

Assignment 2 Task 3 Team x87

Matthew Maloof -

a)

- Single board computer, Quad-core Multicore CPU, 1GB Ram, I/O (display, power, hdmi, camera, ethernet, USB), and power button.
- There are 4 cores in a quad-core CPU.
- The first difference between Intel x86 and ARM computer implementations is the instruction set. Intel uses CISC (Complex Instruction Set Computing) processor which allows complex instructions to access memory. This enables it to have more operations, addressing modes, but less registers than ARM. ARM computer implementation uses RISC (Reduced instruction set computing) processor which uses a simplified instruction set with more general purpose registers compared to Intel x86. ARM's instructions operate on registers and can access the memory using Load/Store instructions. This reduced instruction set allows for faster instruction execution time, and greater speed. ARM instructions also can be used for conditional execution. Intel processors use little-endian format whereas ARM processors became BI-endian after version 3.
- Sequential (Serial) computation is where a problem is broken down into a discrete series of instructions. These instructions are executed sequentially and by a single processor. Because of this, only one instruction can be executed at any moment in time. Parallel computing is when you use multiple compute resources at the same time to solve a computational problem. The problem is broken down into separate discrete parts that are solved concurrently. These parts are then broken down into smaller series of instructions that execute from different parts at the same time. Parallel computing includes an overall control mechanism. Computers used today are parallel from a hardware perspective. This allows for multiple functional units, execution units, and hardware threads. Sequential computation ignited the start of computers and where they are now. This includes the oldest type of computer with older generation mainframes, minicomputers, workstations, and single processor/core PCs.
- Data parallelism is a category of parallelism where computation is being run for multiple data items. The amount of available parallelism is proportional to the input size, where the amount of potential parallelism is very large. Task parallelism applies to solutions where parallelism is organized with the functions rather than around the data. The emphasis is on functional decomposition, which is used with either tasks or threads.
- A process is the abstraction of a running program which does not share memory with each other. Single-core CPUs can only operate one process at a time. A thread is a lightweight process which allows for a single process to be decomposed into smaller, independent parts. These threads share common memory of the process they belong to. An example is an OS that will schedule threads on separate cores/CPU's when they are available.
- OpenMP is a library/language or API that is commonly used among developers. OpenMP pragmas is a type of compiler directive that enables the compiler to generate threaded code. OpenMP uses an implicit multithreading model where the library handles the thread creation

and management. This allows for much simpler tasks for the programmers, and as such it is the standard the compilers that implement it adhere to.

- Database servers, Web servers, Compilers, and Multimedia applications all benefit from multi-core.
- A single-core CPU operates one process at a time, and multi-core allows for more than one process to be run at a time. Because of this, multicore increases the throughput of the system. Multicore also allows for threads where it allows for faster speed/runtime. An example is how the OS will schedule threads on separate cores where if it was single-core it would have to do one at a time. Multicore runs on lower frequency compared to the single-core which allows for a reduction in power dissipation.

b) Parallel programming

- To start part b, we needed to put the code from the slide into a new file using nano spmd2.c. After typing the code you can press control+x, press y, then press enter to save the file and exit the editor.

```
File Edit Tabs Help
GNU nano 3.2 spmd2.c

/*spmd2.c
 *... illustrates the SPMD pattern in OpenMP,
 *using the commandline arguments
 *to control the number of threads.
 *
 *Matthew Maloof, Georgia State University, February 2020
 */
*Exercise:
 *-Compile & run with no commandline args
 *-Rerun with different commandline args
 */
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv) {
int id, numThreads;
printf("\n");
if (argc>1){
[ Read 29 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spcrot
```

```
File Edit Tabs Help
GNU nano 3.2 spmd2.c

*Exercise:
 *-Compile & run with no commandline args
 *-Rerun with different commandline args
 */
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv) {
int id, numThreads;
printf("\n");
if (argc>1){
omp_set_num_threads(atoi(argv[1]));
}
#pragma omp parallel
{
id=omp_get_thread_num();
numThreads=omp_get_num_threads();
printf("Hello from thread %d of %d\n", id, numThreads);
}
}
```

