Parallel Programming

```
pi@raspberrypi: ~
 GNU nano 3.2                    trap-notworking.c

#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP7
 /* Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.141592653589793238462643383079;
int main(int argc, char** argv) {
 /* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;
double f(double x);
/* parse command-line arg for number of threads */
 if (argc > 1) {
threadcnt = atoi(argv[1]);
 }
                              [ Read 38 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 trap-notworking.c

```
pi@raspberrypi:~ $ nano trap-notworking.c
pi@raspberrypi:~ $
pi@raspberrypi:~ $ nano trap-notworking.c
pi@raspberrypi:~ $ gcc -lm trap-notworking.c -o trap-notworking -fopenmp
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.440373
pi@raspberrypi:~ $
```

Since line 37 of the code is set to shared(a, n, h, integral), the output is 1.440373. And this is because the shared key word indicates that all threads will be using the same copy of the variable in the memory.

```
pi@raspberrypi: ~
GNU nano 3.2                    trap-working.c

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


 /* Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.141592653589793238462643383079;
int main(int argc, char** argv) {
 /* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;
double f(double x);
 /* parse command-line arg for number of threads */
if (argc > 1) {

^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 trap-working.c

```
pi@raspberrypi:~ $ nano trap-working.c
pi@raspberrypi:~ $ gcc -lm trap-working.c -o trap-working -fopenmp
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $
```

In this code, line 37 is being altered to private(i) shared (a, n, h) reduction(+: integral). In this case the threads that will be using the same variables as in the memory is a,n and h. The integral variable has been separated from the shared. So therefore the output is 2.000000.

```
pi@raspberrypi: ~
GNU nano 3.2                          barrier.c

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv) {
printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);
// #pragma omp barrier
printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
}
printf("\n");
return 0;
}
                              [ Read 19 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 barrier.c with line 31 (#pragma omp barrier) being commented.

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

Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
```

This is the output with line 31 is commented. It shows that the threads are not aligned and are random. Since the barrier has not been mentioned, the threads execute randomly.

```
pi@raspberrypi: ~

  GNU nano 3.2                  barrier.c                    Modified

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv) {
printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);
#pragma omp barrier
printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
}
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
```

This is the source code for barrier.c where line 31 is not commented.

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

Thread 1 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
```

This is the output after the comments on line 31 is removed. In this it can be seen that the first four lines display “Before” and the last four lines display “After”. Because the barrier has been uncommented, it allows the Before threads to execute before the After threads.

```
pi@raspberrypi: ~
 GNU nano 3.2                          masterWorker.c

#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP
int main(int argc, char** argv) {
printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]) );
}
// #pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
if ( id == 0 ) { // thread with ID 0 is master
printf("Greetings from the master, # %d of %d threads\n",
id, numThreads);
} else { // threads with IDs > 0 are workers
printf("Greetings from a worker, # %d of %d threads\n",
id, numThreads);
}

^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 masterWorker.c where the line 31(#pragma omp

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

Greetings from the master, # 0 of 1 threads
```

When line 31 is commented, the output results into #0 from only 1 thread so only one thread is displayed on the screen because the master is equal to 0 when the pragma is commented.

```
pi@raspberrypi: ~
 GNU nano 3.2                          masterWorker.c                       Modified

#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP
int main(int argc, char** argv) {
printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
if ( id == 0 ) { // thread with ID 0 is master
printf("Greetings from the master, # %d of %d threads\n",
id, numThreads);
} else { // threads with IDs > 0 are workers
printf("Greetings from a worker, # %d of %d threads\n",
id, numThreads);
}
                              [ Read 23 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 masterWorker.c where line 31 (#pragma omp parallel) is not commented.

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

Greetings from the master, # 0 of 4 threads
Greetings from a worker, # 2 of 4 threads
Greetings from a worker, # 3 of 4 threads
Greetings from a worker, # 1 of 4 threads

pi@raspberrypi:~ $
```

The output prints out Greetings from the master with four threads shown and it's not in order. Making line 31 uncommented makes all four threads be displayed on the screen. Because the thread now is not equal to 0, it will print the worker threads.

<u>Arm Programming</u>

