Developing Soft and Parallel Programming Skills Using Project-Based Learning

Spring-2019

Team HUSTLE

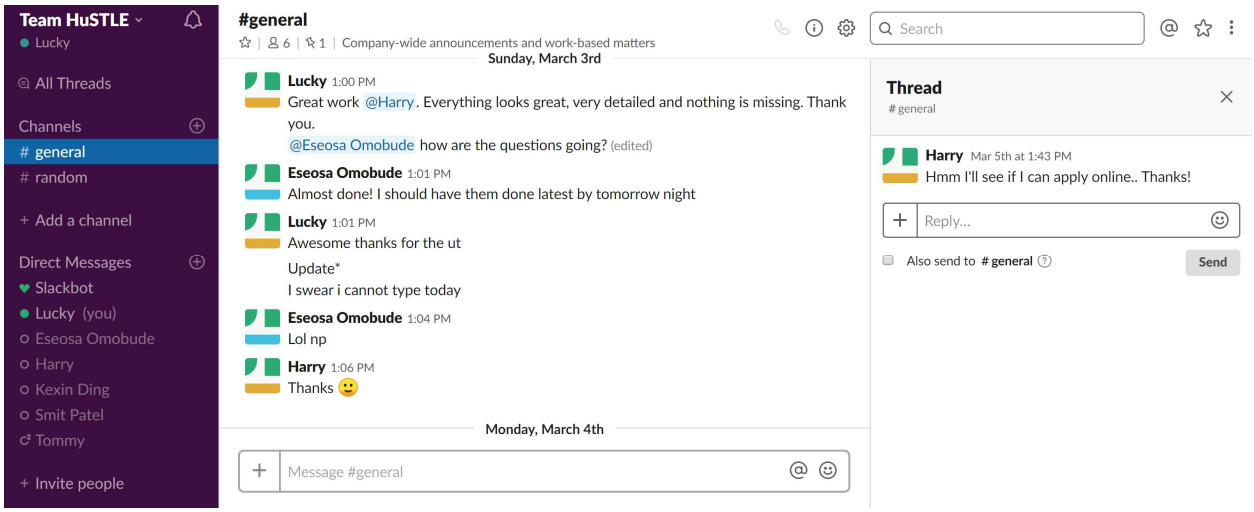
Lucky Phan, Eseosasere (Eseosa) Omobude, Hwysun (Harry) Park, Smit Patel, Tommy Lim

I. Planning and Scheduling

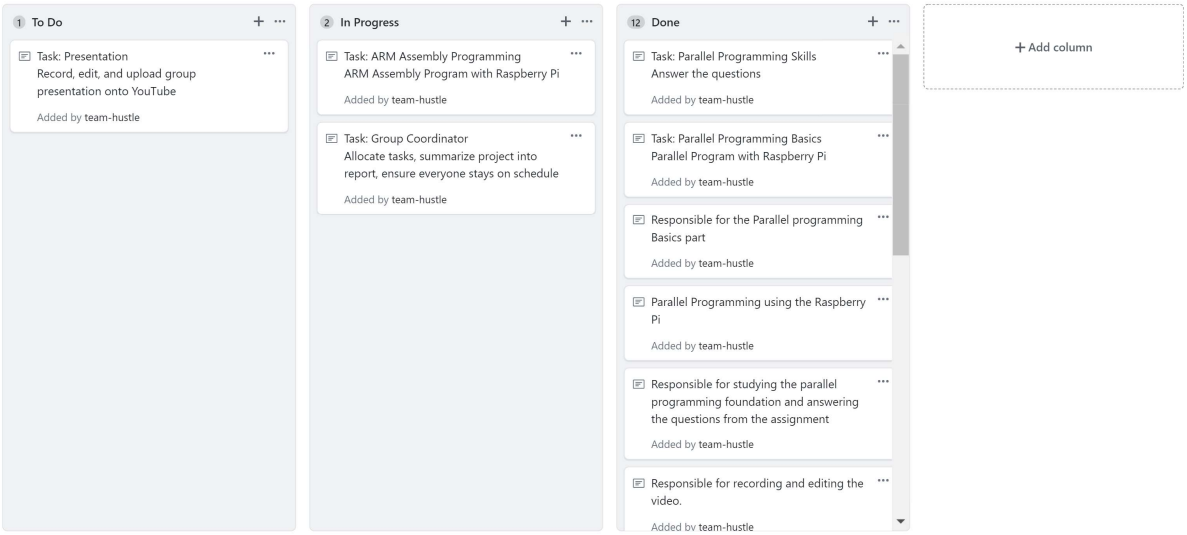
Name	E-mail	Tasks	Duration	Note
Lucky Phan (Coordinator)	lphan7@student.gsu.edu	Allocate tasks, summarize project into a report, ensure everyone stays on schedule, update GitHub	3 hours	Check on Slack, update GitHub, review report, Final Report due Fri, 03/09/19
Eseosasere Omobude	eomobude1@student.gsu.edu	Answer the questions for Parallel Programming Skills	3 hours	Parallel Programming Skills questions due Fri, 03/09/19
Smit Patel	spatel261@student.gsu.edu	ARM Assembly Programming and lab report	5 hours	ARM Programming Lab Report due Fri, 03/09/19
Tommy Lim	tlim3@student.gsu.edu	Record, edit, and upload group presentation onto YouTube	5 hours	Set recording date on Wed,03/06/19. Video due Fri, 03/09/19
Hwysun Park	hpark44@student.gsu.edu	Parallel Programming and lab report	4 hours	Parallel Programming to be done before Mon,03/04/19 Lab Report due Fri, 03/09/19

