

Developing Soft and Parallel Programming Skills Using Project-Based Learning

Fall 2019

Team CHABU

Chris Lavey, Harsh Jivani, Anggiela Yupanqui, Binh Ha Duy Le, Ugonma Nnakwe

1. Planning and Scheduling	2
2. Parallel Programming Skills	2
2.1. Foundations: Definitions	2
2.2. Foundations, contd.	3
2.2.1. Flynn's Taxonomy.	3
2.2.2. Parallel Programming Models	3
2.2.3. Parallel Memory Computer Architectures	3
2.2.4. Shared Memory vs. Threads	4
2.2.5. Parallel Programming	5
2.2.7. Benefits of SoC	5
2.3. Parallel Programming Basics	6
2.3.1. parallelLoopEqualChunks.c	6
2.3.2. parallelLoopChunksOf1.c	7
2.3.3. reduction.c	9
3. ARM Assembly Programming	11
3.1. Part 1	11
3.2. Part 2	12
4. Appendix	14
4.1. Github	14
4.2. Slack	15
4.3. Youtube (Channel)	15
4.4. Youtube (Video)	15

1. Planning and Scheduling

Work Breakdown

Assignee Name	Email	Task	Duration (Hours)	Dependency	Due Date	Note
Harsh Jivani	hjivani1@student.gsu.edu	Slack, ARM Assembly Programming, Parallel Programming (Foundation)	3 hours	None	10/20/19	
Chris Lavey	clavey@student.gsu.edu	Technical writing (getting the report ready) as described in the assignment	3 hours	Slack, Github, Parallel Programming, ARM Assembly Programming	10/24/19	
Anggiela Yupanqui	ayupanquirojas1@student.gsu.edu	GitHub, Parallel Programming	3 Hours	None	10/24/19	
Ugonma Nnakwe	unnakwe1@student.gsu.edu	Video editing, Parallel Programming	3 Hours	Video recording	10/24/19	
Binh Le (coordinator)	ble8@student.gsu.edu	ARM Assembly Programming, Parallel Programming (Foundation)	3 hours	None	10/20/19	

2. Parallel Programming Skills

2.1. Foundations: Definitions

2.1.1. Task:

A set of instructions that are run by the processor, or processors in the case of parallel programming.

2.1.2. Pipelining:

Breaking tasks down into individual steps that can be handled by different processing units. Increases number of instructions performed in a given time period, and allows the architecture to fetch instructions while the processor is executing arithmetic operations.

2.1.3. Shared Memory:

Memory can be accessed simultaneously by multiple processes regardless of the physical location of the memory.

2.1.4. Communications:

An exchange of data is required in order to do parallel computing and/or tasks. This can be done through many methods, including over a network or through a shared memory bus.

2.1.5 Synchronization:

The coordination of parallel tasks in real time. This results in a task having to wait on the others so they can be on the same standard of progress or logical equivalence. This will increase execution time.

2.2. Foundations, contd.

2.2.1. Flynn's Taxonomy.

Flynn's Classical Taxonomy	Description
SISD (Single Instruction, Single Data)	A single processing unit executes one instruction on one piece of data at a time.
SIMD (Single Instruction, Multiple Data)	Multiple processing units execute the same instruction on different pieces of data.
MISD (Multiple Instruction, Single Data)	Multiple processing units execute different instructions on a single piece of data.
MIMD (Multiple Instruction, Multiple Data)	Multiple processing units execute different instructions on different pieces of data.

2.2.2. Parallel Programming Models

2.2.2.1. Shared Memory

2.2.2.2. Threads

2.2.2.3. Distributed Memory/Message Passing

2.2.2.4. Data Parallel

2.2.2.5. Hybrid

2.2.2.6. Single Program Multiple Data (SPMD)

2.2.2.7. Multiple Program Multiple Data (MPMD)

2.2.3. Parallel Memory Computer Architectures

2.2.3.1. Uniform Memory Access (UMA):

All processors share the physical memory uniformly. It has equal access and access times to memory. If one processor updates a location in shared memory, then all the other processors will know about this update.

2.2.3.2. Non-Uniform Memory Access (NUMA):

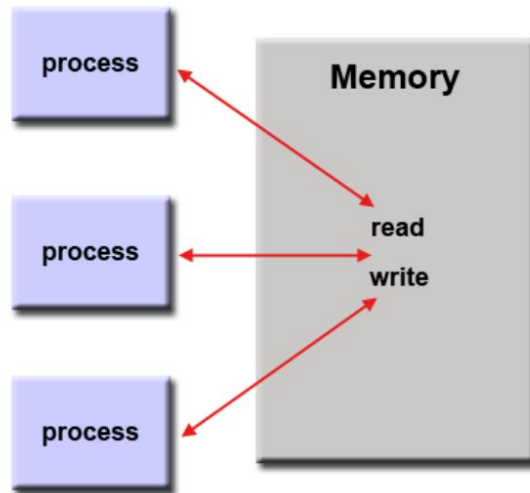
Memory access time is dependent on the location of the memory relative to the processor. Access time is not equal for all processors and memory access across a link is slower.