```
pi@raspberrypi: ~
  GNU nano 3.2                    fourth.s

@ Fourth program
@ This program compute the following if statement construct:
@ intx;
@ inty;
@ if(x == 0)
@ y = 1;
.section .data
x: .word 0 @ 32-bit signed integer, you can also use int directive instead of .$
y: .word 0 @ 32-bit signed integer,
.section .text
.globl _start
_start:
ldr r1, =x @ load the memory address of x into r1
ldr r1, [r1] @ load the value x into r1
cmp r1,#0 @
beq thenpart @ branch (jump) if true (Z==1) to the thenpart
b endofif @ branch (jump) if false to the end of IF statement body (branch alwa$
thenpart: mov r2,#1
ldr r3, =y @ load the memory address of y into r3
                              [ Read 24 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
```

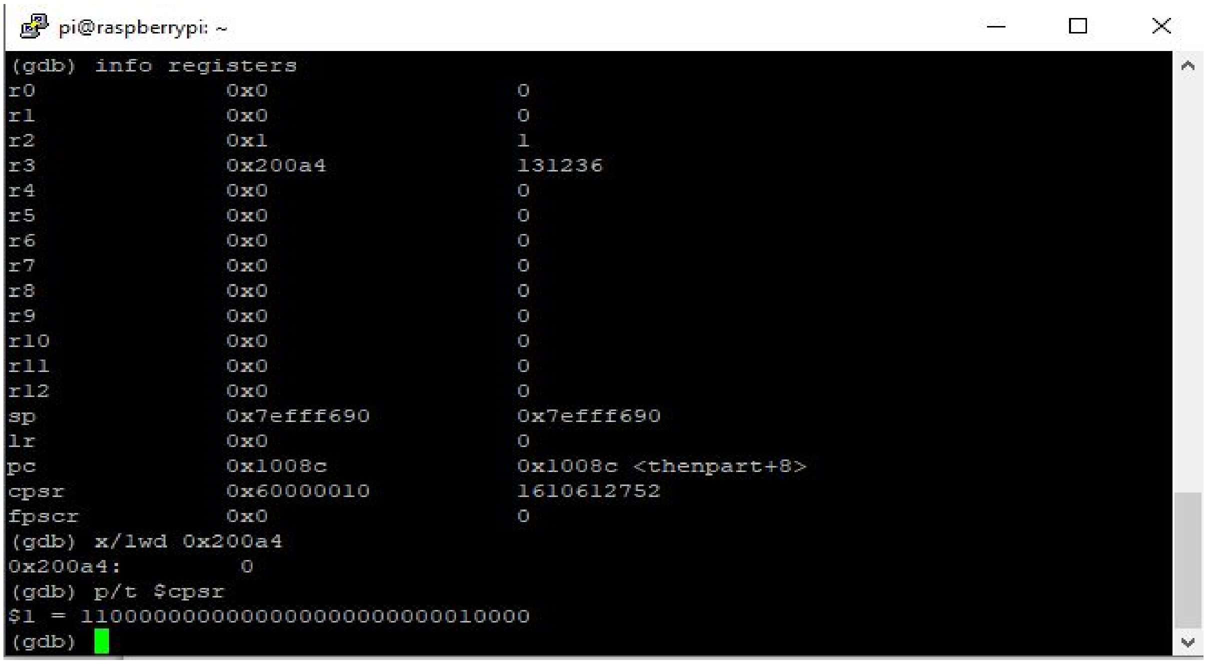
This is the source code of fourth.s without any code modification.