II. Collaboration:

Slack:



GitHub:



III. Parallel Programming Skills

Part A (Foundation):

Define the following:(in your own words)

Task- This is a basic unit of programming that the operating system controls. However, the task may be an entire program or the smaller portions of a program.

Pipelining- This is known as the continuous and kind of overlapped movement of instruction transferred to the processor. It is also present in the arithmetic/mathematical steps that are taken by the processor to perform an instruction.

Shared Memory Communications- memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs.

Synchronization- Refers to two related concepts. These concepts are the synchronization of processes (multiple processes connect at a point to settle an arrangement of action) and the synchronization of data (data consistency between multiple databases).

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

1. **Single-instruction, single-data (SISD) systems –**
   A SISD computing system is a uniprocessor machine which can execute a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed must be stored in primary memory.
2. **Single-instruction, multiple-data (SIMD) systems –**
   A SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets (N-sets for N PE systems) and each PE can process one data set.
3. **Multiple-instruction, single-data (MISD) systems –**
   An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset.

4. **Multiple-instruction, multiple-data (MIMD) systems** –
   An MIMD system is a multiprocessor machine which can execute multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore, machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.
   -

What are the Parallel Programming Models?

They are an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs. They are referred to as the bridging between hardware and software.

*Multi Programming Model
* Shared Address Space (Shared memory)
* Message Passing
-

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

-Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

- Multiprocessors
- Multicomputers

The evolution of parallel computers I spread along the following tracks −

- Multiple Processor Tracks
  - Multiprocessor track
  - Multicomputer track
- Multiple data track
  - Vector track
  - SIMD track
- Multiple threads track
  - Multithreaded track
  - Dataflow track

-OpenMP Higher level interface based on: compiler directives, library routines, runtime Emphasis on high-performance computing

* supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows.

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

Shared Memory Model

General Characteristics:

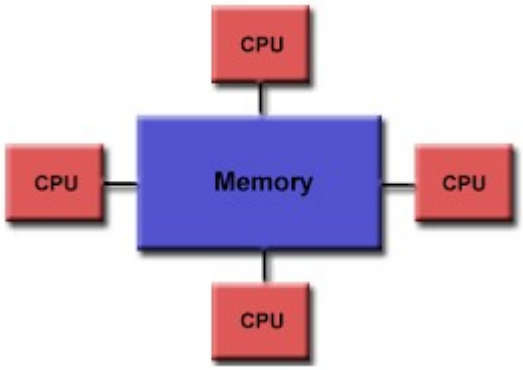
•The Shared memory parallel computers have many models, but usually have one thing in common. They can all allow all the processors to access every piece of memory as a "global address space".

•The multiple processors can all operate independently, however they can still share the same memory resources.

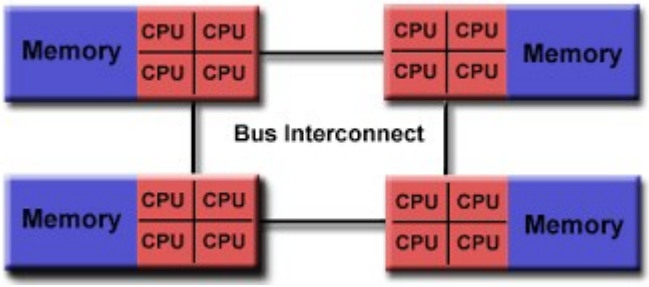
•If there are any changes in a memory location and effected by one processor, they are all visible to all the other processors.

