**PARALLEL PROGRAMMING QUESTIONS**

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

Task: basic unit of programming that an operating system controls.

Pipelining: process of gathering and executing computer instructions and tasks from the processor through a logical pipeline.

Shared memory: it is part of random access memory that can be accessed by all the processors in a multiprocessor system.

Synchronization: process of accurately coordinating or matching multiple activities, devices, or processes in time.

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

SISD(Single Instruction, Single Data stream): a CPU runs only on one instruction at a time and also stores only one data at a time to manage on data stored in the single memory unit.

MISD(Multiple Instructions, Single Data stream): in which each functional unit does different operations by processing different instructions on the same data.

SIMD(Single Instruction, Multiple Data streams): an ISA that only has one control unit and many different processing units

MIMD(Multiple Instruction, Multiple Data streams): is an ISA for parallel computing which consists of multiprocessing computers.

**(7p) What are the Parallel Programming Models?**

Parallel Programming Models include Shared memory, Thread memory, Distributed memory, Data Parallel, Hybrid, Single Program Multiple Data, Multiple Program Multiple Data. Parallel programming serve as an abstraction above hardware and memory architectures.

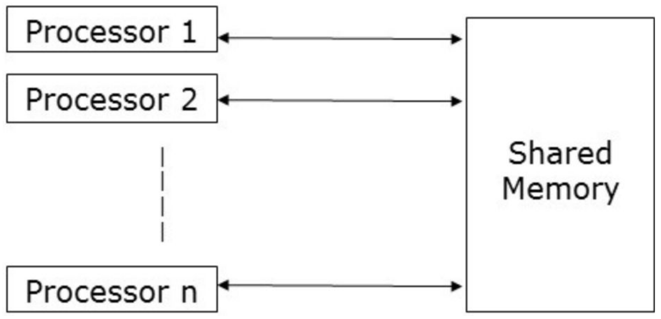
**(12p) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**

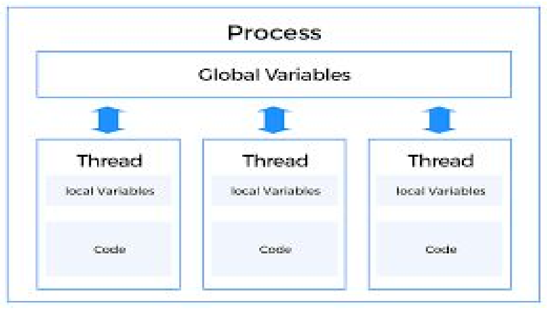
Uniform Memory Access(UMA): all the processors have equal access time to all the memory.

Non-uniform memory access(NUMA): the access time depends on the location of the memory word. OpenMP uses uniform memory access(UMA) because it uses single memory controller.

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

Share Memory Model is when tasks and processes are able to write and read to a shared space at different timings in a serial way. Thread Model is when a big process is broken down to smaller processes in a simultaneous way.





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

Parallel Programming is where a problem is broken down into smaller steps, gives instructions and through the processors gives the output at once.

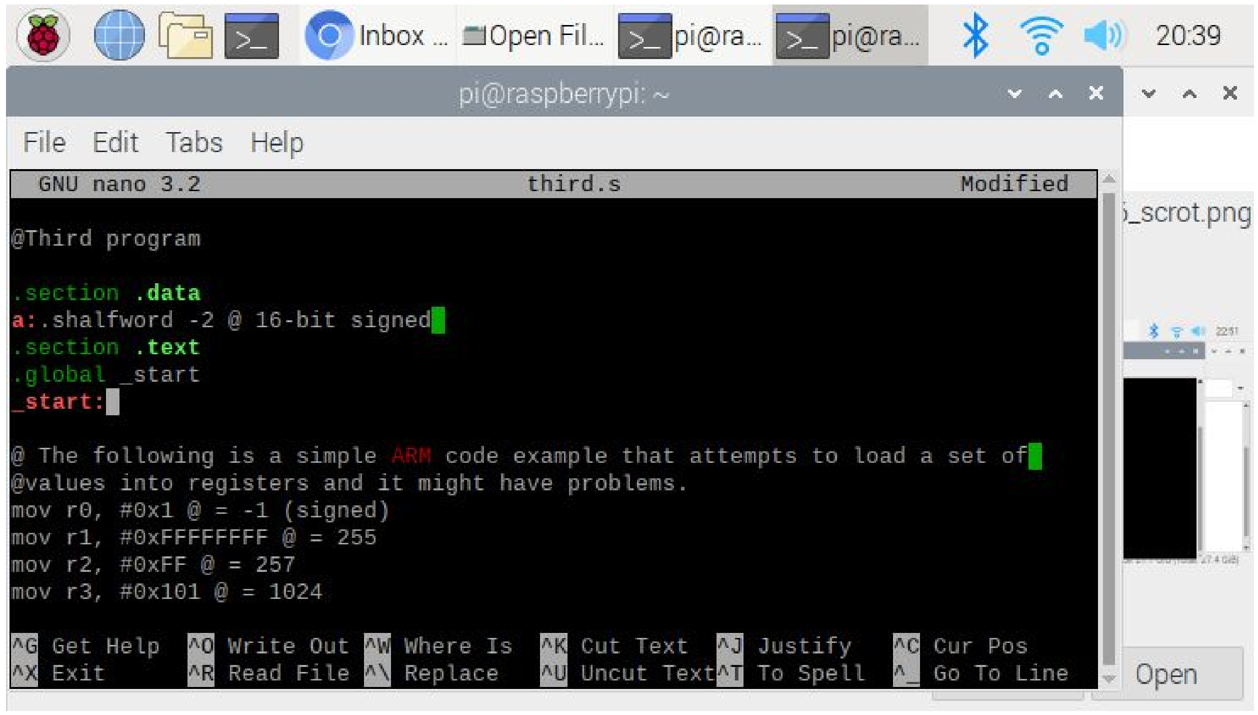
**(5p) What is system on chip (SoC)? Does Raspberry PI use system on SoC? -**

System on chip (SoC) combines the CPU, GPU, USB controller, wireless radios and power management circuits in a single chip. Raspberry PI uses a system of SoC.

