

22CSC 3210- Computer Organization and Programming
Assignment 4: 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	Parallel Programming	2 hrs	None	3/27/20	g++ -o trap-working trap-working.c -lm -fopenmp
Humaira Ridi	Hridi1@student.gsu.edu	Parallel Programming	2 hrs	None	3/27/20	Remember where ALU flags are located
Kirby Liu	Kliu8@student.gsu.edu	Team Coordinator	Email TA Set up the GitHub Task Plan	None	3/27/20	
Dhruv Parikh	Dparikh5@student.gsu.edu	Zoom Setup	3hrs	Presentation	3/27/20	
Dhananjay Khazanchi	Dkhazanchi1@student.gsu.edu	Video	3hrs	Parallel and ARM Assembly Programming	3/27/20	

Task 2 – Github screenshot:

Assignment 4
Updated 5 minutes ago

Filter cards

4 To Do

- ☒ Parallel Programming Task
Added by Team-0101
- ☒ ARM Assembly Programming
Added by Team-0101
- ☒ YouTube Recording
Added by Team-0101
- ☒ Lab Meeting
Added by Team-0101

2 In Progress

- ☒ Task Table
Added by Team-0101
- ☒ Report Compilation
Added by Team-0101

2 Done

- ☒ Email TA from Team Coordinator
Added by Team-0101
- ☒ Git Hub Project Board
Added by Team-0101

Dhananjay Khazanchi

Task 3 A)- Parallel Programing Foundation

Race condition:

- What is race condition?
 - An event in which the output is dependent on the timing of other uncontrollable events
- Why race condition is difficult to reproduce and debug?
 - The end result is nondeterministic and depends on the timing between interfering threads. Also, when in debugging mode, the issues can often disappear.
- How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
 - A race condition can be fixed by assigning each thread its own private variable, so that the threads are sharing the same memory. This is similar to what was done in spmd2.c when ids and numThreads were created for each thread.

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).

There are many two main patterns found in parallel programming. One type of parallel programming patterns is known as strategies. Strategies can be further broken down into parallel algorithm strategies and implementation strategies. Parallel algorithm strategies focus on utilizing different algorithms to achieve parallelism, such as include data and task decomposition. Implementation strategies focus on the program and data structure of the code. Another category of parallel programming is concurrent execution mechanisms. Concurrent execution mechanisms involve process/thread control and coordination. This deals with how the hardware control the parallel execution at runtime and the setup for multiple concurrent tasks.

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

- Collective synchronization (barrier) with Collective communication (reduction)
 - Collective synchronization forces threads to stop at a point and not proceed until all other threads have reached the point. Collective communication, on the other hand, distributes the work into smaller threads that do a computation and push the result to a larger thread.
- Master-worker with fork join.
 - In master-worker, the main thread generates subproblems for the workers threads, and the worker then returns the result to the master thread. Fork-join is when a program’s execution branches off in parallel (fork) and then joins back to continue sequential execution. In fork-join, there is no master thread.

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

- Where can we find parallelism in programming?
 - Parallelism can be found in the task, data, and resource views. Program views includes statement level, block level, loop level, routine level, and process level.
- What is dependency and what are its types (provide one example for each)?
 - A dependency is when one step depends on a previous step to complete a computation before the program can proceed to the next step.
 - Flow dependency – S1: a = 2; S2: b = a; the second is dependent on the first
 - Output dependency – S1: a = f(x); S2: b = a; the second is dependent on the output of the first

- Anti dependency – S1: a = b; S2: b = 3; first is dependent on the second
- When a statement is dependent and when it is independent (Provide two examples)?
 - A statement is dependent if the order in which the statements are executed matters. (a = 0; a = 4; b = a + 4 will not have the same end result as b = a + 4; a = 4; a = 4)
 - A statement is independent when the order in which the statements are executed does not matter (a = 3; b = 4 will have the same end result as b = 4; a = 3)
- When can two statements be executed in parallel?
 - Two statements can only be executed in parallel only if there are no true, anti, or output dependencies are present.
- How can dependency be removed?
 - Some dependencies can be removed by rearranging statements or eliminating statements.
- How do we compute dependency for the following two loops and what type/s of dependency?
 - In both loops, the statements within the loop are independent, since no dependencies exist between them. The loops themselves are flow dependence because the statements in both loops use the variable in the for loop for computations. In loop 1, since there is only one statement, there are no other dependencies. In loop 2, since the statements are independent from each other, there are no other dependencies as well.

<pre>for (i=0; i<100; i++) S1: a[i] = i;</pre>	<pre>for (i=0; i<100; i++) { S1: a[i] = i; S2: b[i] = 2*i; }</pre>
---	---

Task 3 B)- Parallel Programming Basics

Trap-NotWorking

Code:

```

1  #include <math.h>
2  #include <stdio.h> // printf()
3  #include <stdlib.h> // atoi()
4  #include <omp.h> // OpenMP
5  /* Demo program for OpenMP: computes trapezoidal approximation to an integral */
6  const double pi = 3.141592653589793238462643383879;
7  int main(int argc, char** argv) {
8      double a = 0.0, b = pi; /* limits of integration */;
9      int n = 1048576; /* number of subdivisions = 2^20 */
10     double h = (b - a) / n; /* width of subdivision */
11     double integral; /* accumulates answer */
12     int threadcnt = 1;
13     double f(double x);
14
15     if (argc > 1) {
16         threadcnt = atoi(argv[1]);
17     }
18     #ifdef _OPENMP
19     omp_set_num_threads( threadcnt );
20     printf("OMP defined, threadcnt = %d\n", threadcnt);
21     #else
22     printf("OMP not defined");
23     #endif
24     integral = (f(a) + f(b))/2.0;
25     int i;
26     #pragma omp parallel for private(i) shared (a, n, h, integral)
27     for(i = 1; i < n; i++) {
28         integral += f(a+i*h);
29     }
30     integral = integral * h;
31     printf("With %d trapezoids, our estimate of the integral from\n", n);
32     printf("%f to %f is %f\n", a,b,integral);
33 }
34 double f(double x) {
35     return sin(x);
36 }

```

Screenshots:

```

pi@raspberrypi: ~
File Edit Tabs Help
trap-notworkingdha.c: Error: can't resolve value for symbol 'integral'
pi@raspberrypi:~$ g++ -o trap-notworkingDha trap-notworkingDha.c -lm -fopenmp
pi@raspberrypi:~$ ./trap-notworkingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.367930
pi@raspberrypi:~$ ./trap-notworkingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.502329
pi@raspberrypi:~$ ./trap-notworkingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.474102
pi@raspberrypi:~$ ./trap-notworkingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.444154
pi@raspberrypi:~$ ./trap-notworkingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.577322
pi@raspberrypi:~$

```

The output should be 2, but the program is running into an error because there is no reduction code present to accumulate the values in the individual threads, which is why an incorrect result is being given each time the code is called.

Trap-Working

Code:

[Edit file](#)
[Preview changes](#)

```

1  #include <math.h>
2  #include <stdio.h> // printf()
3  #include <stdlib.h> // atoi()
4  #include <omp.h> // OpenMP
5
6  const double pi = 3.141592653589793238462643383079;
7  int main(int argc, char** argv) {
8  double a = 0.0, b = pi; /* limits of integration */;
9  int n = 1048576; /* number of subdivisions = 2^20 */
10 double h = (b - a) / n; /* width of subdivision */
11 double integral; /* accumulates answer */
12 int threadcnt = 1;
13 double f(double x);
14 if (argc > 1) {
15     threadcnt = atoi(argv[1]);
16 }
17 #ifdef _OPENMP
18     omp_set_num_threads( threadcnt );
19     printf("OMP defined, threadcnt = %d\n", threadcnt);
20 #else
21     printf("OMP not defined");
22 #endif
23 integral = (f(a) + f(b))/2.0;
24 int i;
25 #pragma omp parallel for \
26 private(i) shared (a, n, h) reduction(+: integral)
27 for(i = 1; i < n; i++) {
28     integral += f(a+i*h);
29 }
30 integral = integral * h;
31 printf("With %d trapezoids, our estimate of the integral from %f to %f is %f\n", n, a, b, integral);
32 printf("%f to %f is %f\n", a, b, integral);
33 }
34 double f(double x) {
35     return sin(x);
36 }

```

Screenshot:

```

pi@raspberrypi:~$ nano trap-workingDha.c
Use "fg" to return to nano.

[1]+  Stopped                  nano trap-workingDha.c
pi@raspberrypi:~$ nano trap-workingDha.c
pi@raspberrypi:~$ g++ -o trap-workingDha trap-workingDha.c -lm -fopenmp
pi@raspberrypi:~$ ./trap-workingDha 4
OMP defined, threadcnt = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.573501
pi@raspberrypi:~$ ./trap-workingDha 4
OMP defined, threadcnt = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~$

```

With the addition of the reduction clause, the correct output of 2 is returned because the threads are able to properly communication together.