•Shared memory machines have been classified as UMA and NUMA, based upon memory access times

(SHARED MEMORY (UMA))



(SHARED MEMORY (NUMA))



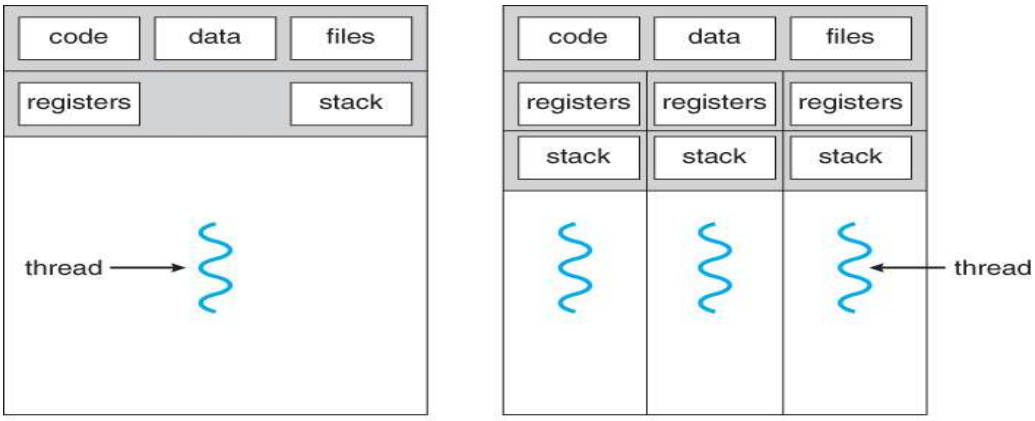
Threads model

- They exist within a process

-They can share address spaces

- Threads are lighter (faster creation, context switching, ...)

-Threads also communicate via shared variables



-

What is Parallel Programming? (in your own words)

It is the use of multiple resources and processors to solve a problem.

-

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

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

-

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

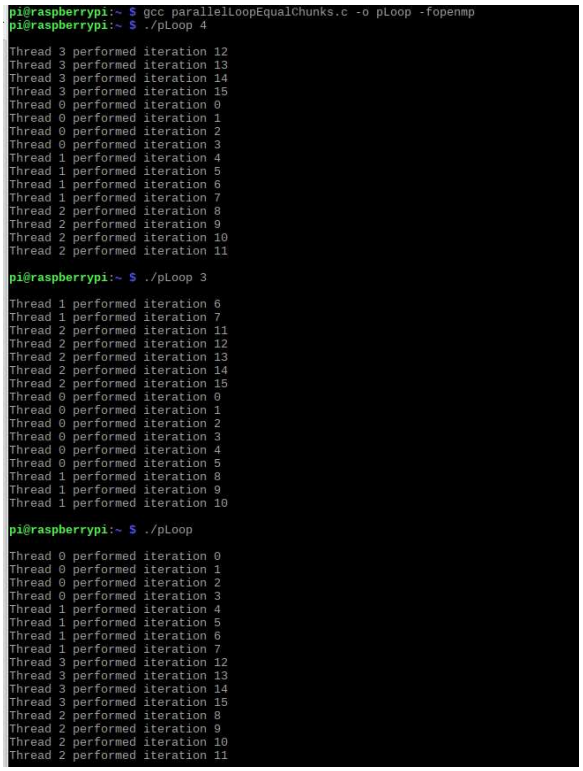
Along with a CPU, an SoC usually contains a GPU (a graphics processor), memory, USB controller, power management circuits, and wireless radios (WiFi, 3G, 4G LTE, and so on). Whereas a CPU cannot function without dozens of other chips, it's possible to build complete computers with just a single SoC.

Part B (Parallel Programming Basics):

2.1 Running Loops in Parallel

```
/* parallelLoopEqualChunks.c

... illustrates the use of OpenMP's default parallel for loop in which
threads iterate through equal sized chunks of the index range
(cache-beneficial when accessing adjacent memory locations).

*

Joel Adams, Calvin College, November 2009.

*

Usage: ./parallelLoopEqualChunks [numThreads]

*

Exercise
- Compile and run, comparing output to source code
- try with different numbers of threads, e.g.: 2, 3, 4, 6, 8

*/

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

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n")
;
if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
 }

#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}

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

- nano parallelLoopEqualChunks.c
  - Code copied and pasted in the file
  - Divides the work into consecutive iterations of the loop



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

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

pi@raspberrypi:~ $ ./pLoop 3

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

pi@raspberrypi:~ $ ./pLoop

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

- gcc parallelLoopEqualChunks.c -o pLoop -fopenmp

1) ./pLoop 4 (4 threads)
   - Thread 0 takes iterations 0-3 (4)
   - Thread 1 takes iterations 4-7 (4)
   - Thread 2 takes iterations 8-11 (4)
   - Thread 3 takes iterations 12-15 (4)

2) ./pLoop 3 (3 threads)
   - Thread 0 takes iterations 0-5 (6)
   - Thread 1 takes iterations 6-10 (5)
   - Thread 2 takes iterations 11-15 (5)

3) ./pLoop (the number of threads not specified)
   - 4 threads by default

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

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

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

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

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

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

- Changing the number of iterations

1) const int REPS = 15;
   - Thread 0 takes iterations 0-3 (4)
   - Thread 1 takes iterations 4-7 (4)
   - Thread 2 takes iterations 8-11 (4)
   - Thread 3 takes iterations 12-14 (3)

2) const int REPS = 14;
   - Thread 0 takes iterations 0-3 (4)
   - Thread 1 takes iterations 4-7 (4)
   - Thread 2 takes iterations 8-10 (3)
   - Thread 3 takes iterations 11-13 (3)

3) const int REPS = 13;
   - Thread 0 takes iterations 0-3 (4)
   - Thread 1 takes iterations 4-6 (3)
   - Thread 2 takes iterations 7-9 (3)
   - Thread 3 takes iterations 10-12 (3)