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

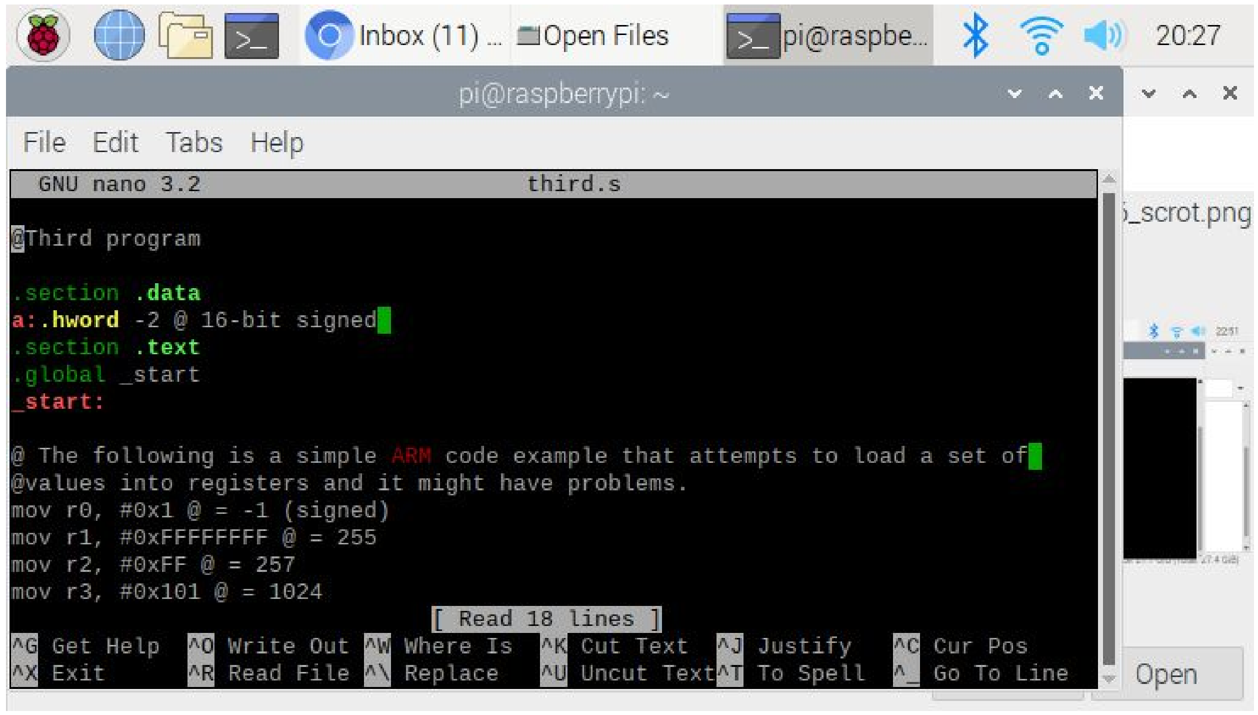
The system on a chip is more convenient because of the greater system reliability. It is smaller in size so it is more portable. System on a Chip has a faster execution because of high speed processors and memory.

**ARM PROGRAMMING**



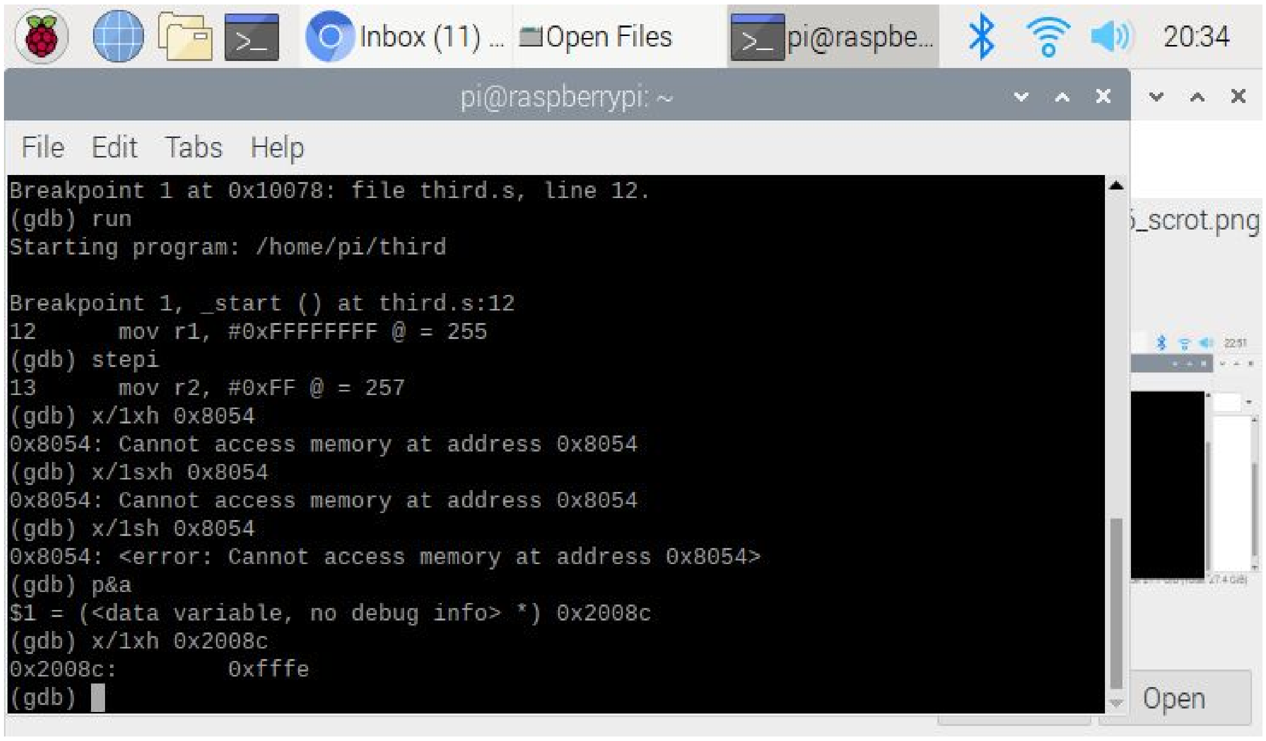
```
  GNU nano 3.2                    third.s                     Modified

@Third program

.section .data
a:.shalfword -2 @ 16-bit signed
.section .text
.global _start
_start:

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

^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
```

This is the source code for third.s. Under the .data section, .shalfword caused a compilation error and would give an error after entering the next command line.



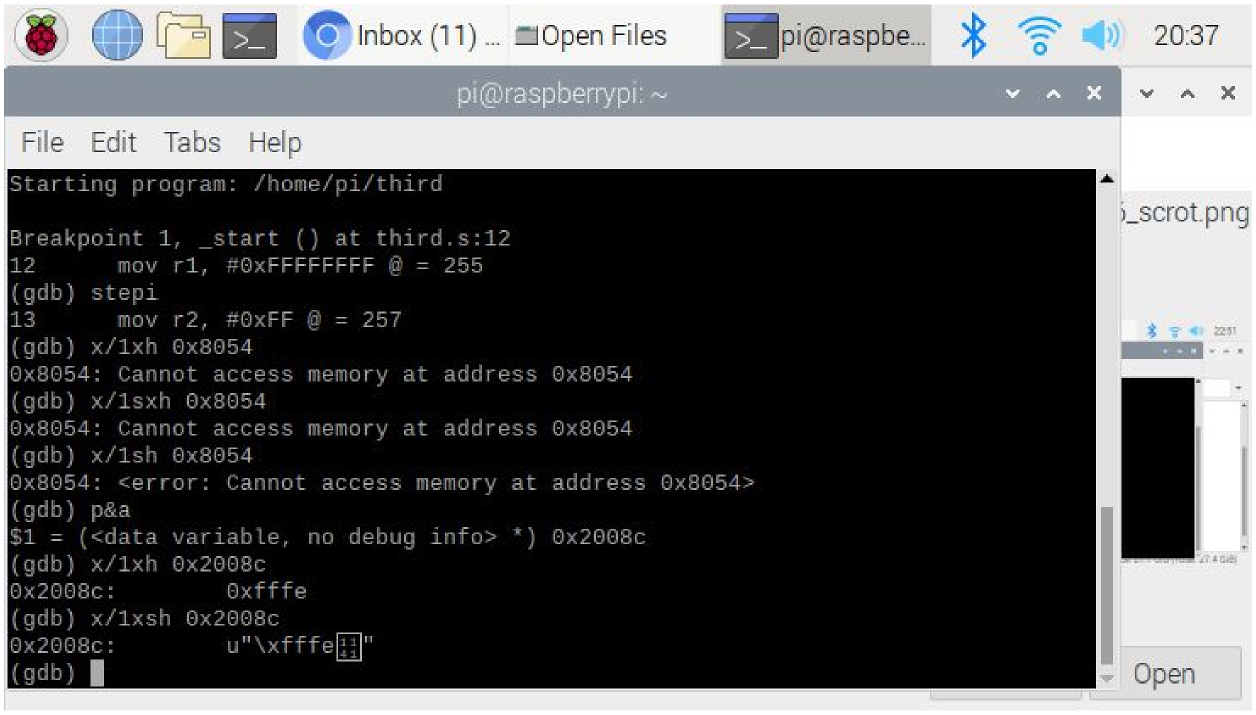
```
  GNU nano 3.2                    third.s

@Third program

.section .data
a:.hword -2 @ 16-bit signed
.section .text
.global _start
_start:

@ The following is a simple ARM code example that attempts to load a set of
@values into registers and it might have problems.
mov r0, #0x1 @ = -1 (signed)
mov r1, #0xFFFFFFFF @ = 255
mov r2, #0xFF @ = 257
mov r3, #0x101 @ = 1024
                              [ Read 18 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
```