Barrier

Code:

Branch: master | CSC3210-Team0101 / Assignment4 / Code / Dhananjay / barrierDha.c

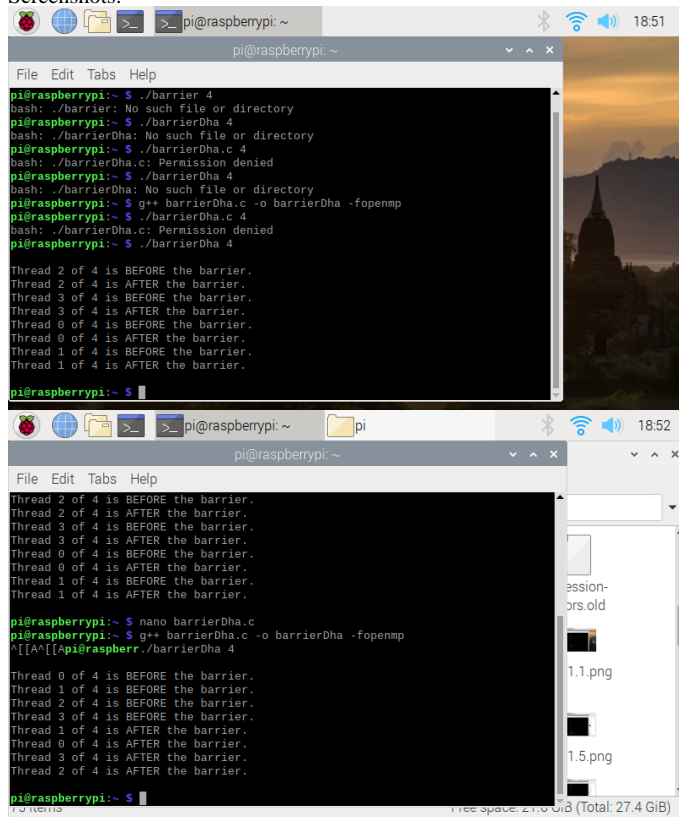
Team-0101 Add files via upload

1 contributor

23 lines (29 sloc) 456 Bytes

```
1 #include <stdio.h>
2 #include <omp.h>
3 #include <stdlib.h>
4
5 int main(int argc, char** argv){
6     printf("\n");
7     if(argc > 1){
8         omp_set_num_threads(atoi(argv[1]));
9     }
10
11     #pragma omp parallel
12     {
13         int id = omp_get_thread_num();
14         int numThreads = omp_get_num_threads();
15         printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);
16
17         #pragma omp barrier
18         printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
19     }
20
21     printf("\n");
22     return 0;
23 }
```

Screenshots:



When the pragma omp parallel line is commented, the result is what is shown in the first screenshot. There is no separation of the before and after sections because no barrier is in place. After the line is uncommented, there is a barrier in place between the before and after sections.

MasterWorker

Code:

Branch: master CSC3210-Team0101 / Assignment4 / Code / Dhananjay / masterWorkerDha.c

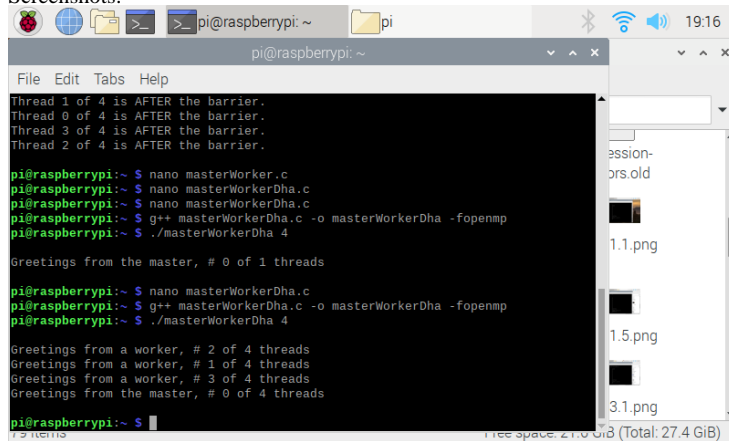
Team-0101 Add files via upload

1 contributor

25 lines (21 sloc) 466 Bytes

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main(int argc, char** argv){
6     printf("\n");
7     if(argc > 1){
8         omp_set_num_threads(atol(argv[1]));
9     }
10
11     #pragma omp parallel
12     {
13         int id = omp_get_thread_num();
14         int numThreads = omp_get_num_threads();
15
16         if(id == 0){
17             printf("Greetings from the master, # %d of %d threads\n", id, numThreads);
18         }else{
19             printf("Greetings from a worker, # %d of %d threads\n", id, numThreads);
20         }
21     }
22
23     printf("\n");
24     return 0;
25 }
```

Screenshots:



In the center of the screenshot, the result, when the pragma omp parallel line is commented, only shows the master thread because the parallel threads are not created. In the result afterwards, the line has been uncommented and shows the master and worker threads.

Task 4- ARM Assembly Programming

Part1

Code:

Branch: master [CSC3210-Team0101 / Assignment4 / Code / Dhananjay / fourthDha.s](#)

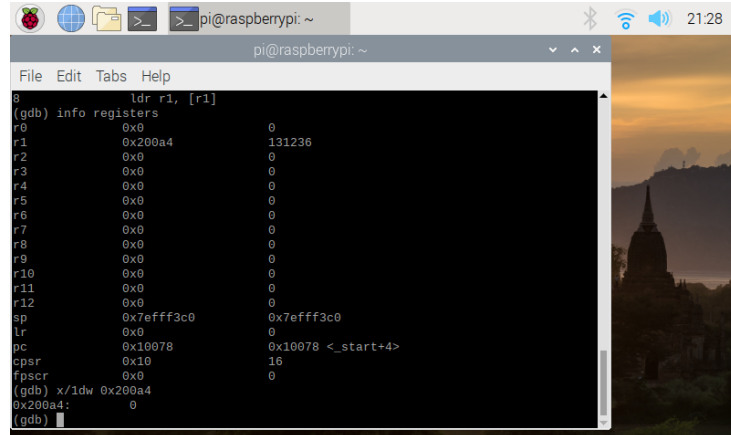
Team-0101 Add files via upload

1 contributor

20 lines (17 sloc) 230 Bytes

```
1  .section .data
2  x: .int 0
3  y: .int 0
4  .section .text
5  .globl _start
6  _start:
7      ldr r1, =x
8      ldr r1, [r1]
9
10     cmp r1, #0
11     beq thenpart
12     b endifif
13
14 thenpart: mov r2, #1
15           ldr r3, =y
16           ldr r2, [r3]
17
18 endifif:  mov r7, #1
19           svc #0
20           .end
```

Screenshots:



Step 1: Since ARM assembly consists of register-register implementation, in order to pull a value from the memory, the memory address must first be loaded on to a register. Similar methods are followed when loading the other memory values into the registers.

```

pi@raspberrypi: ~
File Edit Tabs Help
0x200a4: 0
(gdb) stepi
10      cmp r1, #0
(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      0x7efff3c0 0x7efff3c0
lr      0x0      0
pc      0x1007c   0x1007c <_start+8>
cpsr    0x10     16
fpscr   0x0      0
(gdb)

```

Step 2: The value located at the memory address of the previous step is loaded into r1, and in this case since the value is stored at the address 0x200a4 is 0, the value in r1 is 0. Similar steps are seen when the other memory values are pulled from the memory.

```

pi@raspberrypi: ~
File Edit Tabs Help
fpscr   0x0      0
(gdb) stepi
11      beq thenpart
(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      0x7efff3c0 0x7efff3c0
lr      0x0      0
pc      0x10080   0x10080 <_start+12>
cpsr    0x60000010 1610612752
fpscr   0x0      0
(gdb) p/t $cpsr
0

```

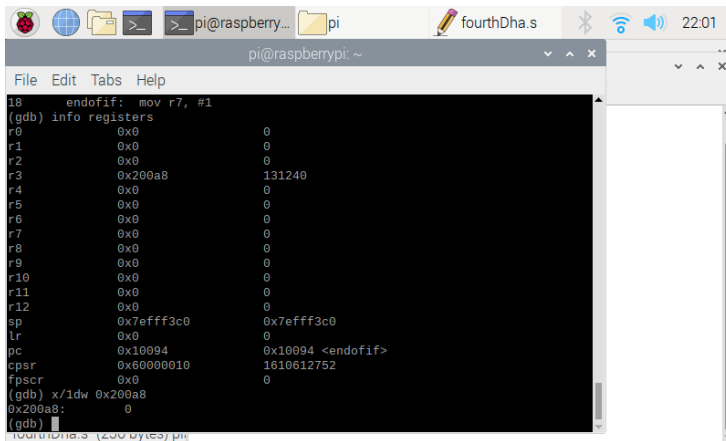
Step 3: In this step, the value in register r1 is being compared to 0. If cmp returns that they are equal (which they are), the code then jumps to thenpart. If cmp returned that they were not equal, the code would jump to endofif. This code does use two jump statements in a row, which is not efficient.

