

Project A2 Task 3 – Parallel Programming

Part 1 - Foundation

1. