Matthew Maloof
Project 3 Task 3

Part A) Foundation

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

Task - A program or set of instructions that the processor executes. Multiple tasks are run on multiple processors which enables higher efficiency in parallel programs.

Pipelining - A type of parallel computing where a task is broken up into different steps which are assigned to different processor units.

Shared Memory - A type of computer architecture involving all processors having direct access to common physical memory. This is a model in which all parallel tasks can directly address and access the same logical memory without knowledge of where the physical memory is.

Communications - The actual event of data exchange that parallel tasks perform which is usually through a shared memory bus or over a network.

Synchronization - Associated with communications, synchronization involves the coordination of parallel tasks in real time. Using a synchronization point, the specified task will not be able to proceed past it until the other tasks reach the same point. This can increase a parallel application's wall clock execution time to increase because of how synchronization leads to waiting by one or more tasks.

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

Single Instruction, Multiple Data (SIMD) -

Processing units execute the same instruction altogether at any given clock cycle. These processing units operate on different data elements to provide higher efficiency. Two types - Processor Arrays and Vector Pipelines. An example of a use is graphics processor units in modern PCs which employ SIMD instructions.

Multiple Instruction, Single Data (MISD) -

The processing units operate independently on the data with separate streams. A single data stream is fed into multiple processing units which can be beneficial in a situation where multiple cryptography algorithms are attempting to crack a single coded message.

Multiple Instruction, Multiple Data (MIMD) -
This takes the benefit of every processing being able to execute different instruction streams and that every processor can work on different data streams. This allows for synchronous or asynchronous execution. This type of instruction is used in most modern supercomputers.

(7p) What are the Parallel Programming Models ? -
Shared Memory (without threads)
Threads
Distributed Memory / Message Passing
Data Parallel
Hybrid
Single Program Multiple Data (SPMD)
Multiple Program Multiple Data (MPMD)

(12p) List and briefly describe the types of Parallel Computer Memory Architecture. What type is used by OpenMP and why? -
Uniform Memory Access (UMA) -
Represented today by Symmetric Multiprocessor (SMP) machines, including identical processors with equal access and access times to memory. This type is also called Cache Coherent UMA (CC-UMA) which means that if one processor updates a location in shared memory, the other processors know about this update. This is done at the hardware level.
Non-Uniform Memory Access (NUMA) -
Made by linking two or more SMPS where one SMP has direct access memory of another SMP. These processors do not have equal access time to the memories which leads to slower memory access link. Maintaining cache-coherency allows for CC-NUMA. This is the type that OpenMP uses because it is designed for multi-processor / core, shared memory machines. It accomplishes parallelism exclusively through threads in explicit parallelism.

(10p) Compare Shared Memory Model with Threads Model ? (in your own words and show pictures) -
In a shared memory model, different processes or threads share a common address space. This includes different mechanisms like semaphores which are used to control access to this shared memory and to prevent deadlocks. This is considered the simplest parallel programming model where it has an advantage where there is no need to specify explicitly communication of data between tasks. A disadvantage includes it being more difficult to understand / manage data locality. This is used on different native operating systems, compilers, and stand-alone shared

memory machines. With distributed memory machines, the memory is physically distributed all across a network of machines.

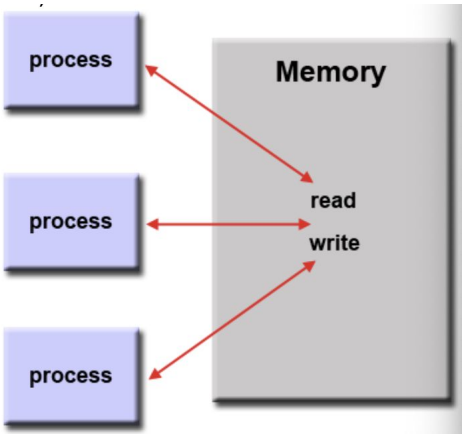

Figure 1: Shared Memory Model

In the threads model, a type of shared memory programming, of parallel programming, a single process can have multiple concurrent execution paths. This includes a main program being scheduled to run by the native OS, which then creates a number of tasks which are scheduled and run by the OS concurrently. Each of these threads include local data and share the resources of the main program. Threads implementations include a library of subroutines called within parallel source code and a set of compiler directives in either serial or parallel source code. Two different standardization efforts are used here, POSIX Threads and OpenMP. Other examples are Microsoft threads, Java/Python threads, and CUDA threads for GPUs.

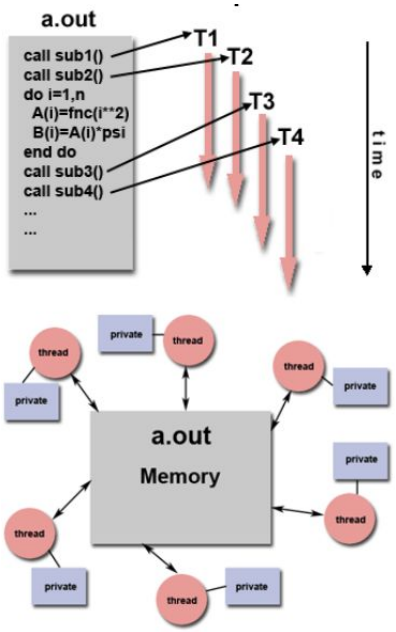

Figure 2: Threads Model

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

Parallel programming consists of the simultaneous use of multiple different resources to solve a computational program. This includes a problem being broken down into multiple smaller problems, which are broken down into instructions. These instructions are executed simultaneously on different processors to achieve efficiency.

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

The SoC integrates all 3 (CPU, RAM, GPU) units into a single chip. This is advantageous because a CPU cannot function without dozens of other chips but a single SoC can be used to build complete computers. The Raspberry Pi does use a SoC unit.

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

Advantages - SoC is only a bit larger than the CPU yet it contains the CPU, RAM, and GPU altogether. This allows for complete computers to be built in items such as smartphones and tablets taking little room. The SoC also uses considerably less power than the separate components. This allows for more power efficiency for mobile computing. It's also cheaper to build a SoC over building the CPU, GPU, and RAM separately. One disadvantage includes lack of flexibility because you can't exchange the CPU, GPU, and RAM for SoCs, but you can if they are in your computer separately by just taking them out and putting the new one in.

Part B) Parallel Programming Basics

```
  GNU nano 2.3.1                File: parallelLoopEqualChunks.c

/* 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).
*
Matthew Maloof, Georgia State University, March 2020.
*
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;
}
```

Figure 1: parallelLoopEqualChunks.c contents

First, I use nano parallelLoopEqualChunks.c to create the new file and copy and paste the code.

```
[mmaloof6@gsuad.gsu.edu@snowball ~]$ gcc -std=c99 parallelLoopEqualChunks.c -o pLoop -fopenmp
[mmaloof6@gsuad.gsu.edu@snowball ~]$ ./pLoop 4

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 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

[mmaloof6@gsuad.gsu.edu@snowball ~]$
```

Figure 2: gcc command

Next, we use the gcc command to turn the file into an executable program. Then we type ./pLoop 4 to execute the program with 4 threads and the output is shown above.

```
[mmaloof6@gsuad.gsu.edu@snowball ~]$ ./pLoop 5

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

[mmaloof6@gsuad.gsu.edu@snowball ~]$
```

Figure 3: pLoop 5 (non divisible by # threads)

When we use a number not divisible by # threads, a 5th thread is made, thread 4 shown above.

```
/* parallelLoopChunksOf1.c
* ... illustrates how to make OpenMP map threads to
* parallel loop iterations in chunks of size 1
* (use when not accesssing memory).
*
* Matthew Maloof, Georgia State University, March 2020.
*
* 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",
  a. id, i);
}
}
*/
printf("\n");
return 0;

^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```

Figure 4: parallelLoopChunksOf1.c file contents

Above is the new file we created using nano parallelLoopChunksOf1.c and copy/pasted code.