To fix the error, .shalfword was changed to .hword which fixed the compilation error and the code was running like it is supposed to run after the next command line was entered.



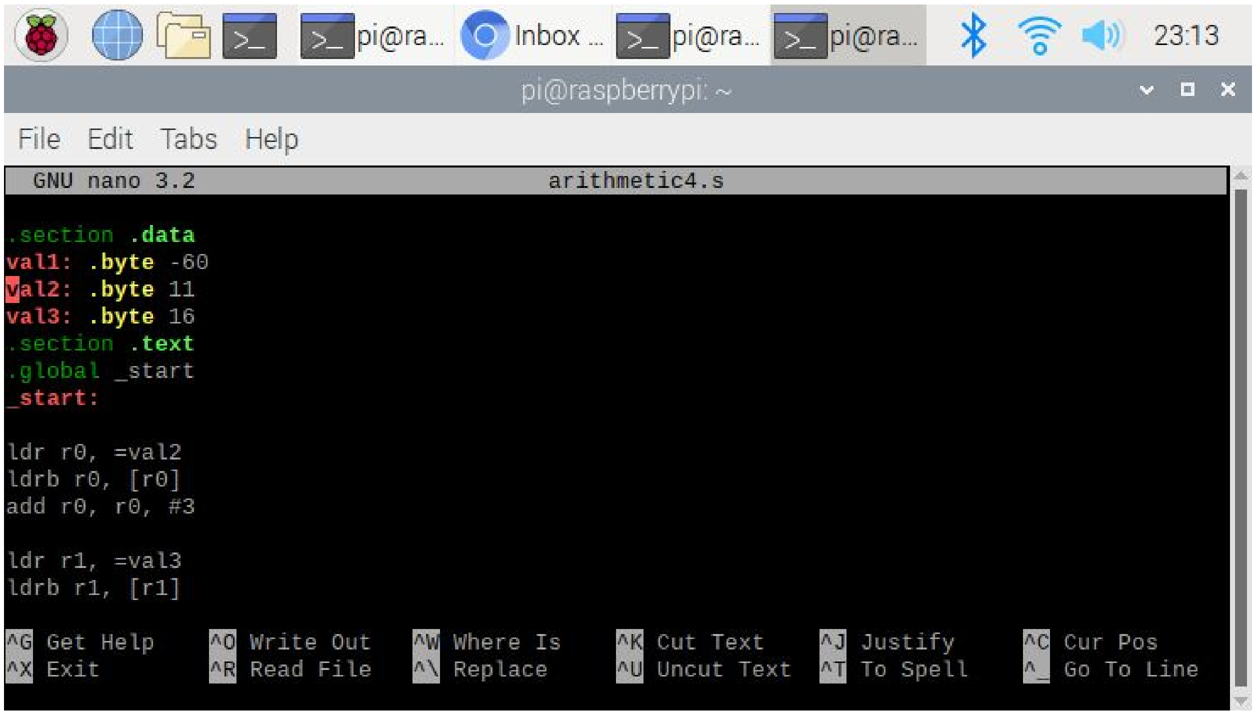
```
Breakpoint 1 at 0x10078: file third.s, line 12.
(gdb) run
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:12
12      mov r1, #0xFFFFFFFF @ = 255
(gdb) stepi
13      mov r2, #0xFF @ = 257
(gdb) x/1xh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1sxh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1sh 0x8054
0x8054: <error: Cannot access memory at address 0x8054>
(gdb) p&a
$1 = (<data variable, no debug info> *) 0x2008c
(gdb) x/1xh 0x2008c
0x2008c:        0xfffe
(gdb)
```

Since the memory address varies, in order to find the memory address p&a was typed in the gdp and from that continue with the next command line.

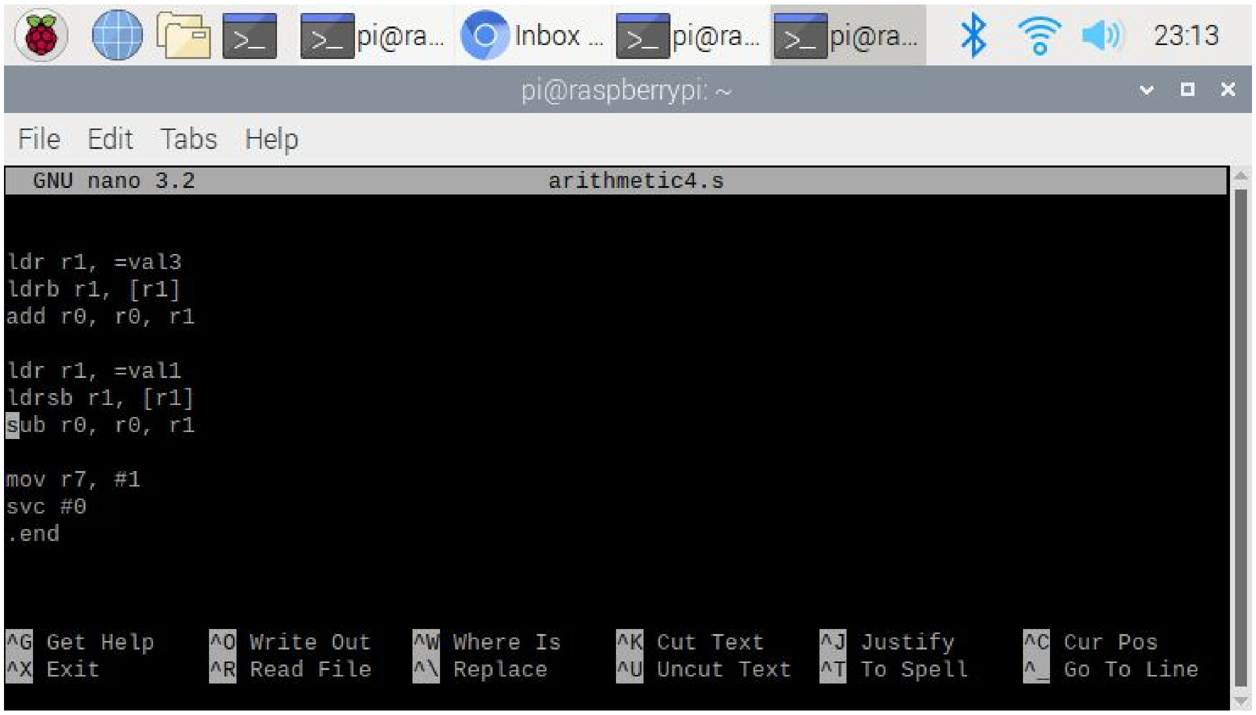


```
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:12
12      mov r1, #0xFFFFFFFF @ = 255
(gdb) stepi
13      mov r2, #0xFF @ = 257
(gdb) x/1xh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1sxh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1sh 0x8054
0x8054: <error: Cannot access memory at address 0x8054>
(gdb) p&a
$1 = (<data variable, no debug info> *) 0x2008c
(gdb) x/1xh 0x2008c
0x2008c:        0xfffe
(gdb) x/1xsh 0x2008c
0x2008c:        u"\xfffe"
(gdb)
```