```c
/* parallelLoopChunksOf1.c
* ... illustrates how to make OpenMP map threads to
* parallel loop iterations in chunks of size 1
* (use when not accesssing memory).
*
* Joel Adams, Calvin College, November 2009.
*
* Usage: ./parallelLoopChunksOf1 [numThreads]
*
* Exercise:
* 1. Compile and run, comparing output to source code,
* and to the output of the 'equal chunks' version.
* 2. Uncomment the "commented out" code below,
* and verify that both loops produce the same output.
* The first loop is simpler but more restrictive;
* the second loop is more complex but less restrictive.
*/

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

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");

if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for schedule(static,1)
for (int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}

/*
printf("\n---\n\n");
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
for (int i = id; i < REPS; i += numThreads) {
printf("Thread %d performed iteration %d\n",
id, i);
}
}
*/

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

3.3 Another Way to Divide the Work

- nano parallelLoopChunksOf1.c

  - Code copied and pasted in the file
  - Gives one iteration of the loop to one thread, the next to the next thread, and so on.

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

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

pi@raspberrypi:~ $ ./pLoop2 3

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

pi@raspberrypi:~ $ ./pLoop2

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

```c
/* reduction.c
* ... illustrates the OpenMP parallel-for loop's reduction clause
*
* Joel Adams, Calvin College, November 2009.
*
* Usage: ./reduction
*
* Exercise:
* - Compile and run. Note that correct output is produced.
* - Uncomment #pragma in function parallelSum(),
* but leave its reduction clause commented out
* - Recompile and rerun. Note that correct output is NOT produced.
* - Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()
* - Recompile and rerun. Note that correct output is produced again.
*/

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

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

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

if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE) );

return 0;
}

/* fill array with random values */
void initialize(int* a, int n) {
        int i;
        for (i = 0; i < n; i++) {
                a[i] = rand() % 1000;
        }
}

/* sum the array sequentially */
int sequentialSum(int* a, int n) {
        int sum = 0;
        int i;
        for (i = 0; i < n; i++) {
```

- gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp

1) ./pLoop2 4 (4 threads)
   - Iterations 0-3 taken by threads 0-3, respectively
   - Iterations 4-7 by threads 0-3
   - Iterations 8-11 by threads 0-3
   - Iterations 12-15 by threads 0-3

2) ./pLoop2 3 (3 threads)
   - Iterations 0-2 taken by threads 0-2, respectively
   - Iterations 3-5 by threads 0-2
   - Iterations 6-8 by threads 0-2
   - Iterations 9-11 by threads 0-2
   - Iterations 12-14 by threads 0-2
   - Iteration 15 by thread 0