```
pi@raspberrypi: ~
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x1                 1
r3             0x200a8             131240
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             0x7efff690          0x7efff690
lr             0x0                 0
pc             0x10090             0x10090 <thenpart+8>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
(gdb) x/1wd 0x200a8
0x200a8:        0
(gdb) p/t $cpsr
$1 = 1100000000000000000000000010000
(gdb)
```

This is the output for the following version of the code. Register 2 should be 1 because the mov instruction moves the immediate value of 1 into r2. To find the memory x/1wd 0x200a8 from register 3 was entered to get the y memory which is 0. To find the flags, command p/t $cpsr was executed, where the outputs are thumb, fast, interrupt, overflow, carry, zero and negative. Since the zero flag is marked by 0001, this flag is set to 1.

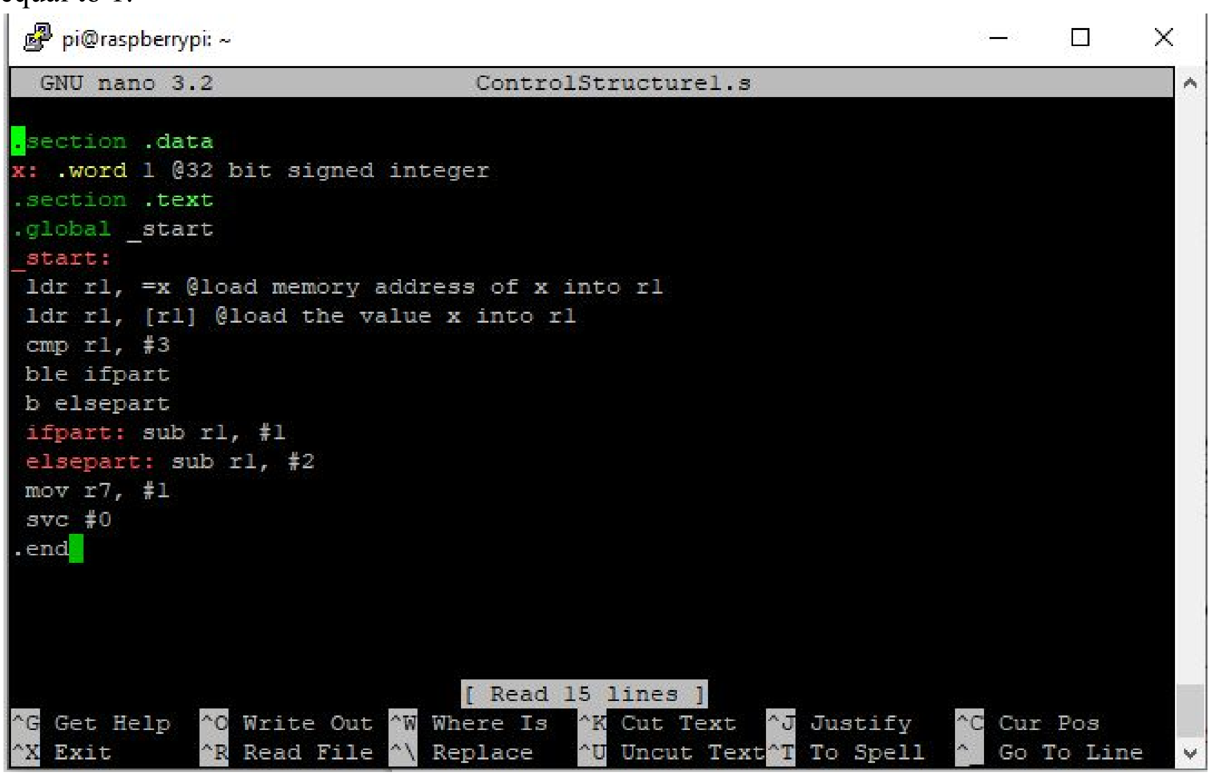
```
pi@raspberrypi: ~
  GNU nano 3.2                        fourth.s

@ Fourth program
@ This program compute the following if statement construct:
@ intx;
@ inty;
@ if(x == 0)
@ y = 1;
.section .data
x: .word 0 @ 32-bit signed integer, you can also use int directive instead of .$
y: .word 0 @ 32-bit signed integer,
.section .text
.globl _start
_start:
ldr r1, =x @ load the memory address of x into r1
ldr r1, [r1] @ load the value x into r1
cmp r1,#0 @
bne thenpart @ branch (jump) if true (Z==1) to the thenpart

thenpart: mov r2,#1
ldr r3, =y @ load the memory address of y into r3

^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 fourth.s with modifications where the keyword beq was replaced with bne and line 17 was removed.



```
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x1                 1
r3             0x200a4             131236
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             0x7efff690          0x7efff690
lr             0x0                 0
pc             0x1008c             0x1008c <thenpart+8>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
(gdb) x/1wd 0x200a4
0x200a4:        0
(gdb) p/t $cpsr
$1 = 1100000000000000000000000010000
(gdb)
```

This is the output after those changes were made. In this we can see that register 2 value has not changed because the goal was to make the code more efficient and not changing any outputs. For accessing the y memory, command x/1wd 0x200a4 was used because now register 3 holds that memory address. So by using this we get that y memory has 0. And the z flag is 0001 which is equal to 1.



```
  GNU nano 3.2                  ControlStructure1.s

.section .data
x: .word 1 @32 bit signed integer
.section .text
.global _start
_start:
 ldr r1, =x @load memory address of x into r1
 ldr r1, [r1] @load the value x into r1
 cmp r1, #3
 ble ifpart
 b elsepart
 ifpart: sub r1, #1
 elsepart: sub r1, #2
 mov r7, #1
 svc #0
.end

                         [ Read 15 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 ControlStructure1.s