Just doing x/1xh gives just the hexadecimal value of the memory address and x/1xsh gives the signed hexadecimal values.

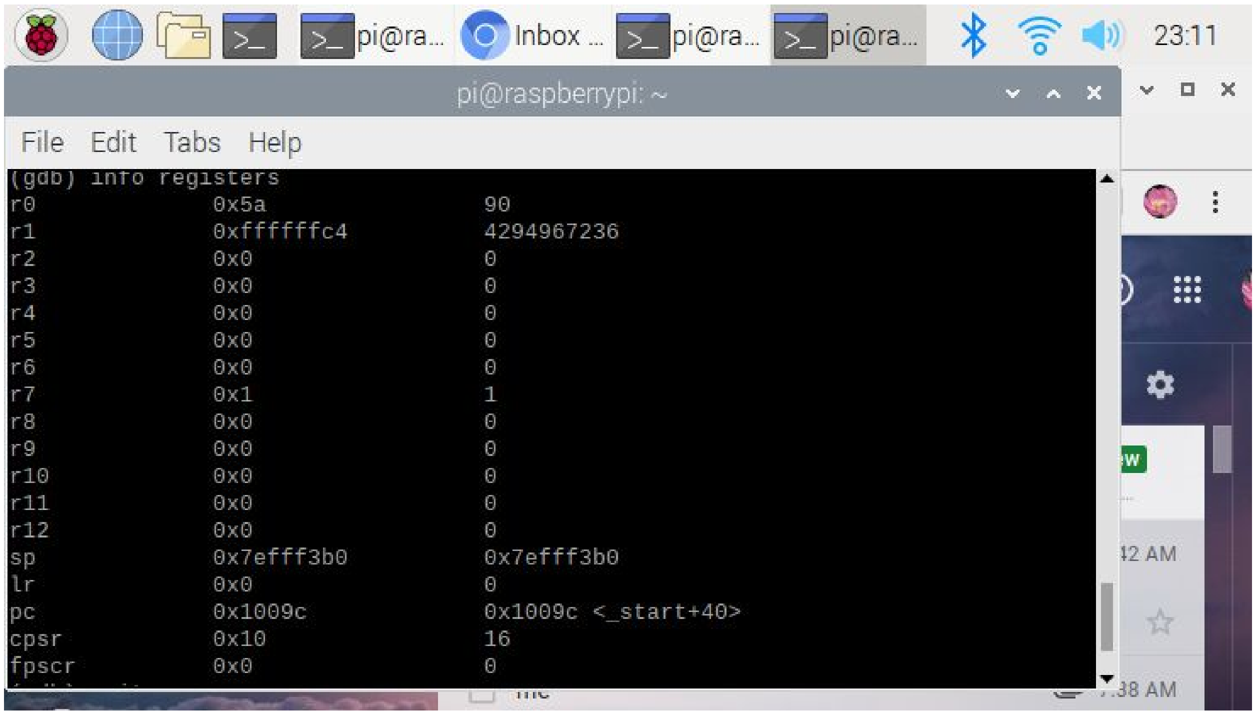


```
GNU nano 3.2                              arithmetic4.s

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

ldr r0, =val2
ldrb r0, [r0]
add r0, r0, #3

ldr r1, =val3
ldrb r1, [r1]

^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
```



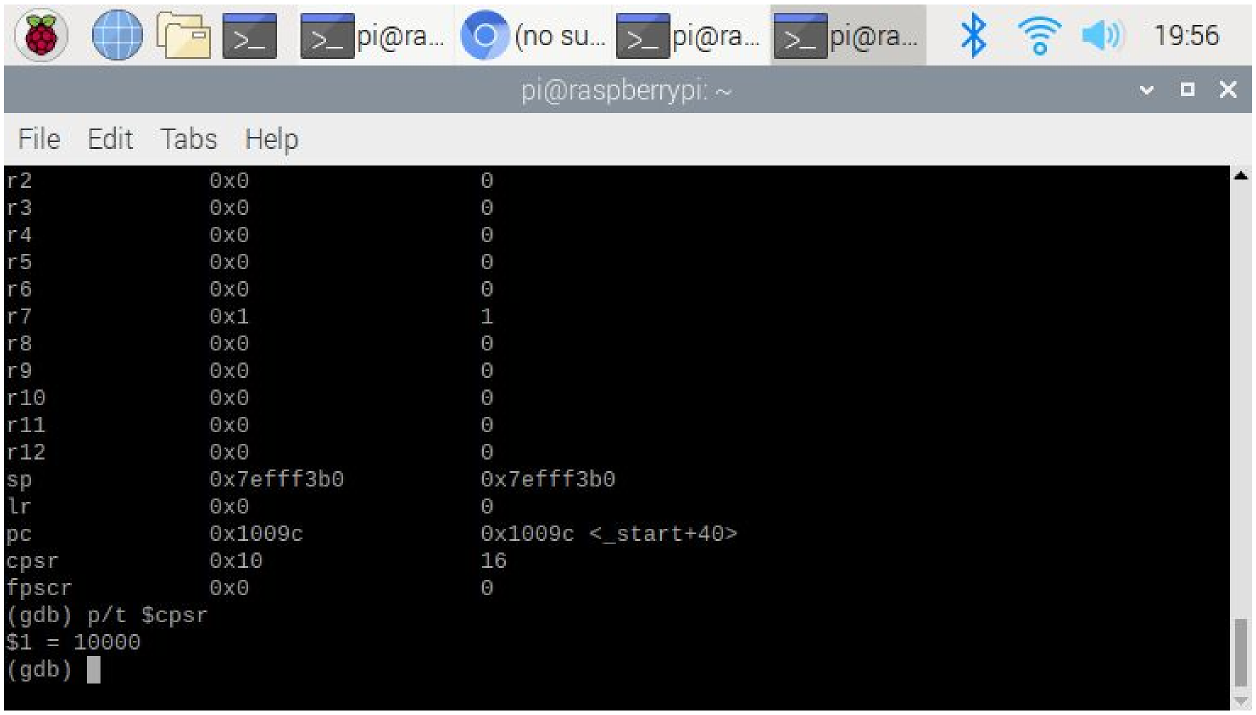
```
GNU nano 3.2                              arithmetic4.s


ldr r1, =val3
ldrb r1, [r1]
add r0, r0, r1

ldr r1, =val1
ldrsb r1, [r1]
sub r0, r0, r1

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
```