3) ./pLoop2 (the number of threads not specified)
   - 4 threads by default

4.1 When Loops Have Dependencies

- nano reduction.c

  - Code copied and pasted in the file
  - Takes an array of randomly assigned integers and calculate the sum of all values in the array in two ways: one in sequential computation and another in parallel with threads.

```
if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE) );

return 0;
}

/* fill array with random values */
void initialize(int* a, int n) {
        int i;
        for (i = 0; i < n; i++) {
                a[i] = rand() % 1000;
        }
}

/* sum the array sequentially */
int sequentialSum(int* a, int n) {
        int sum = 0;
        int i;
        for (i = 0; i < n; i++) {
                sum += a[i];
        }
return sum;
}

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

- nano reduction.c

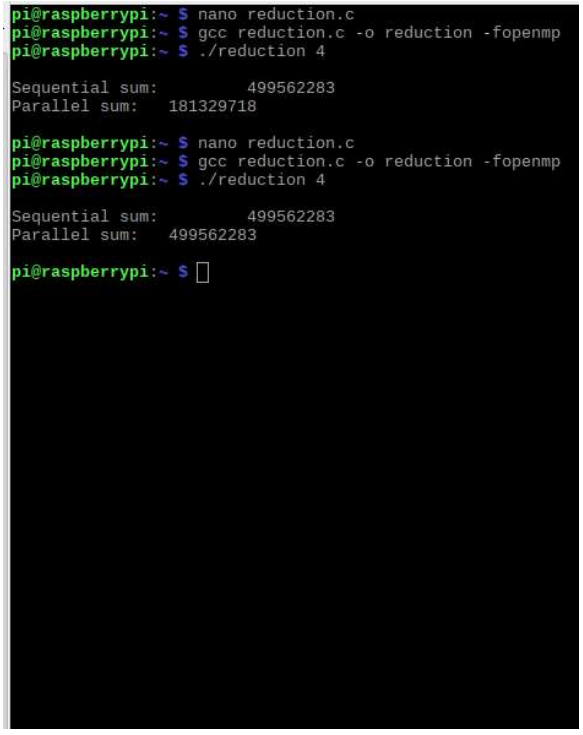
  - Note that "#pragma omp parallel for reduction (+:sum)" is in the comment.
  - To run the program in parallel, the comment indicators (//) need to be removed.
  - All values in the array are summed together by using the OpenMP parallel for pragma with the reduction(+:sum) clause on the variable sum.
  - The plus sign in the pragma reduction clause indicates the variable sum is being computed by adding values together in the loop.

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

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction 3

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $ ./reduction

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~ $
```

- gcc reduction.c -o reduction -fopenmp

1) ./reduction 4 (4 threads)
   ./reduction 3 (3 threads)
   - Sequential Sum: 499562283
   - Parallel Sum: 499562283

2) ./reduction (the number of threads not specified)
   - 4 threads by default



- Running parallelSum

1) Removed only the first //
   - Sequential sum result (499562283) does not match the Parallel sum result (181329718).

2) Removed the second //
   - The reduction clause is uncommented.
   - Sequential sum result (499562283) matches the Parallel sum result (499562283).

- Why doesn't the parallel for pragma without the reduction clause produce the correct result?
  - In this parallel for loop example, the reduction variable, or accumulator called sum, needs to be private to each thread as it does its work. The reduction clause in this case makes that happen. When each thread is finished, the final sum of their individual sums is computed.

3) Removed only the first //
   - Sequential sum result (499562283) does not match the Parallel sum result (181329718).

4) Removed the second //
   - The reduction clause is uncommented.
   - Sequential sum result (499562283) matches the Parallel sum result (499562283).

- Why doesn't the parallel for pragma without the reduction clause produce the correct result?
  - In this parallel for loop example, the reduction variable, or accumulator called sum, needs to be private to each thread as it does its work. The reduction clause in this case makes that happen. When each thread is finished, the final sum of their individual sums is computed.

IV. ARM Assembly Programming

Part 1

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

.section .text
.globl _start
_start:

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

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

-What error message did you get and why (report that)?

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

- The error in the program was the incorrect data type. The reason is that every data type that is declared, must turn yellow. Also, the code is trying to declare a sign halfword which is not possible in this way. We tried 'byte' and 'word' which seems to work and made us think about the mistake. In this case, it was shalfword and which was printed out to be normal. The correct answer to declare 16-bit is hword and short. After the error is fixed, all of the following commands worked fine and lead to the (gdb). The program below shows the corrected part.

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

.section .text
.globl _start
_start:

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

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

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from third...done.
(gdb) list
1                       @Third program
2       .section .data
3       a: .hword -2    @ 16bit-signed integer
4
5       .section .text
6       .globl _start
7       _start:
8
9       mov r0,#0x1             @ = 1
10      mov r1, #0xFFFFFFFF     @ =-1(signed)
(gdb)
11      mov r2, #0xFF           @ = 255
12      mov r3, #0x101          @ = 257
13      mov r4, #0x400          @ = 1024
14
15      mov r7,#1       @Program Termination: exit syscall
16      svc #0          @Program Termination: wake kernel
17      .end
18
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 7.
(gdb) run
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:10
10      mov r1, #0xFFFFFFFF     @ =-1(signed)
(gdb) stepi
11      mov r2, #0xFF           @ = 255
(gdb) x/1xh 0x20090
0x20090:        0xfffe
(gdb) x/1sh 0x20090
0x20090:        u"\xfffe\101\023\000\000"
(gdb) x/1xsh 0x20090
0x20090:        u"\xfffe\101\023\000\000"
(gdb)
```

- To view the memory address we used ‘p & ’ and to view the containing value in the specific memory, we used ‘x/nfs address.
- We observed that the first command shows the stored value inside the memory (0xfffe) but the sh command gives a long output with the same \xfffe\.

Part 2