```
(gdb) stepi
thenpart () at fourthDha.s:14
14 thenpart: mov r2, #1
(gdb) info registers
r0          0x0          0
r1          0x0          0
r2          0x0          1
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10088      0x10088 <thenpart>
cpsr        0x60000010    1610612752
fpshr       0x0          0
(gdb)
```

Step 4: In this screenshot, it can be seen that the program jumped to the thenpart section because of the third line.

```
(gdb) stepi
15 ldr r3, =y
(gdb) info registers
r0          0x0          0
r1          0x0          0
r2          0x1          1
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x1008c      0x1008c <thenpart+4>
cpsr        0x60000010    1610612752
fpshr       0x0          0
(gdb)
```

Step 5: The value 1 is loaded into register r2.



```
18      endif: mov r7, #1
(gdb) info registers
r0          0x0          0
r1          0x0          0
r2          0x0          0
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10094      0x10094 <endif>
cpsr       0x60000010    1610612752
fpscr       0x0          0
(gdb) x/1dw 0x200a8
0x200a8: 0
(gdb)
```

Step 6: The memory address of y is loaded onto register r3. Then, the memory address is updated with the value inside r2, which is 0.

Part 2

Code:

Branch: master [CSC3210-Team0101 / Assignment4 / Code / Dhananjay / fourthDha2.s](#)

 Team-0101 Add files via upload

1 contributor

18 lines (16 sloc) 217 Bytes

```
1  .section .data
2  x: .int 0
3  y: .int 0
4  .section .text
5  .globl _start
6  _start:
7      ldr r1, =x
8      ldr r1, [r1]
9
10     cmp r1, #0
11     bne endif
12 thenpart: mov r2, #1
13     ldr r3, =y
14     ldr r2, [r3]
15
16 endif: mov r7, #1
17     svc #0
18     .end
```

Screenshots:

```

(gdb) stepi
thenpart () at fourthDha2.s:12
12  thenpart: mov r2, #1
(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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10084      0x10084 <thenpart>
cpsr       0x60000010    1610612752
fpscr       0x0          0
(gdb)

```

In this program, the same result is achieved as the previous program. However, this program accomplishes the same result with one jump statement instead of two. When 0 is compared to r1, if the result is not equal, then the program jumps to the end. Otherwise, it continues sequentially.

ControlStructure

Code:

Branch: master [CSC3210-Team0101 / Assignment4 / Code / Dhananjay / ControlStructure1Dha.s](#)

Team-0101 Add files via upload

1 contributor

25 lines (21 sloc) 244 Bytes

```

1  .section .data
2  X: .int 1
3  .section .text
4  .globl _start
5  _start:
6      ldr r0, =X
7      ldr r1, =X
8      ldr r0, [r0]
9
10     cmp r0, #3
11     bgt thenpart
12
13     body:
14         sub r0, #1
15         str r0, [r1]
16         b endifif
17
18     thenpart:
19         sub r0, #2
20         str r0, [r1]
21
22     endifif:
23         mov r7, #1
24         svc #0
25         .end

```

Screenshots:

```

Breakpoint 1, _start () at ControlStructure1Dha.s:7
7       ldr r0, [r0]
(gdb) info registers
r0      0x200a4      131236
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      0x7efff3a0   0x7efff3a0
lr      0x0         0
pc      0x10078      0x10078 <_start+4>
cpsr    0x10       16
fpscr   0x0         0
(gdb)

```

Step 1: The memory address of X is loaded onto r1, so that the value will be able to be loaded into r1.

```

fpscr   0x0         0
(gdb) stepi
11      bgt thenpart
(gdb) info registers
r0      0x1         1
r1      0x200a8      131240
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      0x7efff3a0   0x7efff3a0
lr      0x0         0
pc      0x10084      0x10084 <_start+16>
cpsr    0x80000010   -2147483632
fpscr   0x0         0
(gdb)

```

Step 2: The program compares 3 to the value in r1. If the value that results from the comparison is greater than 0, then the program jumps to thenpart.

```

pi@raspberrypi: ~
File Edit Tabs Help
body () at ControlStructure1Dha.s:14
14      sub r0, #1
(gdb) stepi
15      str r0, [r1]
(gdb) info registers
r0      0x0      0
r1      0x200a8   131240
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      0x7efff3a0 0x7efff3a0
lr      0x0      0
pc      0x1008e   0x1008e <body+4>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
Free space: 21.7 GiB (Total: 27.4 GiB)

```

Step 3: Since the result of the comparison is not greater than 0, a jump to thenpart does not occur, and the program moves onto the body section of the code. Then, the program subtracts 1 from r0, resulting in r0 containing a 0.

```

pi@raspberrypi: ~
File Edit Tabs Help
16      b endif
(gdb) info registers
r0      0x0      0
r1      0x200a8   131240
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      0x7efff3a0 0x7efff3a0
lr      0x0      0
pc      0x10090   0x10090 <body+8>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
(gdb) x/1dw 0x200a8
0x200a8: 0
(gdb)
Free space: 21.7 GiB (Total: 27.4 GiB)

```

Step 4: The value of r0 is stored into the memory at the memory address that is being pointed to in r1. The value was successfully stored in the memory by the address 0x200a8 holding a 0.

Thu Vo

Task 3 A)- Parallel Programing Foundation

Race condition:

- What is race condition?
 - A situation when the device or system performs more than two operations at the same time and the output is dependent on the sequence of those uncontrollable events. It becomes a bug when the operations are completed in an incorrect sequence.
- Why race condition is difficult to reproduce and debug?

- Because the output is unpredictable, so sometimes it would take multiple executions for the timing to be exactly right for the bug to show. Therefore, it is hard to find the bug and to reproduce the same failure.
- How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)
 - It can be fixed by logging the program, printing out almost every variable to map what is expected to what's happening. In spmd2.c the bug was fixed by logging the variables inside the threads, providing each thread its own private variable. So, the threads are no longer sharing the same memory location and the error was fixed.

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).

- Parallel programming contains several patterns which are demonstrated best practice ways of producing code that developers of all levels can use. Two distinct patterns are Strategies and Concurrent Execution Mechanisms. In Strategies, which algorithmic strategies one should select and also which implementation strategies should be employed are important. Algorithmic strategies are focused on what tasks can be done concurrently. Implementation is engaged with the overall structure of the program. Concurrent Execution patterns often separate into two main pathways, which are Process/Thread control patterns which determine in which manner processing units of parallel execution on the hardware are controlled and Coordination patterns which focus on the coordination of the parallel computation. The two coordination patterns employed are message passing and mutual exclusion. An upcoming implementation known as hybrid computation employs a use of MPI and OpenMP and has the ability to run on several computers in a given group.

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

- Collective synchronization (barrier) with Collective communication (reduction)
 - Collectives synchronization blocks one part of the statement until all the threads have reached a point, the threads are synchronized in movement. Collective communication, let each thread finished their process, then collects data from each thread and performs an operation on that data to compute a result.
- Master-worker with fork join.
 - In fork join pattern, the master thread executes until a parallel region is encountered then the process will be forked to multiple threads then joined again into a master thread. In master-worker pattern, in a parallel region there will be a master thread that execute a code that different from other threads, but the processes still happened concurrently.

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

- Where can we find parallelism in programming?
 - We can find parallelism in program view, data view, and resource view. Program view: statements of program, blocks of code, loops, routines, processes can be executed at the same time. Data view: How is data operated on? And where does

data reside? Resource view: computers connected by a network or a single computer with multi core processor.

- What is dependency and what are its types (provide one example for each)?
 - Dependency is when one operation depends on a previous operation to be completed before this operation.
 - Flow dependence, read-after-write, when a statement depends on the result of a previous. For example, $a=2$, $b=a$, $c=b$.
 - Output dependence, write-after-write, when the ordering of statements will affect the final output value of a variable. For example, $a=1$, $b=a+1$, $a=5$; the final output of a will depend on the ordering of the statements.
 - Anti-dependence, write-after-read, a statement requires a value that is later updated, For example, $a=1$, $b=a+1$, $a=5$; the output of b depends on whether $a=1$ or $a=5$ executed prior.
- When a statement is dependent and when it is independent (Provide two examples)?
 - A statement is dependent when the outcome is different from the earlier statement executed after this statement. For example, $a=6$, $b=a+1$, $a=0$ will not be the same as $a=6$, $a=0$, $b=a+1$.
 - A statement is independent. The order of execution does not matter. For example, $a=6$, $b=3$ and $b=3$, $a=6$ will produce the same value of a and b .
- When can two statements be executed in parallel?
 - Two statements can be executed in parallel when the two have no dependencies from each other or can be modified to have no dependencies like rearranging statements or eliminating statements.
- How can dependency be removed?
 - Dependency can be removed by rearranging statements or eliminating statements.

$a=6$	$a=6$
$b=a+1$	$b=a+1$
$a=0$	$c=0$

By changing the last statement anti-dependency was removed, but the flow dependency is still there

- How do we compute dependency for the following two loops and what type/s of dependency?
 - In both loops the statements and all iterations are independent on each other but depends on the index of the loop. Since the index i was write first then read by $S1$ and $S2$, so it is read-after-write a.k.a. flow dependency

$i=0$ $a_i = i$	$i=0$ $a_i = i$ $b_i = 2*i$
--------------------	-----------------------------------

<pre>for (i=0; i<100; i++) S1: a[i] = i;</pre>	<pre>for (i=0; i<100; i++) { S1: a[i] = i; S2: b[i] = 2*i; }</pre>
---	---

Thu Vo
Task 3 B)- Parallel Programming Basics

trap-notworking.c

```
pi@raspberrypi:~$ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.440111
pi@raspberrypi:~$ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.405286
pi@raspberrypi:~$ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.424103
pi@raspberrypi:~$ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.461555
pi@raspberrypi:~$
```

- The output is different every time the program is run. This error is similar to the error of reduction.c in project A3, where the output of the threads is not been accumulated together.

trap-working.c

```
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:~$ ./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:~$ ./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:~$
```

- The output is consistent every time the program is run after adding accumulator variable. The prior program missing the concept of integral, which mean adding all the area of the trapezoid to find the overall are of the function from point a to b

barrier.c

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

- With the line 31 commented. `// #pragma omp barrier`
- Each AFTER statement is between BEFORE statements.

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

- Without the line 31 commented. `#pragma omp barrier`
- With the barrier all the BEFORE statements been executed first, before all the AFTER statements.

masterWorker.c

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

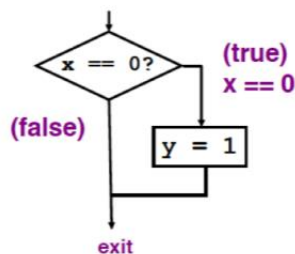
- All the workers threads, thread 1, 2, and 3 execute the same code, while the master thread, thread 0 executed a different code. The program using an if-else condition statement to determine the master thread.

Thu Vo

Task 4- ARM Assembly Programming

fourth.s

```
section .data
x: .int 0
y: .int 0
section .text
.globl _start
_start:
    ldr r1, =x
    ldr r1, [r1]
    cmp r1, #0
    beq thenpart
    b endifif
thenpart: mov r2, #1
          ldr r3, =y
          ldr r2, [r3]
endifif: mov r7, #1
```

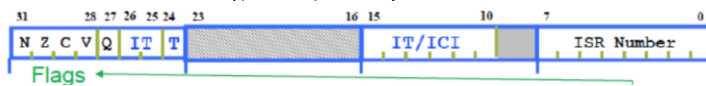


```

(gdb) stepi
thenpart () at fourth.s:14
14      thenpart: mov r2, #1
(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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10088       0x10088 <thenpart>
cpsr       0x60000010    1610612752
fpscr       0x0          0

```

This is the info registers after the start section of the code been run, which mean this after the value of x loaded into register r1, the cmp instruction and the branch instructions.



The status is in hex, the value is 6, 6h=> 0110b. cmp 0, 0 so, 0-0 = 0
N=0, because the result is not negative
Z=1, the result is equal to zero
C=1, because -0 in 2's compliment will result in a carry over
V=0, the numbers are unsinged

```

(gdb) stepi
15          ldr r3, =y
(gdb) stepi
16          ldr r2, [r3]
(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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10090      0x10090 <thenpart+8>
cpsr       0x60000010    1610612752

```

Since `x == 0` is true and `Z=1` the program executed the thenpart, which set the value for y equal to 1

```

(gdb) stepi
19          svc #0
(gdb) stepi
[Inferior 1 (process 3386) exited normally]
(gdb)

```

The program continues to execute the endif part.

fourth2.s

```

section .data
x: .int 0
y: .int 0
section .text
globl _start
_start:
    ldr r1, =x
    ldr r1, [r1]
    cmp r1, #0
    bne endifif
thenpart: mov r2, #1
          ldr r3, =y
          ldr r2, [r3]
endifif: mov r7, #1
          svc #0
          .end

```

`bne endifif`, if the x is not equal to 0 then jump to endifif, which will skip the assigned y=1 part.

```

fpscr      0x0      0
(gdb) stepi
11      bne endofif
(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      0x7efff3c0  0x7efff3c0
lr      0x0      0
pc      0x10080     0x10080 <_start+12>
cpsr    0x60000010  1610612752
fpscr    0x0      0
(gdb)

```

This is after the start part.

```

12      thenpart: mov r2, #1
(gdb) stepi
13      ldr r3, =y
(gdb) info registers
r0      0x0      0
r1      0x0      0
r2      0x1      1
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      0x7efff3c0  0x7efff3c0
lr      0x0      0
pc      0x10088     0x10088 <thenpart+4>
cpsr    0x60000010  1610612752
fpscr    0x0      0
(gdb)

```

Because `x` not equal to 9 is false, so the program did not jump over the thenpart.

ControlStructure1.s

```

if X <= 3
    X = X - 1
else
    X = X - 2

```

```

section .data
x: .int 1
section .text
.globl _start
_start:
    ldr r0, =x
    ldr r1, =x
    ldr r1, [r1]
    cmp r1, #3
    bgt else
    sub r1, #1
    str r1, [r0]
    b end
else:
    sub r1, #2
    str r1, [r0]
end:
    mov r7, #1
    svc #0
end

```

If x is greater than 3 is true, which is opposite from the statement $x \leq 3$, then the program will jump to else part and skip $x = x - 1$ statement. If x is greater than 3 is false, which will make $x \leq 3$ true, then the program will execute the rest of the start part and skip over the else part.

Assume that X is 32-bit integer memory variables and assign X 1.

```

(gdb) x/1dw 0x200a8
0x200a8:      1
(gdb)

```

The value in X is 1 before the if statement

```

10          cmp r1, #3
(gdb) stepi
11          bgt else
(gdb) info registers
r0          0x200a8          131240
r1          0x1             1
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          0x10084         0x10084 <_start+16>
cpsr       0x80000010      -2147483632
fpscr       0x0            0
(gdb)

```

After loaded X into r1 and the address of X is stored in r0.

```

15          b end
(gdb) info registers
r0          0x200a8          131240
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          0x7efff3b0      0x7efff3b0
lr          0x0             0
pc          0x10090         0x10090 <_start+28>
cpsr       0x8000010       -2147483632
fpscr      0x0             0
(gdb) x/1dw 0x200a8
0x200a8:    0
(gdb)

```

Since $X=1$ and $X \leq 3$ is true then $X=X-1$, this will result in X equal to 0.