Source code named arithmetic4.s



```
(gdb) info registers
r0             0x5a                90
r1             0xffffffc4          4294967236
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
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             0x1009c             0x1009c <_start+40>
cpsr           0x10                16
fpscr          0x0                 0
```

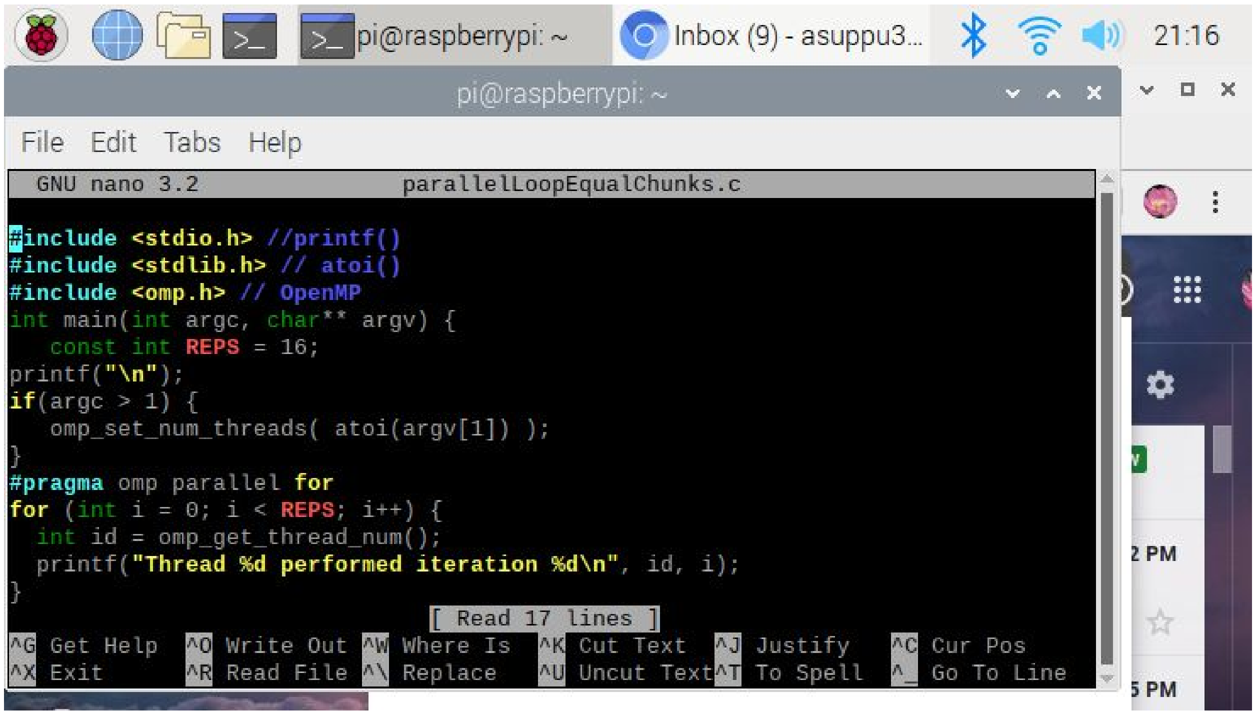
Executing the code for the arithmetic4.s gives a decimal value of 90 in the register.



```
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
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             0x1009c             0x1009c <_start+40>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) p/t $cpsr
$1 = 10000
(gdb)
```

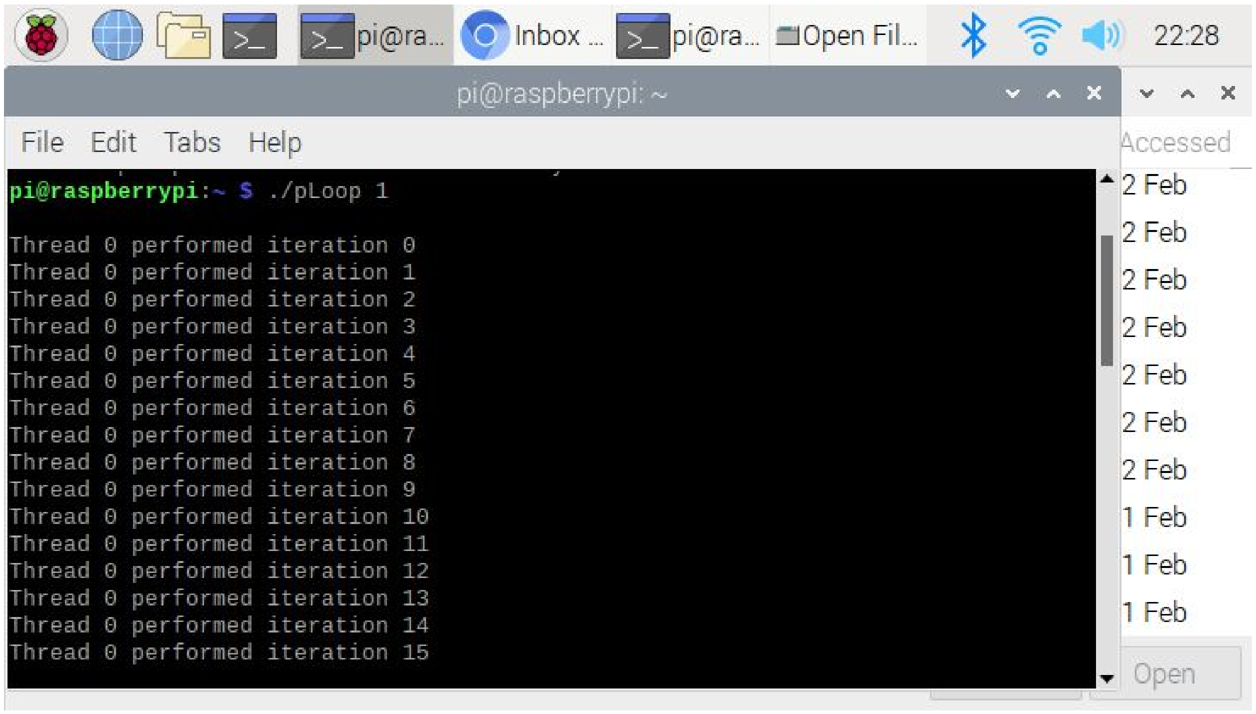
Command p/t $cpsr shows that the only flag set is the interrupt flag and everything else is 0. So fast flag, thumb flag, overflow flag, carry flag, zero flag and negative flag is zero. And only the interrupt flag is 1.