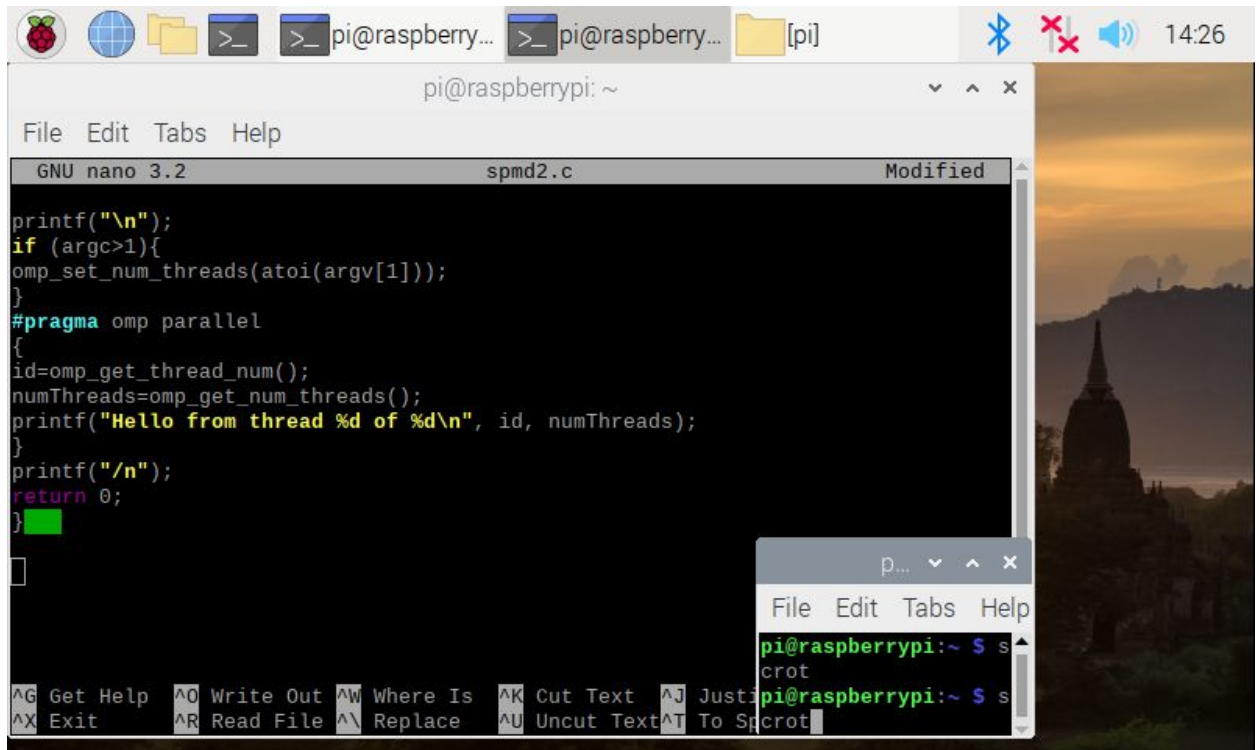