```

(gdb) stepi
15          b end
(gdb) stepi
end () at ControlStructure1.s:22
22          mov r7, #1
(gdb)

```

The program did not execute the else part, but instead just ended.

Kirby Liu

Task 3 A)- Parallel Programing Foundation

Race condition:

- **What is race condition?** Race condition takes place when output has dependencies that takes place in an order other than the intended timing or sequence causing bugs and errors.
- **Why race condition is difficult to reproduce and debug?** Race conditions may not always produce the same result since it is the timing of sequences that may produce an error and so is not always consistent. Producing an error may require multiple runs before even finding race condition.
- **How can it be fixed? Provide an example from your Project_A3 (see spmd2.c)**
Assigning each thread its own private variable, in the case of Project_A3 ids and numThreads, ensuring that the shared threads share the same memory can help reduce race condition error.

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).

Writing parallel programs, user should consider strategies and concurrent execution mechanisms. There are sometimes unintended errors called race conditions that programmers have to be mindful about when coding. When programming, dependencies and independent statements are important to parallel programming for the user to know type and how to remove dependencies.

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

- **Collective synchronization (barrier) with Collective communication (reduction)**
Collective synchronizatin blocks the movement of the statement and synchronizes the threads until a common break point. Collective communication requires each thread to finish their individual processes so that the processes are communicated together to compute a result.
- **Master-worker with fork join**
The fork join takes multiple threads that work concurrently through the assigned tasks to join together after the threads have completed their process to compute. Master-worker has multiple threads that work concurrently but assigns one task that has different instructions than the other tasks.

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

- **Where can we find parallelism in programming?**
We can find parallelism in Program view, Data view, Resource view, Block level / Loop level / Routine level / Process level
- **What is dependency and what are its types. (provide one example for each)?**
Dependency is where an operation requires another operation to be completed before starting.
True (Flow) dependence a dependency reliant on the result of a previous statement. For example. $a=1$, $b=a$, $c=b$, $d=c$.
Output dependence, where the ordering of statements will affect final output of a variable. $a=5$, $b=a+3$, $a=6$; the final output of a will depends on the ordering of the statements.
Anti-dependence, a statement requires a value which is later updated. $x=2$, $y=x+2$, $x=4$; the output of y depends on whether $x=2$ or $x=4$ executed prior
- **When a statement is dependent and when it is independent (Provide two examples)?**
When the order of statements matter, the statements are dependent, and when the opposite is true where the order does not matter then the statements are independent.
Statement 1;
Statement 2;
Being the same as:
Statement 2;
Statement 1;
Indicates Independent Statements.
- **When can two statements be executed in parallel?** When two statements have no dependency, they can be executed in parallel.

- **How can dependency be removed?**
You can remove dependency by either Rearranging or Eliminating statements.
- **How do we compute dependency for the following two loops and what type/s of dependency?** Statements S1 and S2 are independent as the order of the statements does not change the execution. However, the dependencies of both statements is dependent on the for loop. The type of dependencies for the I loop is the flow dependency.

```

for (i=0; i<100; i++)      for (i=0; i<100; i++) {
    S1: a[i] = i;           S1: a[i] = i;
                           S2: b[i] = 2*i;

```

Kirby Liu

Task 3 B)- Parallel Programming Basics

trap-notworking.c-Screenshot

```

pi@raspberrypi: ~/lab4
File Edit Tabs Help
GNU nano 3.2 trap-notworking.c
#pragma omp parallel for private(i) shared (a, n, h, integral)
for (i=1; i<n ; i++){
    integral += f(a*i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a, b, integral);
}

double f(double x){
    return sin(x);
}

pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.405286
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.424103

```

trap-notworking.c-The totals differ because threads were not totaled together and so each thread's results are not put together to come up with the correct answer of 2.

trap-working.c-Screenshot

```
pi@raspberrypi: ~/lab4
File Edit Tabs Help
GNU nano 3.2 trap-working.c Modified

#pragma omp parallel for \
private(i) shared (a, n, h, integral)
for (i=1; i<n ; i++){
integral += f(a+i*h);
}

integral = integral * h;
printf( "With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a, b, integral);
}

double f(double x){
return sin(x);
}

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:~ $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
```

trap-working.c—Here we see that since the total from the different threads were able to work together by totaling the sums of the threads results.

barrier w/o #pragma.c—Screenshot

```
pi@raspberrypi: ~/Downloads
File Edit Tabs Help

pi@raspberrypi:~/Downloads $ ./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:~/Downloads $ ./trap-notworking 4
OMP defined, threadct = 4
With1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.415506
pi@raspberrypi:~/Downloads $ g++ barrier.c -o barrier -fopenmp
pi@raspberrypi:~/Downloads $ nano barrier.c
pi@raspberrypi:~/Downloads $ ./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 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.

pi@raspberrypi:~/Downloads $ nano barrier.c
pi@raspberrypi:~/Downloads $
```

barrier w/o #pragma- The “#pragma omp barrier” is commented so the first #pragma forks each thread and in the same thread prints “BEFORE” and before proceeding with printing “AFTER”, with each thread printing the “BEFORE” and “AFTER” sequentially.

barrier w/ #pragma.c-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
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 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.

pi@raspberrypi:~/Downloads $ nano barrier.c
pi@raspberrypi:~/Downloads $ g++ barrier.c -o barrier -fopenmp
pi@raspberrypi:~/Downloads $ ./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 0 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.

pi@raspberrypi:~/Downloads $

```

barrier w/ #pragma.c- In this code, there is a second #pragma will fork and print the “AFTER”, after the first #pragma forks and prints the “BEFORE”. The two different forks that print their “BEFORE”/”AFTER” statements.

MasterWorker.c (without #pragma)-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
arithmetic2_Kirby.s  fourth2_Kirby.o    second_Kirby      spmd2.c
barrier             fourth2_Kirby.s    second_Kirby.o    trap-notworking
barrier.c           fourth_Kirby       second_Kirby.s    trap-notworking.c
ControlStructure1   fourth_Kirby.o     second.s          trap-working
ControlStructure1.o fourth_Kirby.s      spmd2             trap-working.c

pi@raspberrypi:~/Downloads $ g++ masterWorker.c masterWorker -fopenmp
g++: error: masterWorker: No such file or directory
pi@raspberrypi:~/Downloads $ ls
arithmetic2_Kirby  ControlStructure1.s  fourth.s          spmd2_1
arithmetic2_Kirby.o  fourth2_Kirby       masterWorker.c    spmd2_1.c
arithmetic2_Kirby.s  fourth2_Kirby.o     second_Kirby      spmd2.c
barrier             fourth2_Kirby.s     second_Kirby.o    trap-notworking
barrier.c           fourth_Kirby        second_Kirby.s    trap-notworking.c
ControlStructure1   fourth_Kirby.o      second.s          trap-working
ControlStructure1.o fourth_Kirby.s       spmd2             trap-working.c

pi@raspberrypi:~/Downloads $ nano masterWorker.c
pi@raspberrypi:~/Downloads $ g++ masterWorker_Kirby.c -o masterWorker_Kirby -fopenmp
pi@raspberrypi:~/Downloads $ ./masterWorker_Kirby

Greetings from the master, # 0 of 1 threads

pi@raspberrypi:~/Downloads $

```

MasterWorker.c (without #pragma)- The masterWorker prints the statement “Greeting”, in the code where #pragma is commented, the statement is printed only once. As the thread id is 0, the if statement will print from “master”. Whereas the next code will have the #pragma code that will fork the threads with each printing “Greeting”

MasterWorker.c (with #pragma)-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
arithmetic2_Kirby.s  fourth2_Kirby.o  second_Kirby  spmd2.c
barrier             fourth2_Kirby.s  second_Kirby.o  trap-notworking
barrier.c           fourth_Kirby     second_Kirby.s  trap-notworking.c
ControlStructure1.o fourth_Kirby.o    second.s        trap-working
ControlStructure1.o fourth_Kirby.s    spmd2           trap-working.c
pi@raspberrypi:~/Downloads $ nano masterWorker.c
pi@raspberrypi:~/Downloads $ g++ masterWorker_Kirby.c -o masterWorker_Kirby -fopenmp
pi@raspberrypi:~/Downloads $ ./masterWorker_Kirby

Greetings from the master, # 0 of 1 threads

pi@raspberrypi:~/Downloads $ nano masterWorker_Kirby.c
pi@raspberrypi:~/Downloads $ g++ masterWorker_Kirby.c -o masterWorker_Kirby -fopenmp
pi@raspberrypi:~/Downloads $ ./masterWorker_Kirby

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

pi@raspberrypi:~/Downloads $

```

MasterWorker.c (with #pragma)- The masterWorker code with #pragma will fork the thread with 4 threads with each “Greetings” statement. The difference is the if statement will recognize the thread id of #0 being “master”, the other threads 1-3 will print their worker statement.