```
(gdb) list
1       @arithmetic3 program teamHustle (Register = val2 + 3 + val3 - val1)
2       .section .data
3       val1: .byte -60 @ unisgned 8 bit
4       val2: .byte 11  @ unsigned 8 bit
5       val3: .byte 16  @ signed 8 bit
6
7       .section .text
8       .globl _start
9       _start:
10
(gdb)
11      ldr r1,=val1    @load the address of the mem val1 to r1
12      ldrsb r1,[r1]   @load the value of val1 to r1
13      ldr r2,=val2    @load the address of the mem val2 to r2
14      ldrb r2,[r2]    @load the value of val2 to r2
15      ldr r3,=val3    @load the address of val3 to r3
16      ldrsb r3,[r3]   @load the value of val3 to r3
17
18      add r2, r2,#3   @add 3 to r2 ; store into r2
19      sub r3, r3,r1   @sub r1 from r3 ; store into r3
20      add r0, r2,r3   @add r3 to r2 ; store into r0
(gdb)
21
22      mov r7,#1       @Program Termination: exit syscall
23      svc #0          @Program Termination: wake kernel
24      .end
```

- This is the same type of problem given in the previous assignment including memory to memory data movement.
- Since memory-memory data movement is not possible, we must move all the values between the registers to run this program.

```
@arithmetic3 program teamHustle (Register = val2 + 3 + val3 - val1)
.section .data
val1: .byte -60 @ unisgned 8 bit
val2: .byte 11  @ unsigned 8 bit
val3: .byte 16  @ signed 8 bit

.section .text
.globl _start
_start:

ldr r1,=val1    @load the address of the mem val1 to r1
ldrsb r1,[r1]   @load the value of val1 to r1
ldr r2,=val2    @load the address of the mem val2 to r2
ldrb r2,[r2]    @load the value of val2 to r2
ldr r3,=val3    @load the address of val3 to r3
ldrsb r3,[r3]   @load the value of val3 to r3

add r2, r2,#3   @add 3 to r2 ; store into r2
sub r3, r3,r1   @sub r1 from r3 ; store into r3
add r0, r2,r3   @add r3 to r2 ; store into r0

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

- First, we tried to figure out what the problem was asking for. Afterwards, we solved it as a normal equation, assigning values to each element.
- Register = val2+3+val3-val1; val1= -60, val2= 11, val3= 16
- Register = (11+3) +(16-(-60)) = 14+76 = 90
- The final value must be 90 stored into any register.

```
(gdb) p & val1
$1 = (<data variable, no debug info> *) 0x200ac
(gdb) p & val2
$2 = (<data variable, no debug info> *) 0x200ad
(gdb) p & val3
$3 = (<data variable, no debug info> *) 0x200ae
(gdb) x/1xb 0x200ac
0x200ac:        0xc4
(gdb) x/1xb 0x200ad
0x200ad:        0x0b
(gdb) x/1xb 0x200ae
0x200ae:        0x10
(gdb)
```

These are the values that are assigned to the memory.

- To verify both content in the registers as well as the memory we used ‘x/nfs address’. Also, to view the memory addresses we used ‘p & val1’ command inside the (gdb).
- Val1 memory address (8-bit) – 0x200ac including the value 60(c4h)
- Val 2 memory address (8-bit) – 0x200ad including the value 11(0bh)
- Val3 memory address (8-bit) – 0x200ae including the value 16(10h)

Before going through the program:

- Breakpoints were set to see what data is being moved to the registers and what values are getting changed as the program progresses.
- We were checking the values by using 'stepi' command to see each line running one by one and also 'info registers' to check the content of the registers.

```
11      ldr r1,=val1    @load the address of the mem val1 to r1
12      ldrsb r1,[r1]   @load the value of val1 to r1
13      ldr r2,=val2    @load the address of the mem val2 to r2
14      ldrb r2,[r2]    @load the value of val2 to r2
15      ldr r3,=val3    @load the address of val3 to r3
16      ldrsb r3,[r3]   @load the value of val3 to r3
```