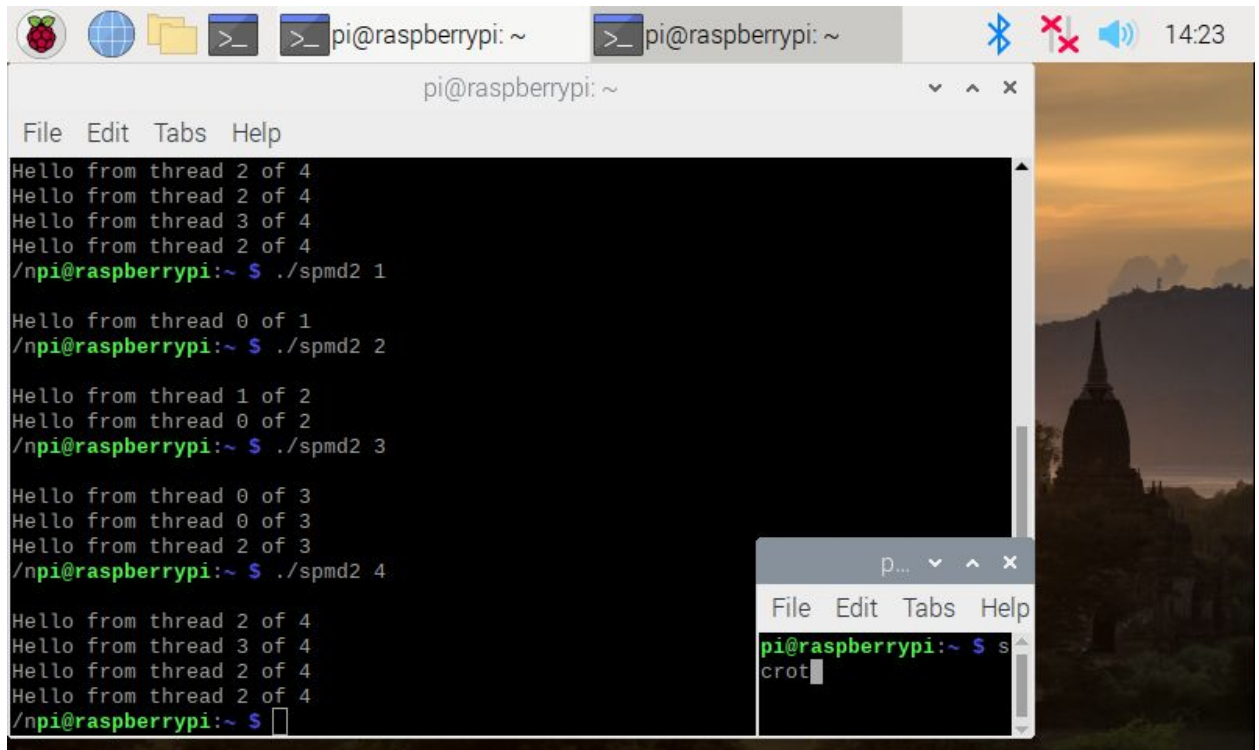


Figure 1-3: spmd2.c file creation

To compile the code and allow it to be executed, we can type `gcc spmd2.c -o spmd2 -fopenmp` then `./spmd2 (threadcount)` to see the results.

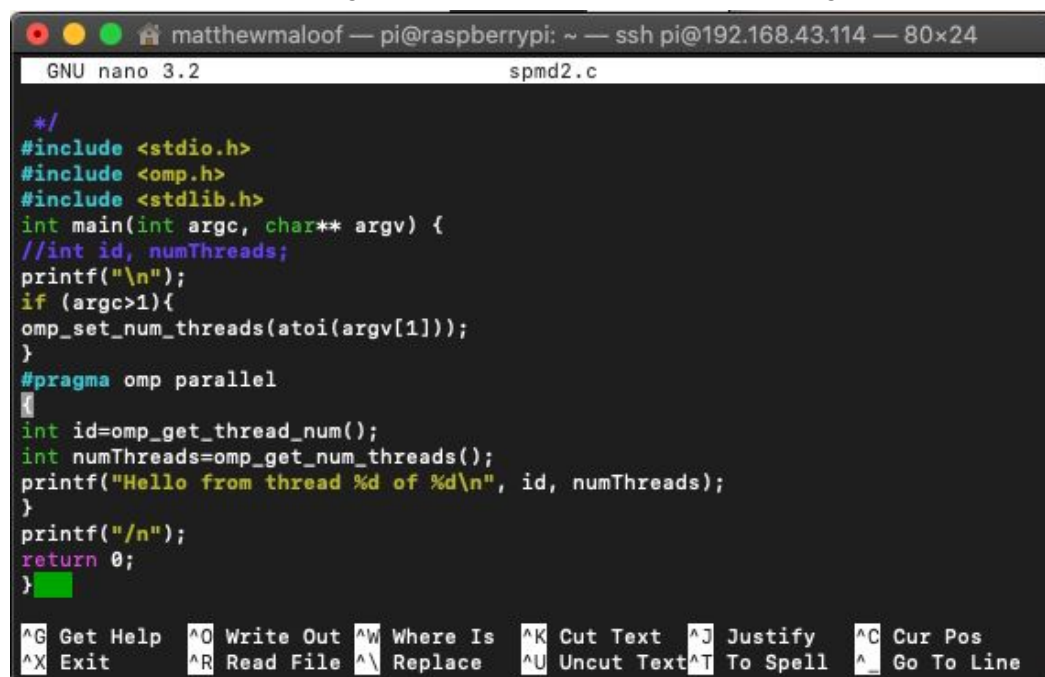


```
pi@raspberrypi: ~
File Edit Tabs Help
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
/pi@raspberrypi:~ $ ./spmd2 1
Hello from thread 0 of 1
/pi@raspberrypi:~ $ ./spmd2 2
Hello from thread 1 of 2
Hello from thread 0 of 2
/pi@raspberrypi:~ $ ./spmd2 3
Hello from thread 0 of 3
Hello from thread 0 of 3
Hello from thread 2 of 3
/pi@raspberrypi:~ $ ./spmd2 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
/pi@raspberrypi:~ $
pi@raspberrypi:~ $ s
crot
```

Figure 4: Thread count for first program

In figure 4, you can see that the thread count output is incorrect for this program because a thread id should not appear more than once, they should all be unique.

To fix this, we can go back into nano spmd2.c and change the code to what's in figure 5.