Kirby Liu

Task 4- ARM Assembly Programming

Fourth.s-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
    @if(x==0);
    @ y=1;
.section .data
x: .word 0
y: .word 0
.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 then part
    b endofif       @branch (jump) if false to the end of the IF statement bo
dy (branch always)
thenpart:
    mov r2,#1
    ldr r3,=y        @load the memory address of y into r3
    ldr r2,[r3]       @load r2 register value into y memory address
endofif:
    mov r7, #1       @Program Termination: exit syscall
    svc #0           @Program Termination: wake kernel
.end
(END)

```

Fourth.s- This code has two signed word variables, x and y, that are loaded into registers and then will compare register against a value, which won't store any values but will set eflags which with the jump function will jump if flags meet jump conditions to different parts of the code. In this case, the x address/value is stored in r1 and then compared against the value of 0. Then will jump if the r1 value is equal to zero (if Z==1), then jump to the thenpart otherwise jump to the endofif.

Fourth Kirby flag register-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
17      beq thenpart    @branch (jump) if true (Z==1) to the then part
(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      0x7efff380     0x7efff380
lr      0x0            0
pc      0x10080        0x10080 <_start+12>
cpsr    0x60000010     1610612752
fpscr   0x0            0
(gdb) next
thenpart () at fourth_Kirby.s:20
20      mov r2,#1
      _start:

```

Fourth Kirby flag register-Using the gdb debugger we put a break at the compare instruction of r1, variable x, to 0 so that we can see the cpsr, which the Program Status Register that contains the eflags, has a hex value of 60000010. This is where we will convert to binary


```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
@Fourth Program
@This program compute the following if statement construct:
    @intx;
    @inty;
    @if(x==0);
    @ y=1;
.section .data
x: .word 0
y: .word 0
.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 endofif     @branch (jump) if true (Z==0) to the then part
    mov r2,#1
    ldr r3,=y        @load the memory address of y into r3
    ldr r2,[r3]      @load r2 register value into y memory address
endofif:
    mov r7, #1      @Program Termination: exit syscall
fourth2_Kirby.s

```

Fourth2 Kirby- The Fourth2 asks the programmer to make a more efficient code. Whereas before it was branch if equal, we now branch if not equal. through the principles of Demorgan's Law, we can shorten it down to one jump function where if false jump to “endofif”. This a much cleaner code.

Fourth2_Kirby_bne-Screenshot

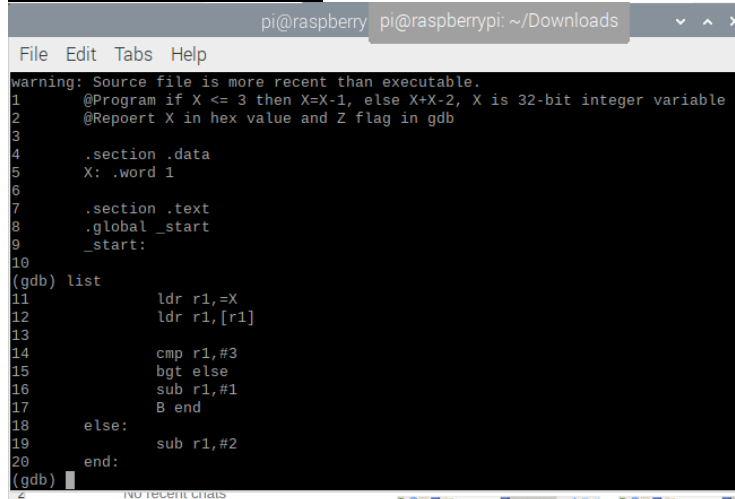
```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
17          bne endofif     @branch (jump) if true (Z==0) to the then part
(gdb) info register
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          0x7efff380     0x7efff380
lr          0x0            0
pc          0x100080       0x100080 <_start+12>
cpsr       0x60000010     1610612752
fpscr      0x0            0
(gdb) next
18          mov r2,#1
(gdb)

```

Fourth2 Kirby bne- Looking at the cpsr, we see that the eflags is still the same as the previous code when we use the compare function even though the branch instruction is different. This is important to note because the compare function was the same as the previous code with the same values.

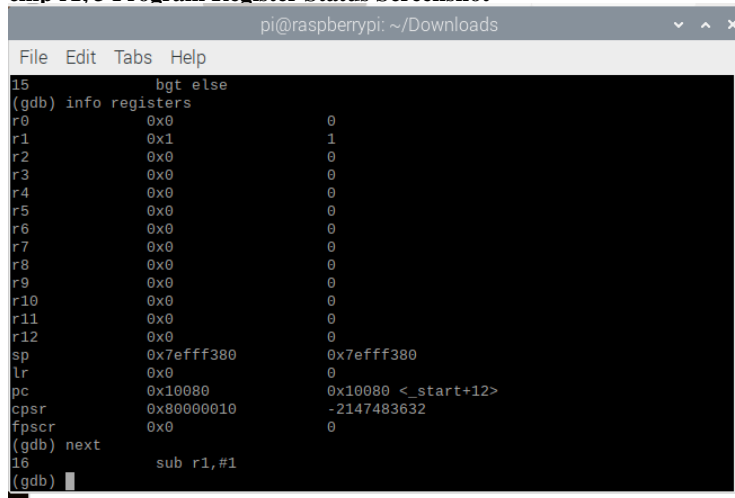
ControlStructure1-Screenshot



```
pi@raspberrypi: ~/Downloads
File Edit Tabs Help
warning: Source file is more recent than executable.
1  @Program if X <= 3 then X=X-1, else X=X+2, X is 32-bit integer variable
2  @Repoert X in hex value and Z flag in gdb
3
4  .section .data
5  X: .word 1
6
7  .section .text
8  .global _start
9  _start:
10
(gdb) list
11      ldr r1,=X
12      ldr r1,[r1]
13
14      cmp r1,#3
15      bgt else
16      sub r1,#1
17      B end
18  else:
19      sub r1,#2
20  end:
(gdb)
```

ControlStructure1- Here is a screenshot of the code for a “if, then, else” program with X=1 being compared to 3 and taking different steps depending the program status register. Using Demorgan’s Law, we inversed the original comparison of $X \leq 3$ to $X > 3$ while also inverting the order of the “jump to” code so that the original logic is preserved.

cmp r1, 3 Program Register Status Screenshot



```
pi@raspberrypi: ~/Downloads
File Edit Tabs Help
15      bgt else
(gdb) info registers
r0      0x0      0
r1      0x1      1
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      0x7efff380 0x7efff380
lr      0x0      0
pc      0x10080 0x10080 <_start+12>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
(gdb) next
16      sub r1,#1
(gdb)
```

cmp r1, 3- Registers hex value shows 80000010. It is the first digit that we are interested in which when convert to binary is 1000, and so the eflags are as follows Negative=1, Zero=0,

Carry=0, V (2's Complement Overflow)=0. As we can see the Negative flag is raised when we compare the r1 with 3, the indication is that the destination operand r1 is less than 3.

bgt else-Screenshot

```

pi@raspberrypi: ~/Downloads
File Edit Tabs Help
r0      0x0      0
r1      0x1      1
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      0x7efff380 0x7efff380
lr      0x0      0
pc      0x100080 0x100080 <_start+12>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
(gdb) next
16      sub r1,#1
(gdb) next
17      B end
(gdb)

```

bgt else- The bgt is “branch on greater than”, so if $bgt((Z \text{ or } (N \text{ xor } V)) == 0)$ is true we jump to else. Due to the Demorgan's Law, we are giving the inverse logical instruction where if $((Z \text{ or } (N \text{ xor } V)) != 0)$ we continue with the code to `sub r1,#1`. We skip over the else function by using the B instruction which is an unconditional jump to end.

Humaira Ridi

Task 3 A)- Parallel Programing Foundation

Race condition:

- What is race condition?
 - Refers to the behavior of a system whose output relies on the organization of uncontrollable events
- Why race condition is difficult to reproduce and debug?
 - Because the final result of a race condition cannot be predicted since they produce different behaviors for each run and relies on the timing between threads
- How can it be fixed? Provide an example from your Project_A3 (see `spmd2.c`)
 - To fix a race condition, each private variable must be assigned its own thread.
 - In `spmd2.c`, the problem was fixed after the variables had their own thread.

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).

- Patterns are used in parallel programs to make it easier for developers to write code. They can be categorized into two types: strategies and concurrent execution mechanisms. After

a programmer understands what algorithmic strategies to use, which involve making decisions about tasks that can run at the same time by more than one processing units. Executing at the same time, they must consider which implementation strategies to use. Concurrent execution patterns can be categorized into two subcategories: process/thread control patterns and coordination patterns. Process/thread control patterns determine the way processing units of parallel execution on hardware are organized. Coordination patterns have two different types: message passing, which occurs between concurrent processes and mutual exclusion, which occurs between concurrent threads.

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

- Collective synchronization (barrier) with Collective communication (reduction)
 - All threads are blocked until a point is reached during collective synchronization. During collective communication, all threads reach a certain point before execution. A single thread gathers data from other processes to conduct an operation.
- Master-worker with fork join.
 - A master thread is separated into smaller pieces which are distributed to worker threads during a master-worker pattern. In a fork-join pattern, the execution branches off in parallel at certain points before joining at a consecutive point to continue sequentially executing.

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

- Where can we find parallelism in programming?
 - In Program(task) View, Data View, and Resource View. In a program view, there is a statement level and a block/loop/routine/process level.
- What is dependency and what are its types (provide one example for each)?
 - Dependency refers to when an operation is reliant on a previous operation for the completion of a result before the operation can run.
 - True(Flow) dependence - S1: a=1; S2: b=1; second depends on first
 - Output dependence - S1: a=f(x); S2: a=b; second depends on first
 - Anti-dependence - S1: a=b; S2: b=1; first depends on second
- When a statement is dependent and when it is independent (Provide two examples)?
 - A statement is dependent when the execution order alters the outcome. Ex: The statements a=1; b=a+1; a=4; does not equal a=1; a=4; b=a+1;
 - A statement is independent when the order of the execution does not matter. Ex: The statements a=1; b=2 equal b=2; a=1;
- When can two statements be executed in parallel?
 - Only if there are no dependencies between the statements
- How can dependency be removed?
 - By rearranging or eliminating statements
- How do we compute dependency for the following two loops and what type/s of dependency?
 - We can compute dependency for the loops by comparing the IN and OUT sets of every node. All iterations are independent of each other. The loops classify as flow dependence because the statements are reliant on the variable in the for loop.

<pre>for (i=0; i<100; i++) S1: a[i] = i;</pre>	<pre>for (i=0; i<100; i++) { S1: a[i] = i; S2: b[i] = 2*i; }</pre>
---	---

Task 3 B)- Parallel Programming Basics

Trap-notworking:

Trap-notworking.c was programmed to compute the trapezoidal approximation of the integral of $\sin(x)$ from 0 to π using 2^{20} equal subdivisions by using a parallel loop pattern, as shown in the code below.

```
GNU nano 3.2      trap-notworking_Hum.c

double f(double x);

/*parse command-line arg for number of threads*/
if(argc>1){
    threadcnt = atoi(argv[1]);
}
#ifdef _OPENMP
    omp_set_num_threads(threadcnt);
    printf("OMP defined, threadcnt = %d\n", threadcnt);
#else
    printf("OMP not defined");
#endif

integral = (f(a)+f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared(a,n,h,integral)

GNU nano 3.2      trap-notworking_Hum.c

#else
printf("OMP not defined");
#endif

integral = (f(a)+f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared(a,n,h,integral)
for(i=1;i<n;i++){
    integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n",n);
printf("%f to %f is %f\n",a,b,integral);

double f(double x){
    return sin(x);
}
```

Result:

After running the program, this was the result:

```

pi@raspberrypi:~ $ g++ -o trap-notworking_Humaira trap-notworking_Humaira.c -lm
-fopenmp
pi@raspberrypi:~ $ ./trap-notworking_Humaira 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.450943
pi@raspberrypi:~ $

```

The line “`#pragma omp parallel for private(i) shared(a, n, h, integral)`” parallelizes the loop to compute the expression. This causes a problem because the variables have been declared so they can be either private or shared.

Trap-working:

Result:

After the code is altered to where the line “`#pragma omp parallel for private(i) shared(a, n, h, integral)`” is changed to “`#pragma omp parallel for \`” followed by “`private(i) shared (a, n, h) reduction(+: integral)`,” the correct result is displayed.

```

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

```

Barrier:

In barrier.c, all threads are programmed to run through a parallel section of code before the program can be executed.

```

GNU nano 3.2      barrier_Hum.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

```

Result:

When “`#pragma omp barrier`” is not commented out, this is the result:

(With pragma)

```

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

Thread 0 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 1 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.

```

When “#pragma omp barrier” is commented out and not included in the program, this is the result:

(Without pragma)

```

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

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

```

MasterWorker:

MasterWorker.c is an example of a program that uses a pattern with one thread that executes a block of code while forking and the rest of the other threads execute different blocks of code.

With pragma code:

```

GNU nano 3.2      masterWorker_Hum.c
}
#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);
    }
}

printf("\n");
[ Wrote 28 lines ]

```

Result with pragma:

This is the result of the program when “#pragma omp parallel” is not commented out and included in the code.

```

pi@raspberrypi:~$ gcc masterWorker_Hum.c -o masterWorker_Hum -fopenmp
pi@raspberrypi:~$ ./masterWorker_Hum 4
bash: ./: Is a directory
pi@raspberrypi:~$ ./masterWorker_Hum 4
Greetings from the master, # 0 of 1 threads

```

Result without pragma:

This is the result of the program when “#pragma omp parallel” is commented out.

```

pi@raspberrypi:~$ gcc masterWorker_Hum.c -o masterWorker_Hum -fopenmp
pi@raspberrypi:~$ ./masterWorker_Hum 4
Greetings from the master, # 0 of 4 threads
Greetings from a worker, # 1 of 4 threads
Greetings from a worker, # 2 of 4 threads
Greetings from a worker, # 3 of 4 threads

```

Task 4- ARM Assembly Programming

Part 1:

Fourth:

In this program, we are using checking to see if the variable x is equal to 0. If that statement is true, then the variable y is set to be equal to 1; this means 1 is stored inside r2, then the address of y is loaded into register r3, and then the value of r3 is loaded into r2.

```

GNU nano 3.2      fourth_Hum.s
.section .text
.globl _start
_start:
    ldr r1, =x
    ldr r1, [r1]

    cmp r1, #0
    beq thenpart
    b endifif
thenpart:
    mov r2, #1
    ldr r3, =y
    ldr r2, [r3]
endifif:
    mov r7, #1
    svc #0
.end
[ Wrote 26 lines ]

```

Fourth Result:

The ALU condition flags can be viewed by referring to the cpsr line in the screenshot below. When 60000010 is converted to binary from its hex state, we want to focus on the binary form for 6 since that is where the 4 ALU condition flags are located. 6h = 0110, therefore N = 0, Z = 1, C = 1, V = 0.

```

The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) info register
r0          0x0          0
r1          0x0          0
r2          0x0          0
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          0x7efff3c0   0x7efff3c0
lr          0x0          0
pc          0x10094      0x10094 <endifif>
cpsr        0x60000010   1610612752
fpscr       0x0          0
(gdb)

```

Part 2:

Fourth:

The code in Part 1 could be written more efficiently by replacing beq with bne and removing “b endifif.” This causes the program to jump to thenpart when the condition is false.

```

GNU nano 3.2      fourth_Hum.s
.section .text
.global _start
_start:
    ldr r1, =x
    ldr r1, [r1]

    cmp r1, #0
    bne thenpart
// b endifif
thenpart:
    mov r2, #1
    ldr r3, =y
    ldr r2, [r3]
endifif:
    mov r7, #1
    svc #0
.end
[ Wrote 26 lines ]

```

Fourth Result:

Again, we can focus on the cpsr line using the “info registers” command. When 60000010 is converted to binary from its hex state, we want to focus on the binary form for 6 since that is where the 4 ALU condition flags are located.

6h = 0110, so N = 0, Z = 1, C = 1, V = 0.


```

(gdb) step1
endofif () at fourth_Hum.s:24
24      mov r7, #1
(gdb) info registers
r0      0x0      0
r1      0x0      0
r2      0x0      0
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      0x7efff3c0 0x7efff3c0
lr      0x0      0
pc      0x10090   0x10090 <endofif>
cpsr    0x60000010 1610612752
fpscr    0x0      0
(gdb)

```

Part 3:

ControlStructure1:

In this program, we want to calculate the expression,

```

if X <= 3
    X = X - 1
else
    X = X - 2

```

I began by loading the memory address of variable x into register r0 and then loading the value into it. I used ble to see if x is less than or equal to 3. Since I assigned the value 1 to x, which is less than 3, it jumped to “then” where it subtracted one.

```

GNU nano 3.2      ControlStructure1.s
.section .data
x: .word 1
.section .text
.globl _start
_start:
    ldr r0, =x
    ldr r0, [r0]
    ldr r1, =x

    cmp r0, #3
    ble then
    b else

then:
    sub r1, #1
    str r1, [r1]
    b endofif

else:
    sub r1, #2

GNU nano 3.2
    cmp r0, #3
    ble then
    b else

then:
    sub r1, #1
    str r1, [r1]
    b endofif

else:
    sub r1, #2

endofif:
    mov r7, #1
    svc #0
    .end

```

Result:

The x value in the final result is 0, which means the Z flag is 1.

```

21      mov r7, #1
(gdb) info register
r0      0x1      1
r1      0x200ab  131243
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      0x100a0 0x100a0 <endofif>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
(gdb) stepi
22      svc #0
(gdb)

```

Dhruv Parikh

Task 3 A)- Parallel Programing Foundation

I. Race condition:

1. What is race condition?

A race condition is the behavior of an electronics, software, or other system where the output is dependent on the sequence or timing of the other uncontrollable events.