- OpenMP uses both Uniform Memory Access and Non-Uniform Memory Access. The reason is because OpenMP supports shared-memory multiprocessing programming which means UMA and NUMA both are capable of shared-memory within a socket or across the socket in a node.

2.2.4. Shared Memory vs. Threads

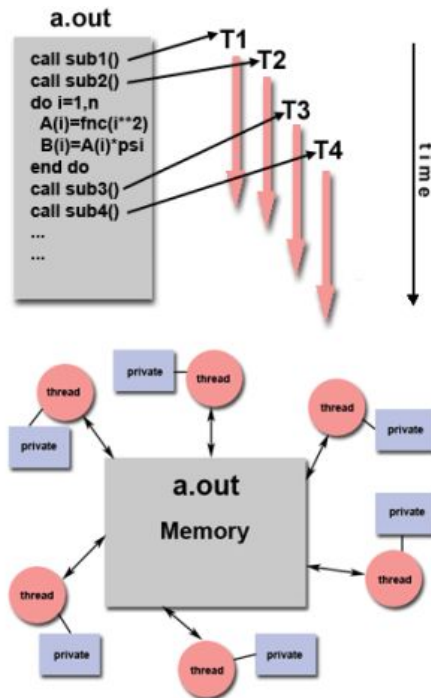
2.2.4.1. Shared Memory Model:

Tasks share a common address space where they can read and write asynchronously. Communication between tasks is implicit. If more than one task accesses the same variable, semaphores, or locks, can be used to control access to the shared memory. From a programmer's point of view, there is no need to explicitly specify the communication of data between tasks. This means that this simplifies program development.



2.2.4.2. Threads Model:

A task is broken up into smaller portions and handled by multiple threads. The main program loads and acquires all of the necessary systems and user resources, then it performs some serial work and creates many tasks (threads) that run concurrently. Each thread has some local data, and they communicate with each other through global memory. Threads can come and go, but there will always be a main thread to provide necessary shared resources until the application has been completed.



2.2.5. Parallel Programming

Parallel programming uses multiple processors to handle either separate parts of a task or multiple tasks. By having the task or tasks be broken up into smaller parts and being handled by multiple processors, the run time of the program can be reduced.

2.2.6. System on Chip (SoC)

SoC usually integrates all of the following components into a single silicon chip: a CPU, a GPU, memory, a USB controller, power management circuits, and wireless radios (Wi-Fi, 3G, 4G LTE, etc.). An example of SoC would be a Raspberry Pi, as a unit does not have RAM, a CPU, or a GPU separate to each other.

2.2.7. Benefits of SoC

2.2.7.1. Easy Integration:

Allows for these components to take up less space, resulting in more room for batteries. This is why this is usually used in smartphones and tablets.

2.2.7.2. Less Power Consumption:

SoC has a very high level of integration and requires much less wiring.

2.2.7.3. Cost

By reducing the number of components, SoC results in a lower cost overall.