**PARALLEL PROGRAMMING**



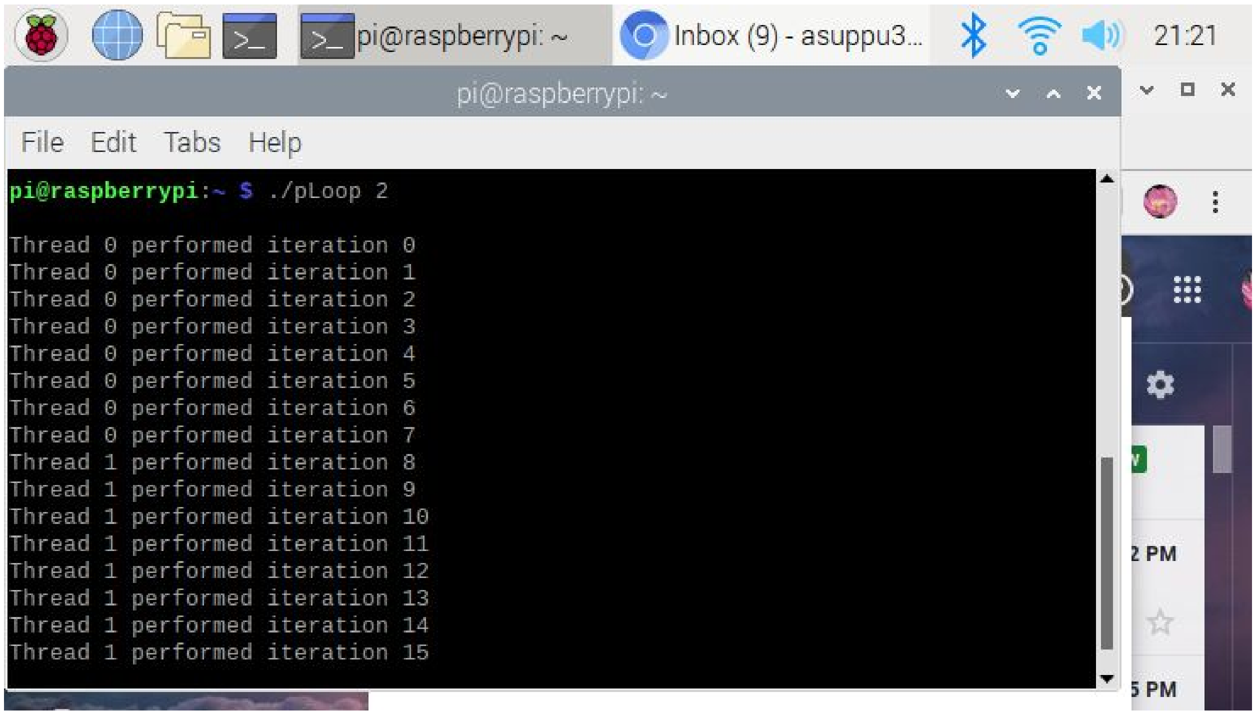
```
GNU nano 3.2                   parallelLoopEqualChunks.c

#include <stdio.h> //printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP
int main(int argc, char** argv) {
   const int REPS = 16;
printf("\n");
if(argc > 1) {
   omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
  int id = omp_get_thread_num();
  printf("Thread %d performed iteration %d\n", id, i);
}
                          [ Read 17 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
```

This is the source code for parallelLoopEqualChunks.c and its the original code without any changes done.



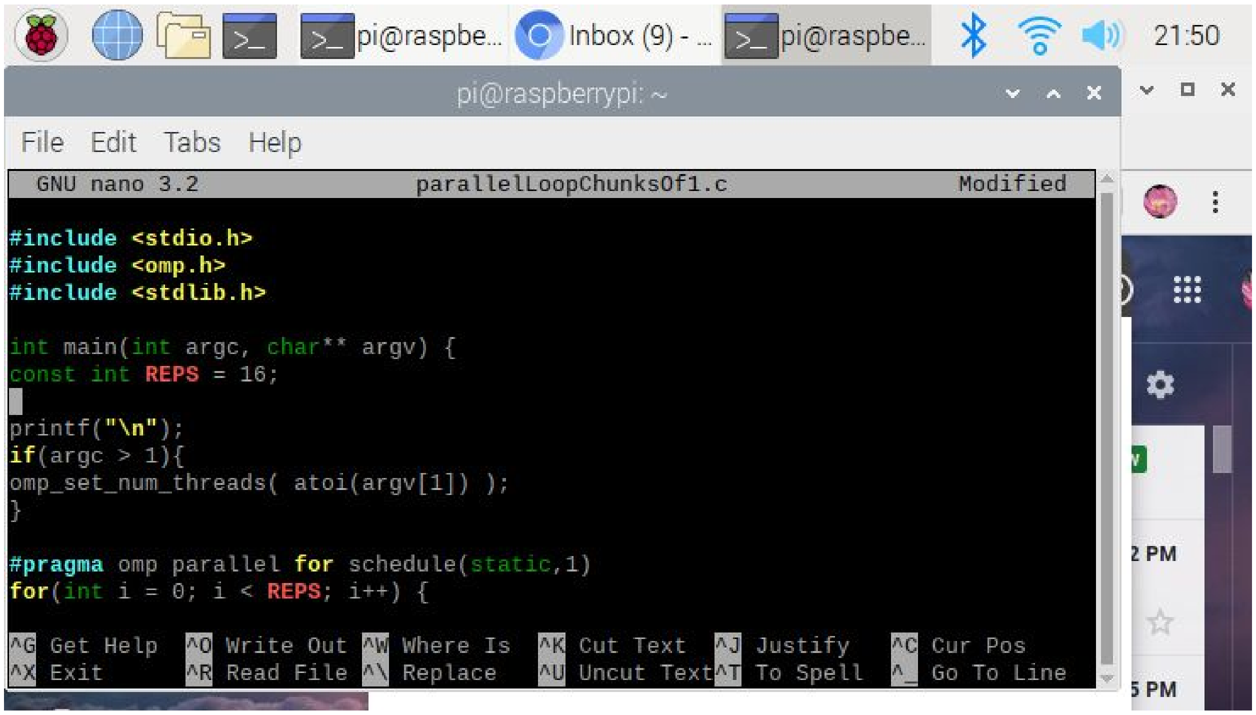
```
pi@raspberrypi:~ $ ./pLoop 1

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

Command ./pLoop 1 gives an output of Thread 0 for all the iterations.



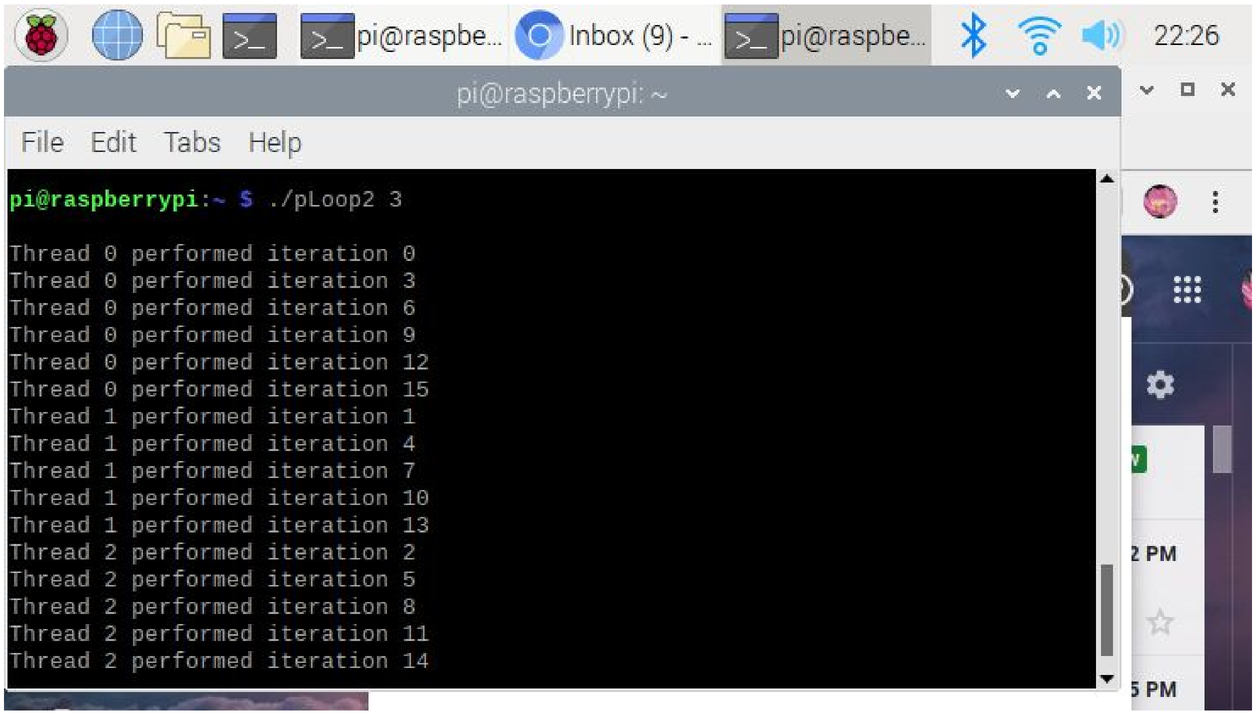
```
pi@raspberrypi:~ $ ./pLoop 2