2. Why race condition is difficult to reproduce and debug?

Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger, often referred to as a "Heisenbug". It is therefore better to avoid race conditions by careful software design rather than attempting to fix them afterwards.

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

We have to assign each variable by its own thread.

II. 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).

There are two main categories of parallel patterns: Strategies and Concurrent Execution Mechanisms. In the examples in this document we introduce some well-used patterns for both algorithmic strategies and implementation strategies. Parallel algorithmic strategies primarily have to do with making choices about what tasks can be done concurrently by multiple processing units executing concurrently. Parallel programs often make use of several patterns of

implementation strategies. The concurrent execution patterns fall into two major categories: Process/Thread control patterns and Coordination patterns.

III. In the section “Categorizing Patterns” in the “Introduction to Parallel

Computing_3.pdf” compare the following:

o Collective synchronization (barrier) with Collective communication (reduction)

A barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

o Master-worker with fork join

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

1. Where can we find parallelism in programming?

- a. Program (task) view:
 - Statement level: Between program statements, which statement can be executed at the same time?
 - Block level/ Loop level/ Routine level/ Process level: Larger-grained program statements
- b. Data view:
 - How is data operated on?
 - Where does data reside?
- c. Resource view

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

A dependency arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.

Example: Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one processor:
 - Statement 1;
 - Statement 2;
- One two processors:

Processor 1: Processor 2:

Statement 1; Statement 2;

- Fundamental (concurrent) execution assumption

Processors execute independent of each other

No assumptions made about speed of processor execution

3. When a statement is dependent and when it is independent (Provide two

examples)?

If true, the statements are independent of each other. Two statements are dependent when the order of their execution affects the computation outcome

Example 1: Statements are independent

S1: a = 1;

S2: b = 1;

Example 2: Dependent(true(flow) dependence)

S1: a = 1; Second is dependent on first

S2: b = a;

4. When can two statements be executed in parallel?

Statements S1 and S2 can execute in parallel if and only if there are no dependences between S1 and S2

- True dependences
- Anti-dependences
- Output dependences

5. How can dependency be removed?

Some dependences can be removed by modifying the program

- Rearranging statements
- Eliminating statements

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

```
for (i=0; i<100; i++)      for (i=0; i<100; i++) {  
  S1: a[i] = i;             S1: a[i] = i;  
                           S2: b[i] = 2*i;  
                           }
```

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
 - IN(S): set of memory locations(variables) that may be used in S
 - OUT(S): set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

Commented [DP1]: I am done

Task 3 - Parallel Programming Basics

Trap-notworking.c

```
pi@raspberrypi:~ $ ./trap-notworkingDP 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.587982
```

1.587982 is not the right answer. The answer should be 2.000000. This occurred because the accumulators could not communicate properly. It needed the reduction clause to make the program work.

Trap-working.c

```
pi@raspberrypi:~ $ ./trap-workingDP 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
```

2.000000 is the correct answer. In this code we added the reduction clause because we have an accumulator variable in the loop called integral.

Barrier.c

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

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

This output is generated when the `#pragma omp barrier` is commented. This happens because there is not barrier between BEFORE and AFTER print statements.

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

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

This output is generated when the `#pragma omp barrier` is uncommented. The barrier pattern is used in parallel programs to ensure that all threads complete a parallel section of code before execution continues. This can be necessary when threads are generating computed data that needs to be completed for use in another computation.

MasterWorker.c

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

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

This is a program that shows how the master worker implementation strategy work. The pattern that programmers use in association with these patterns is to have one thread, called the master, execute one block of code when it forks while the rest of the threads, called workers, execute a different block of code when they fork.

Task 4- ARM Assembly Programming

Fourth.s

<pre>section .data x: .int 0 y: .int 0 section .text .globl _start _start: ldr r1, =x ldr r1, [r1] cmp r1, #0 beq thenpart b endofif thenpart: mov r2, #1 ldr r3, =y ldr r2, [r3] endofif: mov r7, #1</pre>	<pre> cmp r1, #0 beq thenpart b endofif thenpart: mov r2, #1 ldr r3, =y ldr r2, [r3] endofif: mov r7, #1 svc #0 .end</pre>
---	---

This set of code loads the memory address of x into r1. Then loads the value x into r1. Compares r1 value to 0. Jump if true to the thenpart. Jump if false to the end of IF statement body. If true load the memory address of y into r3.

```

12 thenpart: mov r2, #1
(gdb) stepi
13         ldr r3, =y
(gdb) info register
r0          0x0          0
r1          0x0          0
r2          0x1          1
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10088      0x10088 <thenpart+4>
cpsr       0x60000010    1610612752
fpscr       0x0          0
(gdb)

```

This is the info register after “if true” part. The register with “cpsr” is the register to know the flags. The flags are set in hexadecimal. So, we have to convert into binary. Hexadecimal 6 is 0110 binary.

N (Sign Flag) = 0

Z (Zero Flag) = 1

C (Carry Flag) = 1

V (Signed or Unsigned) = 0

```

15         ldr r3, =y
(gdb) stepi
16         ldr r2, [r3]
(gdb) info register
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10090      0x10090 <thenpart+8>
cpsr       0x60000010    1610612752
fpscr       0x0          0
(gdb)

```

This sets the y = 1. And the program executes the endif part.

Fourth2.s

```

.section .data
x: .int 0
y: .int 0
.section .text
.globl _start
_start:
    ldr r1, =x
    ldr r1, [r1]

    cmp r1, #0
    bne endofif
    thenpart: mov r2, #1
               ldr r3, =y
               ldr r2, [r3]

    endofif: mov r7, #1
              svc #0
              .end

```

In this program we replace beq with bneq (branch on not equal (Z == 0))

```

File Edit Tabs Help
10      cmp r1, #0
(gdb) stepi
11      bne endofif
(gdb) info register
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      0x7efff3c0 0x7efff3c0
lr      0x0      0
pc      0x10080 0x10080 <_start+12>
cpsr    0x60000010 1610612752
fpscr   0x0      0
(gdb)

```



```

File Edit Tabs Help
12 thenpart: mov r2, #1
(gdb) stepi
13      ldr r3, =y
(gdb) info register
r0          0x0          0
r1          0x0          0
r2          0x1          1
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          0x7efff3c0    0x7efff3c0
lr          0x0          0
pc          0x10088      0x10088 <thenpart+4>
cpsr       0x00000010    1610612752
fpscr       0x0          0
(gdb)

```

The program is false, because x not equal to 9, so the program did not jump over the thenpart.

ControlStructure1.s

```

section .data
x: .int 1
section .text
globl _start
_start:
    ldr r0, =x      body:
    ldr r1, =x      sub r0, #1
    ldr r0, [r0]    str r0, [r1]
                  b endofif

    cmp r0, #3
    bgt thenpart   thenpart:
                    sub r0, #2
                    str r0, [r1]
body:              sub r0, #1
                  str r0, [r1]
                  b endofif   endofif:
                              mov r7, #1
                              svc #0
thenpart:          .end

```

In this program we see if x is less than or equal to 3. If true, then x is equal to x minus 1. If false x equals to x minus 2.

The image shows two screenshots of a terminal window on a Raspberry Pi, displaying GDB debugging output. The window title is 'pi@raspberrypi: ~'. The top screenshot shows the execution of 'stepi' and 'info register' commands. The bottom screenshot shows the execution of 'stepi', 'b endofif', and 'info register' commands.

```
10      cmp r0, #3
(gdb) stepi
11      bgt thenpart
(gdb) info register
r0          0x1          1
r1          0x200a8      131240
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          0x7efff3a0    0x7efff3a0
lr          0x0          0
pc          0x10084      0x10084 <_start+16>
cpsr       0x80000010    -2147483632
fpscr      0x0          0
(gdb)

15      str r0, [r1]
(gdb) stepi
16      b endofif
(gdb) info register
r0          0x0          0
r1          0x200a8      131240
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          0x7efff3a0    0x7efff3a0
lr          0x0          0
pc          0x10090      0x10090 <body+8>
cpsr       0x80000010    -2147483632
fpscr      0x0          0
(gdb)
```

The value of x is 1, so the 1 is less then equal to 3. So, we execute x equals 1 minus 1. The answer is 0. So, the zero flag is 1.

Task 5

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

Slack: <http://team0101workspace.slack.com/>

Video (Youtube link): <https://youtu.be/5Wlm4mFk12I>