```
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
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             0x7efff670          0x7efff670
lr             0x0                 0
pc             0x1008c             0x1008c <elsepart>
cpsr           0x80000010          -2147483632
fpscr          0x0                 0
```

```
(gdb) p&x
$1 = (<data variable, no debug info> *) 0x2009c
(gdb) x/1xw 0x2009c
0x2009c:        0x00000001
(gdb) p/t $cpsr
$2 = 10000000000000000000000000010000
(gdb) 
```

By using the following lines of code, register 1 should be 0 as we see in the screenshot above. To get the memory address, command p&x was accessed so we got the memory address to be 0x2009c and then x/1xw was used to get the results in hex. Later, to find the z flag, command p/t $cpsr was used which gives us the result that the z flag is 1 because x is 0.

Parallel Programming Questions

Race Condition

What is race condition?

- Race condition is when the output of a program is based on order and timing of the processes that are not controlled by any occurence.

Why is race condition difficult to reproduce and debug?

- Race condition is difficult to reproduce and debug because since it is based on a non controllable occurrence, the errors that are produced by the race condition will not be displaced, which is why it is difficult to understand in what type of situations the race conditions output an error so it is difficult to debug.

How can it be fixed? Provide an example from your Project_A3 (see spmd2.c).

- This can be fixed by either using processes that do not run in an orderly manner or coding in a way that processes which are performed orderly are not running concurrently. For example, in spmd2.c, the threads did not have any processes depending on one another so without throwing any errors, the code ran in a concurrent manner.

Summaries the Parallel Programming Patterns section in the “Introduction to Parallel Computing_3.pdf” (two pages) in your own words (one paragraph, no more than 150 words).

- Based on this reading, there are two different concurrent execution patterns which are process or thread control patterns which is where it guides how the execution of processing units on the hardware are controlled at run time and coordination pattern which is where it sets up many simultaneously running tasks coordinatie to finish the operation. And in this pattern there are two subdivisions which are message passing and mutual exclusions.

In the section “Categorizing Patterns” in the “Introduction to Parallel Computing_3.pdf” compare the following:

Collective synchronization (barrier) with Collective communication (reduction) Master-worker with fork join.

- Collective synchronization arranges the tasks by classifying if the forked threads have done majority enough to go through a ‘barrier’ check in order for it to be together. Collective communication has a function called “reduction” which steadily follows the progression of the processes and stores the data and the threads will be forked and it will then be used to give an output. The Master-worker with fork joining is when the “master” thread, which is only one thread moves one section of code when the fork is performed and when the other threads which are the ‘workers’ move different sections of code when they are forking. Then the fork joining is when the whole code forks into different threads, then groups them without seeing if the threads independent or dependent

Dependency: Using your own words and explanation, answer the following:

Where can we find parallelism in programming?

- Parallelism in programming is found when the central processing unit (CPU) carries out the processes all at ones.

What is dependency and what are its types (provide one example for each)?

- Dependency is when the output is based on the order the statements have been executed. The types are the true(flow) dependency in which the second statement is dependent on the first one. For example x = 5 and y = x, in this the second statement cannot be executed with the first statement. And output dependency is like i = f(x) and i = j whereas anti-dependence is when i = j and j = 1 and this is where the first statement is dependent on the second statement.

When a statement is dependent and when it is independent (Provide two examples)?

- A statement is dependent when the statements are based on one another. For example, x = 5; y= x and x = 1; y = x+1 are both dependent since y is based on x so it affects the output results based on the statements order. A statement is independent when the execution order does not have an impact on the output. For example, x = 4; y = 2 and i = 0; j = 1 are both independent since the statements give the same output regardless of the order of execution.

When can two statements be executed in parallel?

- When two statements are not parallel then they can be executed in a parallel manner.

How can dependency be removed?

- For dependency to not occur, making sure the statements are in a way so that the pattern of execution will not affect the output.

How do we compute dependency for the following two loops and what type/s of dependency?

- To compute dependency, firstly change loops into single statements. Then, look at how the loops are dependent. The first loop follows a true dependence pattern since each value is dependent on the previous index value. The second loop also follows true dependency since the output that the b produces is dependent on the output of a.