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 1 performed iteration 8
Thread 1 performed iteration 9
Thread 1 performed iteration 10
Thread 1 performed iteration 11
Thread 1 performed iteration 12
Thread 1 performed iteration 13
Thread 1 performed iteration 14
Thread 1 performed iteration 15
```

Command ./pLoop 2 gives 0 for 8 Threads and gives 1 for all the rest of the Threads. When the number of iterations cannot be divided evenly, the number will be divided as evenly as possible by the compiler. When a number is even, the threads will be divided evenly.



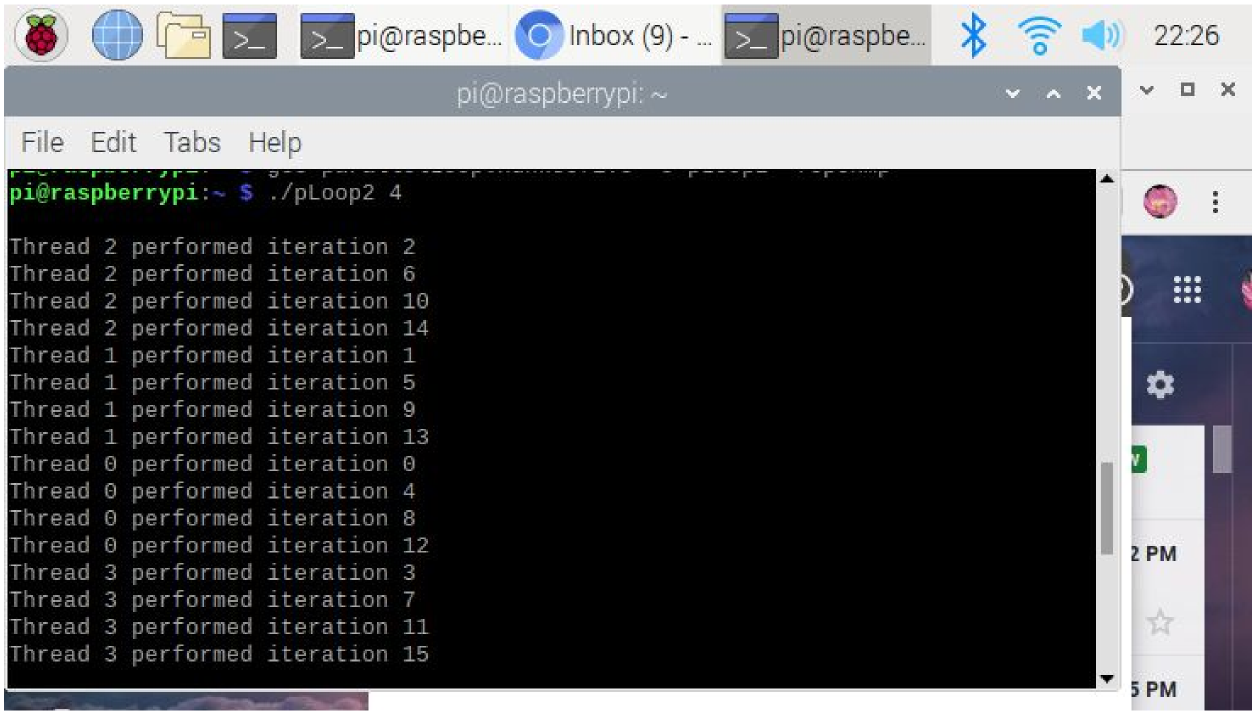
```
  GNU nano 3.2                    parallelLoopChunksOf1.c                Modified

#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++) {

^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
```

This is the source code of parallelLoopChunksOf1.



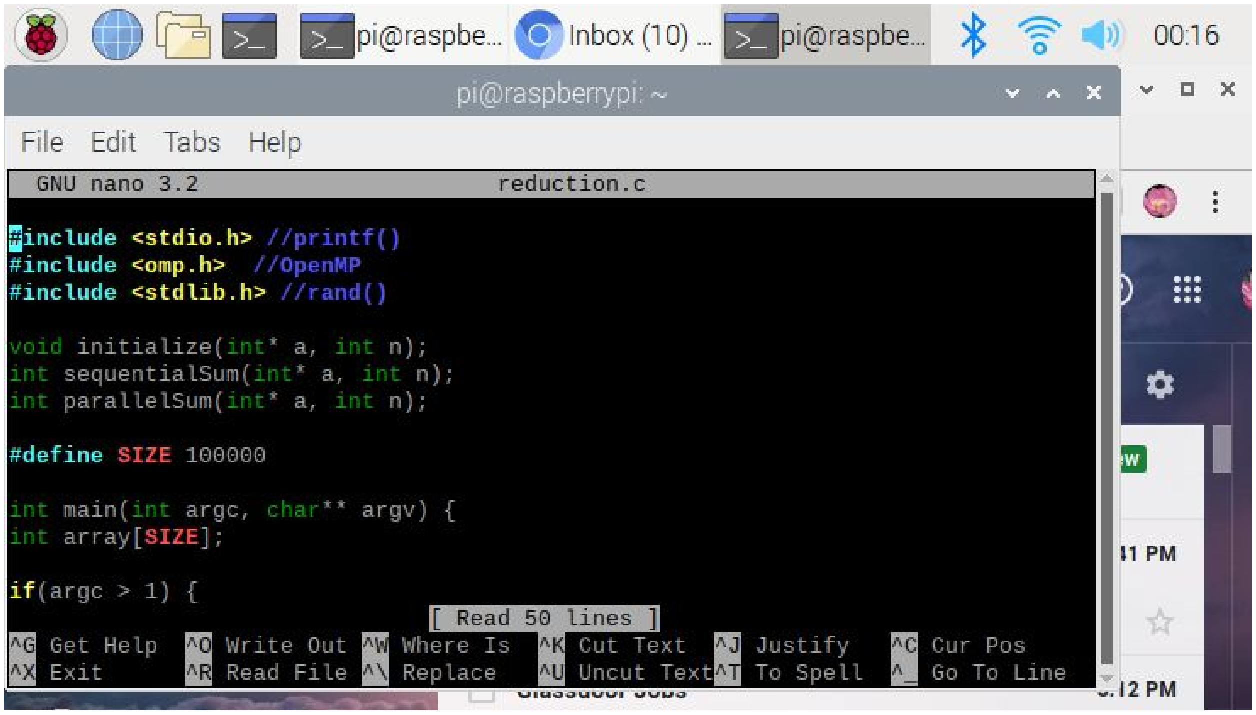
```
pi@raspberrypi:~ $ ./pLoop2 3

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

This is the output when the command ./pLoop2 3 is entered. Three indicates the number of sets of threads will be executed.