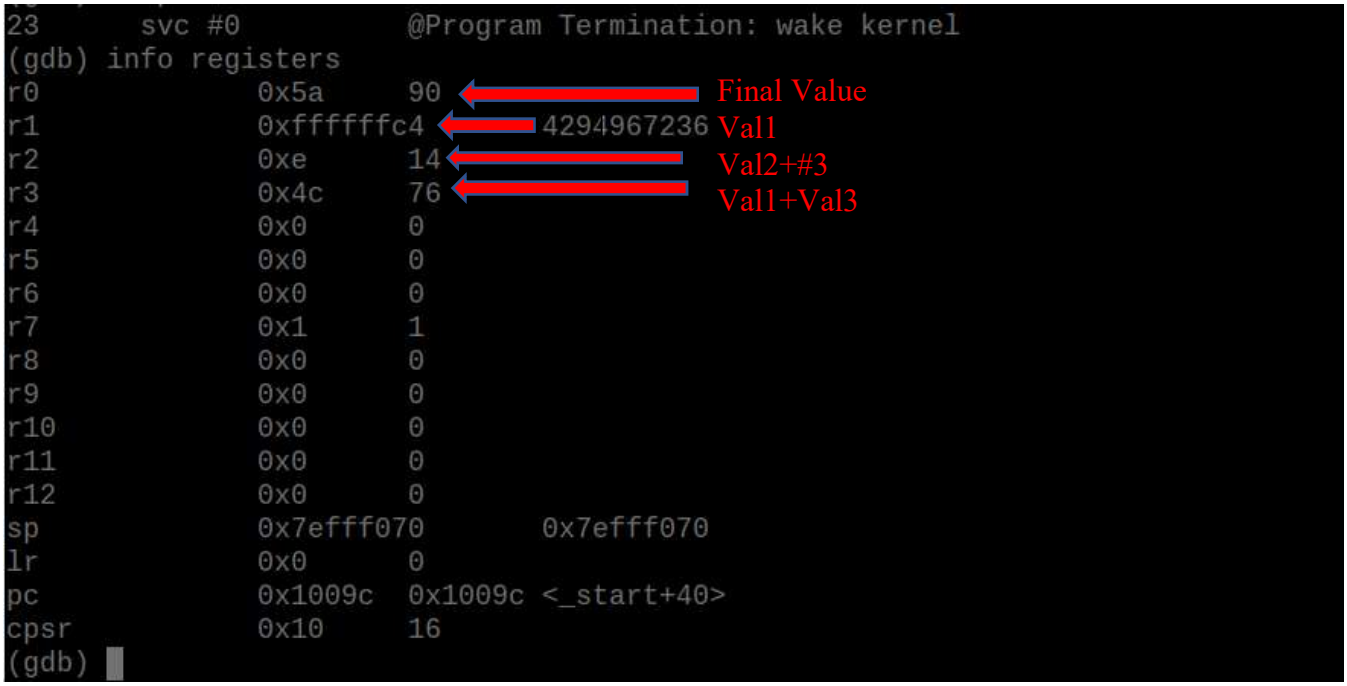
- Breakpoint 1 was created at line #12.
- Lines 11 to 16 were used to load the memory addresses as well as their values to the general-purpose registers (r1, r2, r3).

```
18      add r2, r2,#3   @add 3 to r2 ; store into r2
19      sub r3, r3,r1   @sub r1 from r3 ; store into r3
20      add r0, r2,r3   @add r3 to r2 ; store into r0
(gdb)
```

- Now, we are at the second part of the program which is the equation.
- Line #18 adds 3 to r2(11) and stores the output of 14 into the register 2.
- Line #19 subtracts (16 – (-60) =76) and stores 76 into register 3.
- Line #20 adds both 14 plus 76 and stores the output of 90 into register 0.

```
Breakpoint 2, _start () at arithmetic3.s:18
18      add r2, r2,#3   @add 3 to r2 ; store into r2
(gdb) info registers
r0             0x0      0
r1             0xfffffc4       4294967236
r2             0xb      11
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff070       0x7efff070
lr             0x0      0
pc             0x1008c  0x1008c <_start+24>
cpsr           0x10     16
(gdb)
```

- Breakpoint 2 was created at line #18.
- As we step through the program, the values will go accordingly to the values the equation called for.
- For instance, all of the values in the above screenshot has been loaded into their assigned register which will later change after the 'stepi' command.
- Also, we used the 'info register' in each line to see the data moving of the values.