2.3. Parallel Programming Basics

2.3.1. parallelLoopEqualChunks.c

Our first parallel programming task was to write **parallelLoopEqualChunks.c**. Below is the program:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.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
    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;
}
```

This program uses a number of threads set by the user to perform 16 iterations. We use the following command to compile our program:

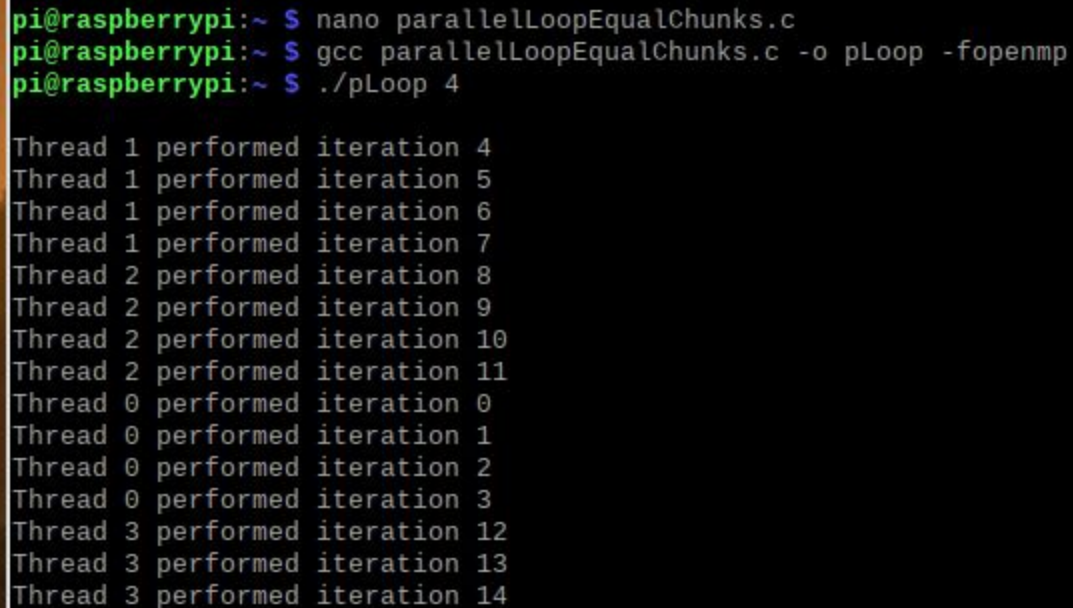
Gcc parallelLoopEqualChunks.c -o pLoop -fopenmp

To test our program, we use the command **./pLoop 4**. This uses 4 threads to perform iterations, with each thread handling an equal number of iterations. Below is the result:

```
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 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
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
```

When just running `./pLoop`, the program assumes that all threads are being used. By changing to number following `./pLoop`, we can change to number of threads performing iterations, and by using the `nano` terminal to change the number of REPS to 15, we can change the number of iterations in the loop, as seen in the screenshot below:



```
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 2 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 3 performed iteration 13
Thread 3 performed iteration 14
```

2.3.2. parallelLoopChunksOf1.c

For the second portion of this programming assignment, we had to write the program `parallelLoopChunksOf1.c`. Below is the program:


```

#include <stdio.h>
#include <stdlib.h>
#include <omp.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();
        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;
}

```

This program performs a similar role to the program in part 1. For this first description and screenshot, imagine that the first half is not commented out and the second half is. In this state, **parallelLoopChunksOf1.c** has the threads handle every fourth iteration (for example, thread 1 handles iterations 1,5,9, and 13) as opposed to **parallelLoopOfEqualChunks.c**, which handled iterations sequentially (for example, thread 1 handles iterations 0,1,2, and 3). This result can be seen once the program is compiled and run, as demonstrated in the screenshot below:

```

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

```

Now imagine the program is as it appears in the above screenshot with the first half commented out and the second half not. In this state, the program prints both the iterations and the threads out of order, but the threads still handle the same iterations as in the previous example. This can be seen in the below screenshot:

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

2.3.3. reduction.c

For the final portion of this programming assignment, we were tasked with writing the program **reduction.c**. Below is the program:

```

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

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;
}

void initialize(int*a, int n) {
    int i;
    for(i = 0; i < n; i++) {
        a[i] = rand()%1000;
    }
}

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

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;
}

```

This program calculates the sum of the values in an array both sequentially and using parallel. When we run the program as is, we find that the sum is found correctly. However, the program as it is right now does not use multiple processors to calculate the parallelSum. When we uncommented just **#pragma omp parallel for**, we received an incorrect value as our sum. However, when we uncommented **reduction(+:sum)** as well, we got the correct result. That is because the reduction clause is used to break up the task

of calculating the sum, then once the parts are finished the results are summed together. Without the reduction clause, the final results are not added together, and as such the final sum is incorrect. Below is a screenshot demonstrating the correct output:

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

Sequential sum:      499562283
Parallel sum:    499562283
```

3. ARM Assembly Programming

3.1. Part 1

In part 1 of the ARM assembly programming portion of this project, we were tasked with writing and correcting **third.s**. Below is the code:

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

.section .text
.globl _start
_start:

@ The following is a simple ARM code example that attempts to load a set of val$
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
```

third.s is a very simple program: it simply loads values into the registers. However, there are some errors in this program. This becomes apparent when we try to assemble the program, as the console returns **error: unknown pseudo-op: ‘.shalfword’**. In line 3, we declare variable **a** as a signed halfword, and this type is not being recognized by the assembler. This data type is illegal in ARM, and as such variables cannot be declared as this type. To fix this error, simply change **.shalfword** to **.hword**. This results in us using a halfword instead of a signed halfword.

After this, we are able to assemble, link, and compile or program without the console returning any errors. We then proceed to use the GNU Debugger to debug our program. We use **list** to list the first 10 lines of our program, and this shows that the area we’re interested in starts at line 7. We decide to set a breakpoint at line 7, then use **run** so that the program will run until the breakpoint. We use **stepi** until we have reached the line just before the program terminates, then we use **info registers** to display the values

stored in the registers at that point. Below are the values:

