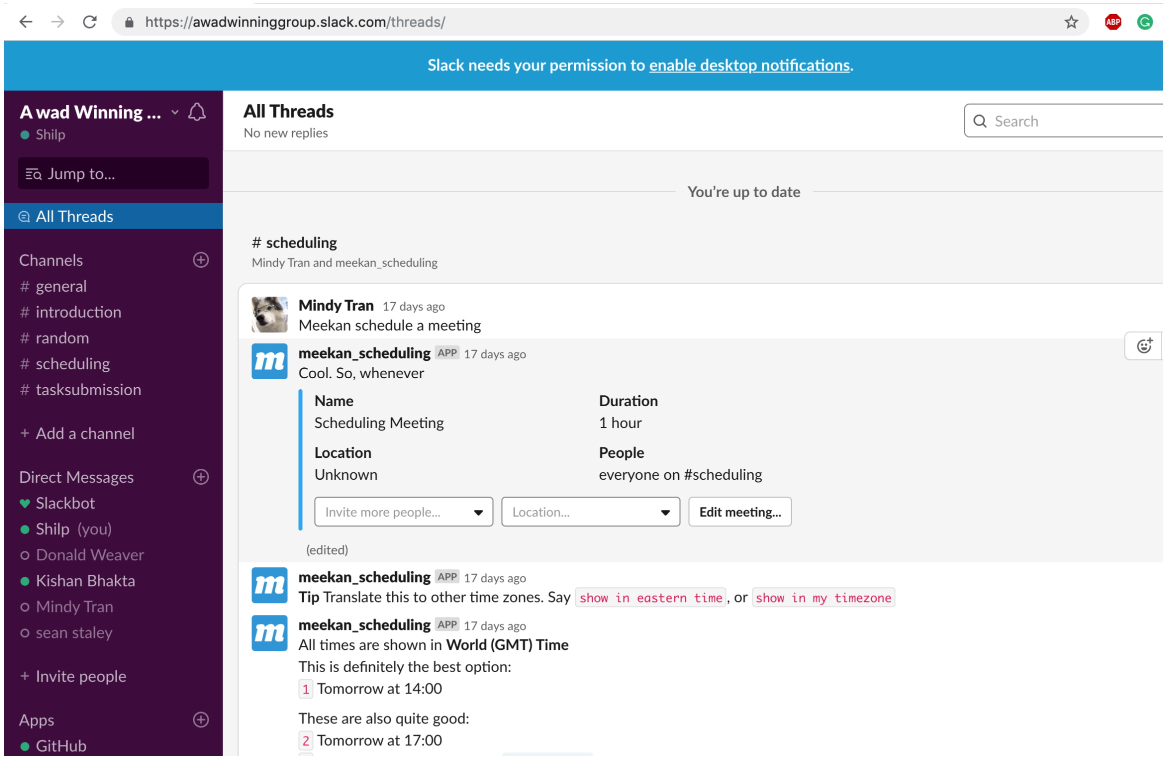
To make sense of this we need to convert it to binary. And we end up with

00010000

The signed flag is located on bit 7 which in this case is 0.

Appendix

Slack: https://awadwinninggroup.slack.com/



scheduling
Mindy Tran and meekan_scheduling

Mindy Tran 17 days ago
Meekan schedule a meeting

meekan_scheduling APP 17 days ago
Cool. So, whenever

Name	Duration
Scheduling Meeting	1 hour
Location	**People**
Unknown	everyone on #scheduling

Invite more people... | Location... | Edit meeting...

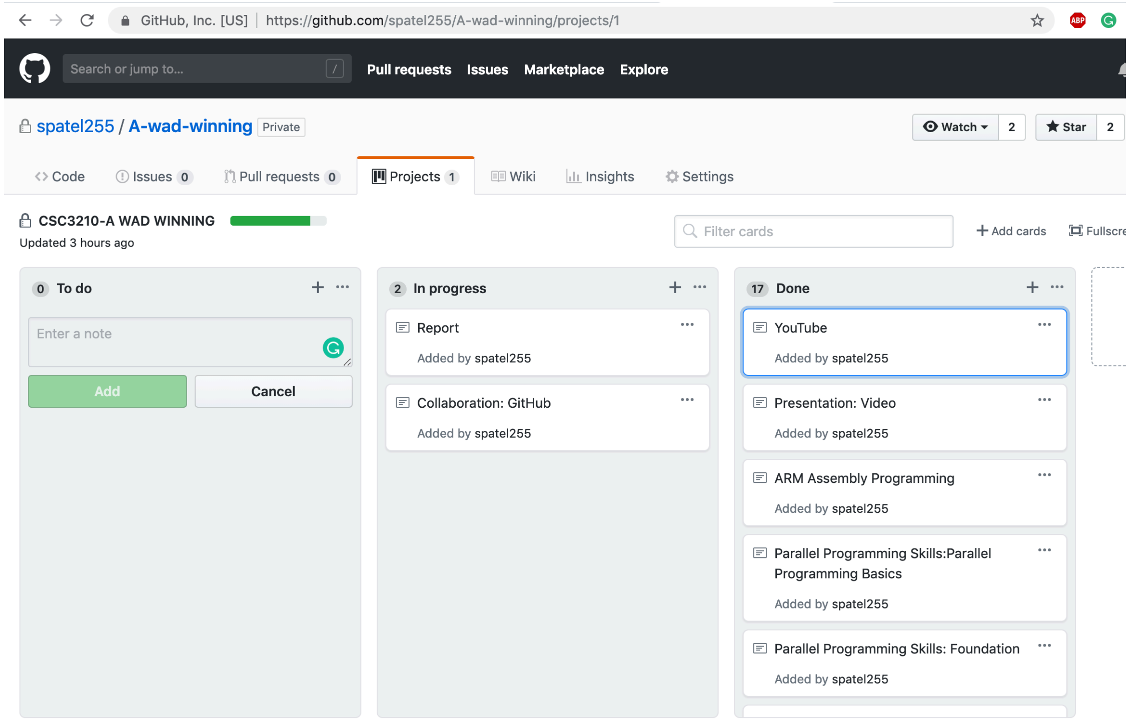
(edited)

meekan_scheduling APP 17 days ago
Tip Translate this to other time zones. Say `show in eastern time`, or `show in my timezone`

meekan_scheduling APP 17 days ago
All times are shown in **World (GMT) Time**
This is definitely the best option:
1 Tomorrow at 14:00

These are also quite good:
2 Tomorrow at 17:00

GitHub: https://github.com/spatel255/A-wad-winning



CSC3210-A WAD WINNING
Updated 3 hours ago

0 To do	2 In progress	17 Done
Enter a note	Report — Added by spatel255	YouTube — Added by spatel255
Add / Cancel	Collaboration: GitHub — Added by spatel255	Presentation: Video — Added by spatel255
		ARM Assembly Programming — Added by spatel255
		Parallel Programming Skills:Parallel Programming Basics — Added by spatel255
		Parallel Programming Skills: Foundation — Added by spatel255

YouTube Channel:
https://www.youtube.com/channel/UCVEvyRC9_1g_8306r68504g