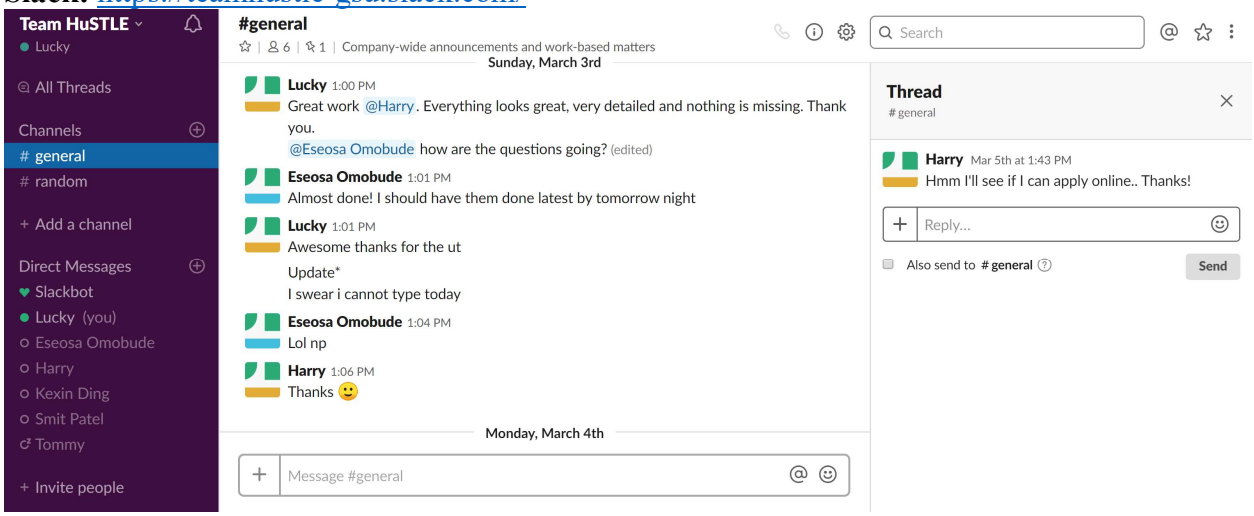


- This is the final part where all of the values are stored in the general-purpose registers.
- R0 is the register where the total value (solution) of the equation is stored which is 90 (5A in hex).
- R1 contains the signed value -60 (ffffffc4 in hex)
- R2 contains the first half of the equation value which is addition of the two values (val2+3 = 14 (E in hex))
- R3 contains the second half of the equation value which is val3-val1(16-(-60) = 76 (4C in hex))

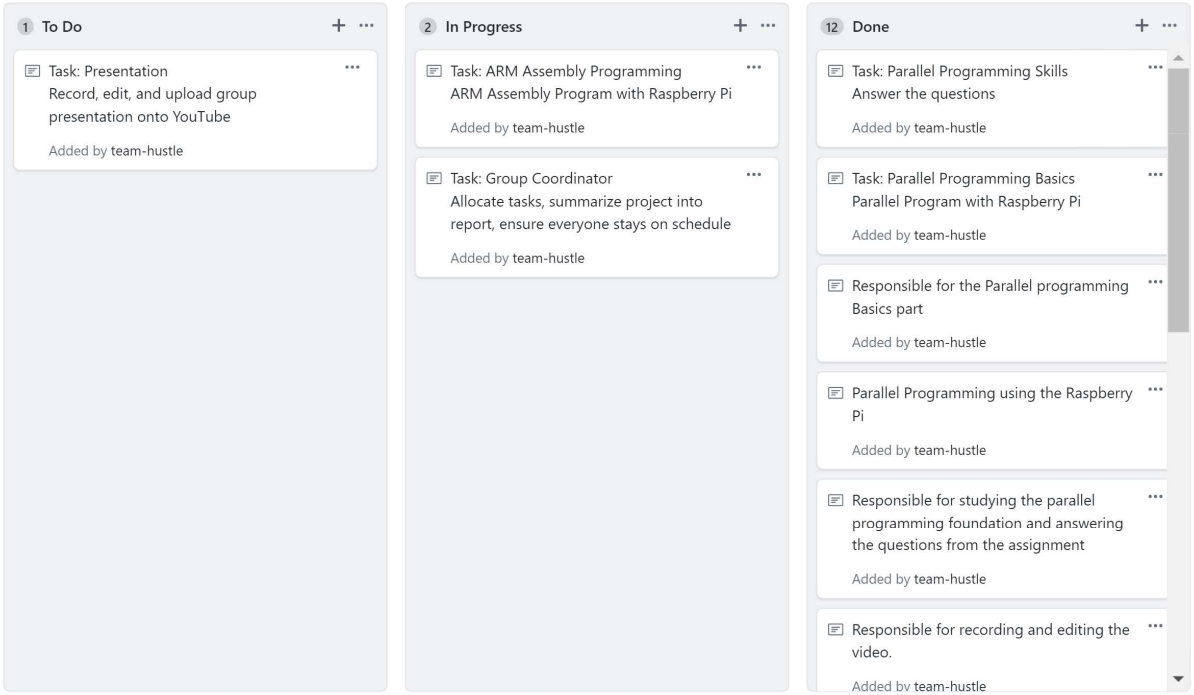
The interesting part in this assignment was to observe assigning signed values to the unsigned 8-bit. Another part was to figure out the specific data types for loading and storing signed and unsigned bytes.

V. Appendix

Slack: https://teamhustle-gsu.slack.com/



GitHub: https://github.com/team-hustle



YouTube: https://www.youtube.com/channel/UCfW_InzE5vaBrMkOPuKUd7g