```
Breakpoint 1, _start () at third.s:11
11      mov r1,#0xFFFFFFFF      @ = -1 (signed)
(gdb) stepi
12      mov r2,#0xFF            @ = 255
(gdb) info registers
r0          0x1                1
r1          0xffffffff         4294967295
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
```

From this, we could see that our program loaded the register values as intended. We then proceeded to examine the memory at the address of the breakpoint using the following two commands:

```
x/1xh 0x1007c
```

```
x/1xsh 0x1007c
```

By using this, we see the 1 halfword at the address for the first command and some garbage for the second. This is because signed halfword is not a valid size for the command.

3.2. Part 2

In part 2 of the ARM assembly programming portion of this project, we were tasked with writing the program **arithmetic3.s** to calculate the following:

Register = val2 + 3 + val3 - val1,

where **val1** and **val2** are unsigned integers and **val3** is a signed integer.

Below is the code:

```

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

.section .text
.globl _start
_start:

ldr r2,=val2
ldrb r2,[r2]
add r2,r2,#3
ldr r3,=val3
ldrb r3,[r3]
add r2,r2,r3
ldr r1,=val1
ldrsb r1,[r1]
sub r2,r2,r1
strb r2,[r2]

```

Arithmetic3.s is a fairly straightforward program. In the **.data** section, we declare **val1**, **val2**, and **val3** to be of the **.byte** data type with their appropriate values. In the **.text** section, we load the address of **val2** into **r2** then use **ldrb r2, [r2]** to load the value of **val2** into **r2** as an unsigned byte. We then added 3 to **r2**, resulting in it storing a value of 14. We then loaded the address of **val3** into **r3** and used **ldrb** to load the value stored in **val3** into **r3** as an unsigned byte. We then add **r2** to **r3** and store the result into **r2**, resulting in **r2** having a value of 30. We then loaded the address of **val1** into **r1** and used **ldrsb** to load the value stored in **val1** into **r1** as a signed byte. We then proceeded to subtract **r1** from **r2** and store the result into **r2**, resulting in **r2** having a final value of 90. We then used **strb** to store the value of **r2** into the address of the register.

Once we had finished writing our program, we then proceeded to assemble and link our program. Since we didn't receive any errors, we then proceeded to debug our program using the GNU debugger. We used **list** until we saw the line number of the line we were interested in. We were interested in the last line before termination, **strb r2, [r2]**, and this was line 19. As such, we set a breakpoint at line 19 and proceeded to run the program in the debugger. We then used **info registers** to view register information at this point. Below is the result:

```
(gdb) info registers
r0          0x0          0
r1          0xffffffffc4 4294967236
r2          0x5a         90
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          0x7efff3b0   0x7efff3b0
lr          0x0          0
pc          0x10098      0x10098 <_start+36>
cpsr       0x10         16
fpscr      0x0          0
```

This showed that the values stored in the registers were as they should be. While **r1** may display as 4294967236 in this panel, this is because the system assumes the value is unsigned. If this hexadecimal value is considered signed, then it is actually -60, which is the value we stored in the register originally.

To find out if the value was stored properly at line 19, we used the following command:

x/1xb 0x10098

When converted from hexadecimal, this value was what we intended to store. As such, we knew we were done with this portion of the project.

4. Appendix

4.1. Github

4.1.1. Link:

<https://github.com/Team-Chabu/CSC3210-CHABU>

4.1.2. Screenshot:

Assignment 3
Updated 4 days ago

Filter cards

2 To do

- ☒ Video Editing: Due date: 10/24/19
Added by Team-Chabu
- ☒ Report- Due date: 10/24/19
Added by Team-Chabu

3 In progress

- ☒ Github maintenance.
Added by Team-Chabu
- ☒ parallel Programming Part 2- Due date: 10/20/19
Added by Team-Chabu
- ☒ Parallel programming part 1-Due date: 10/20/19
Added by Team-Chabu

1 Completed

- ☒ Arm assembly - Due Date:10/20/19
Added by Team-Chabu

4.2. Slack

https://join.slack.com/t/csc-3210-teamchabu/shared_invite/enQtNzc0ODEwMjkyNjU2LlTU4MGQ2MzcyMDg5YTJhMjc4ZmZjZWU4OTgzNjk4NzI1M2ZjMjY5YjBiMjYyMTgyOGVmNGQzMjFhMjYzN2YxOWM

4.3. Youtube (Channel)

<https://www.youtube.com/channel/UCp-0kVA08dU4F79t6slHtLQ>

4.4. Youtube (Video)

<https://youtu.be/hFSkL0W0L2A>