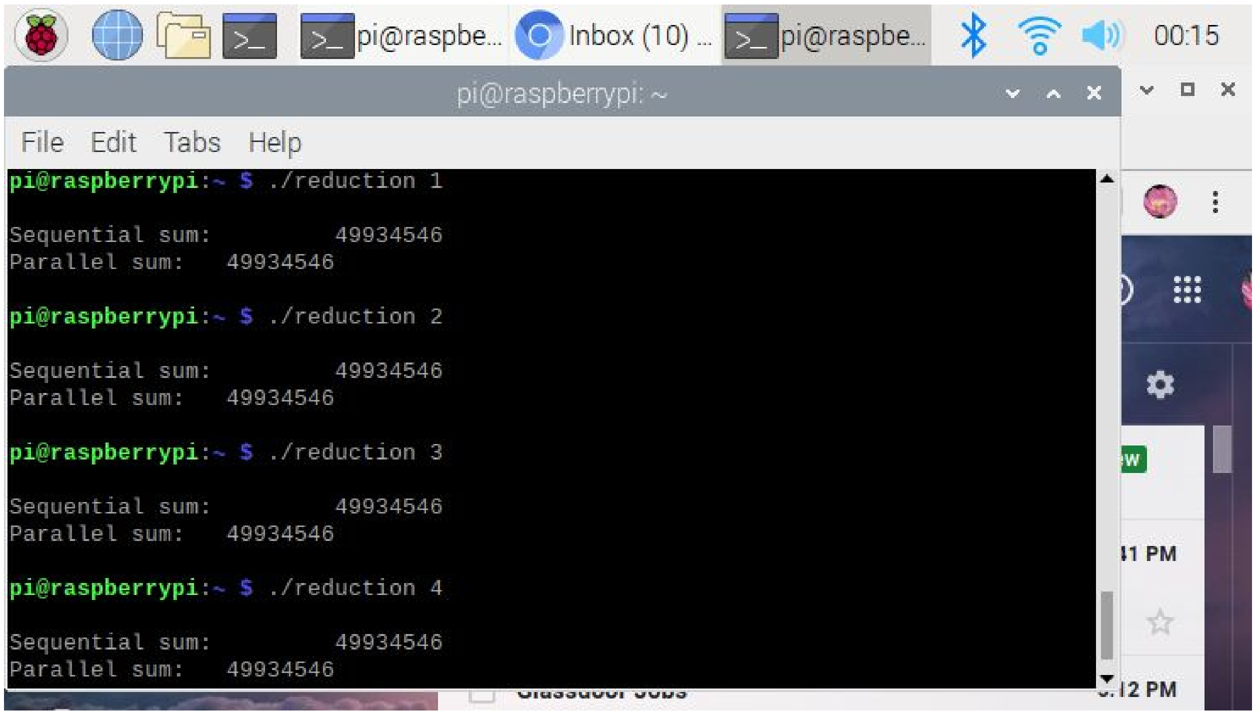
```
pi@raspberrypi:~ $ ./pLoop2 4

Thread 2 performed iteration 2
Thread 2 performed iteration 6
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
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
```

This shows that there are 4 sets of the threads performed which are thread 2 , thread 1, thread 0 and thread 3.



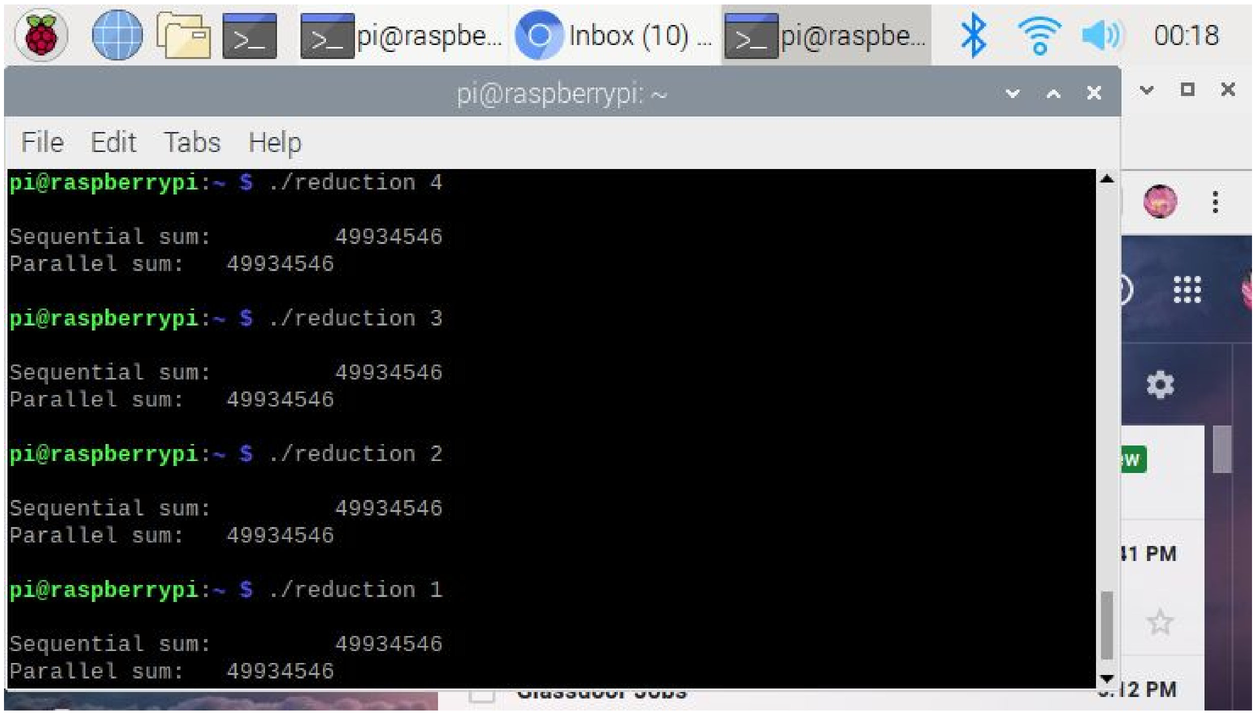
```
  GNU nano 3.2                        reduction.c

#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 100000

int main(int argc, char** argv) {
int array[SIZE];

if(argc > 1) {
                               [ Read 50 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
```

This is the source code for reduction.c



```
pi@raspberrypi:~ $ ./reduction 1

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 4

Sequential sum:         49934546
Parallel sum:   49934546
```

Running the reduction.c source code executes the Sequential sum and Parallel sum.



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

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 2

Sequential sum:         49934546
Parallel sum:   49934546

pi@raspberrypi:~ $ ./reduction 1

Sequential sum:         49934546
Parallel sum:   49934546
```

Removing the line // gives the following output.