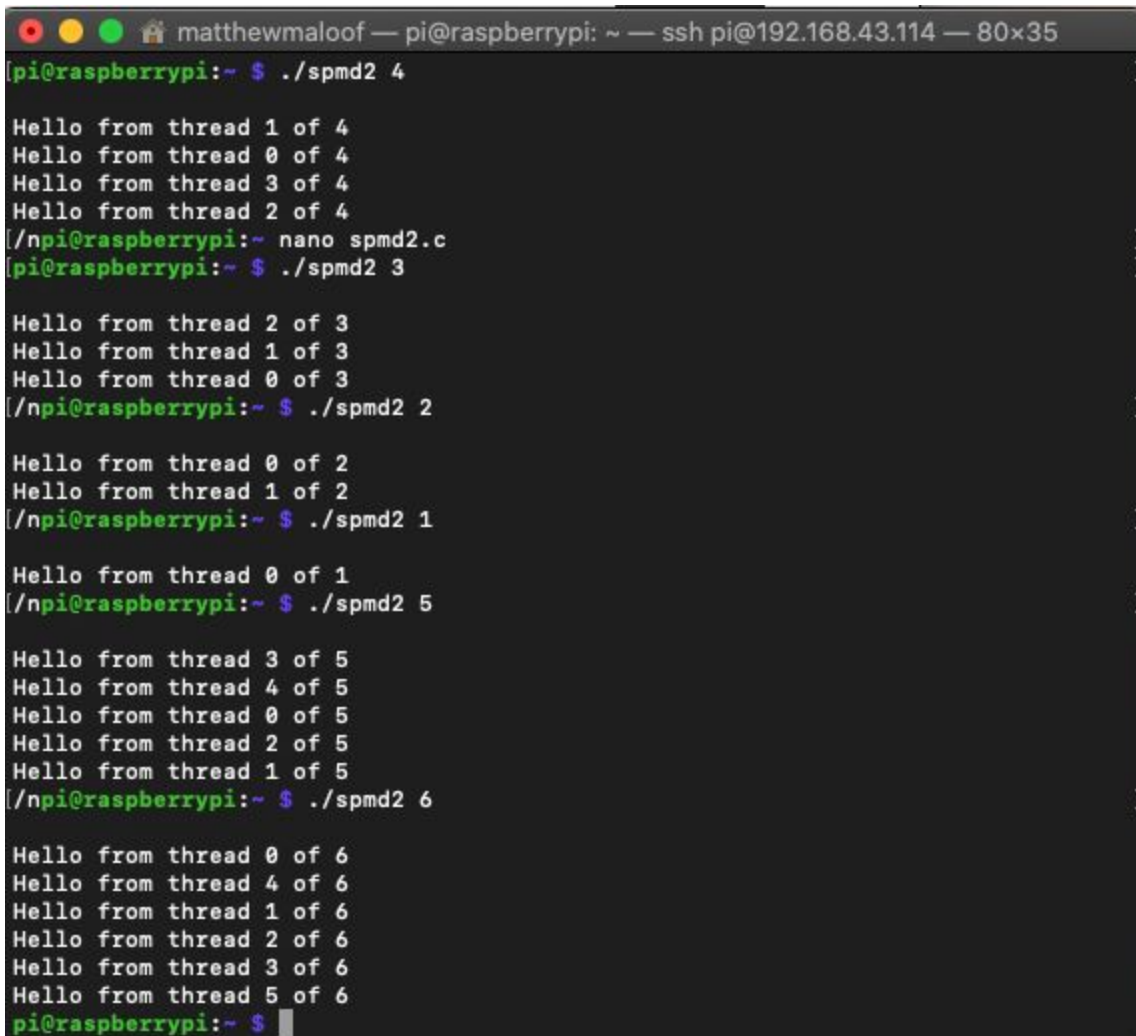


```
matthewmalooof — pi@raspberrypi: ~ — ssh pi@192.168.43.114 — 80x24
GNU nano 3.2      spmd2.c

*/
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char** argv) {
//int id, numThreads;
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("Hello from thread %d of %d\n", id, numThreads);
}
printf("\n");
return 0;
}
```

Figure 5: Corrected Program

To correct the program, we commented out the `int id, numThreads` line and added `int` to the beginning of the `id=omp_get_thread_num()` and `numThreads=omp_get_num_threads()` lines. These changes allow each thread to have their own private copy of the variables named `id` and `numThreads`. We can type `gcc spmd2.c -o spmd2 -fopenmp` and then `./spmd2 (thread count)` to see if the code fixed the issue.



```
matthewmaloof — pi@raspberrypi: ~ — ssh pi@192.168.43.114 — 80x35
pi@raspberrypi:~ $ ./spmd2 4
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
[/npi@raspberrypi:~ nano spmd2.c
pi@raspberrypi:~ $ ./spmd2 3
Hello from thread 2 of 3
Hello from thread 1 of 3
Hello from thread 0 of 3
[/npi@raspberrypi:~ $ ./spmd2 2
Hello from thread 0 of 2
Hello from thread 1 of 2
[/npi@raspberrypi:~ $ ./spmd2 1
Hello from thread 0 of 1
[/npi@raspberrypi:~ $ ./spmd2 5
Hello from thread 3 of 5
Hello from thread 4 of 5
Hello from thread 0 of 5
Hello from thread 2 of 5
Hello from thread 1 of 5
[/npi@raspberrypi:~ $ ./spmd2 6
Hello from thread 0 of 6
Hello from thread 4 of 6
Hello from thread 1 of 6
Hello from thread 2 of 6
Hello from thread 3 of 6
Hello from thread 5 of 6
pi@raspberrypi:~ $
```

Figure 6: Corrected Program Thread Counter

In Figure 6, you can see that each thread has its own unique thread count, which is correct.