```
[mmaloof6@gsuad.gsu.edu@snowball ~]$ gcc -std=c99 parallelLoopChunksOf1.c -o pLoop2 -fopenmp
[mmaloof6@gsuad.gsu.edu@snowball ~]$ ./pLoop2 4

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
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 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 13

[mmaloof6@gsuad.gsu.edu@snowball ~]$
```

Figure 5: gcc command and ./pLoop2 4

We turn the file into an executable then run it with a thread count of 4.

```
[mmaloof6@gsuad.gsu.edu@snowball ~]$ ./pLoop2 5

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

[mmaloof6@gsuad.gsu.edu@snowball ~]$
```

Figure 6: ./pLoop2 5 command

We run this to see how different thread counts affect the program. We can see that a 4th thread is created and used.

```
[mmaloof6@gsuad.gsu.edu@snowball ~]$ nano parallelLoopChunksOf1.c
[mmaloof6@gsuad.gsu.edu@snowball ~]$ ./pLoop2 4

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 0 performed iteration 0
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 3 performed iteration 15
Thread 1 performed iteration 13

[mmaloof6@gsuad.gsu.edu@snowball ~]$
```

Figure 7: Uncommenting out the block of code and running it

We can see that thread 0 gets the first iteration, and thread 1 gets iteration 1 etc. then it goes back to thread 0 etc. Same pattern is shown using different thread counts.

```
/* reduction.c
* ... illustrates the OpenMP parallel-for loop's reduction clause
*
* Matthew Maloof, Georgia State University, March 2020.
*
* 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++) {
 sum += a[i];

^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```

```
 }
 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;
 }
```

Figure 8: reduction.c program contents

We create the new file reduction.c using nano command and copy/paste the code.

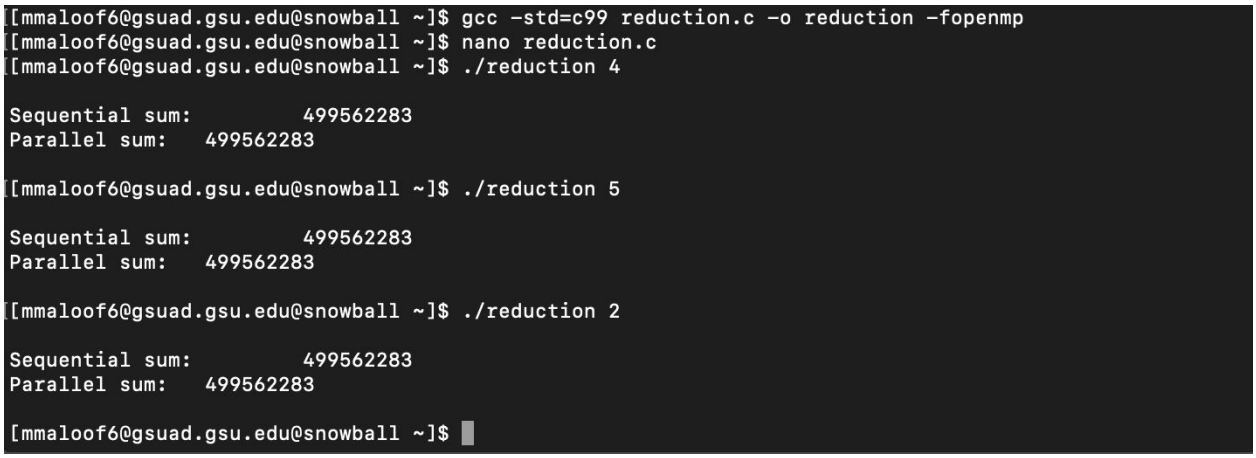

Figure 9: gcc command and ./reduction command with thread count.

We turn reduction.c into an executable using gcc and we run ./reduction 4 (and 5 and 2) which outputs above. This output is shown using the varying thread counts. This is the expected output because they are running the same code. After uncommenting line 39 the sequential sum is correct but parallel sum is incorrect. We remove the second comment in line 39 which is the reduction cause and now the variable sum is being computed by adding values together in a loop and everything is running as it should correctly. The reduction clause makes the variables private to each thread.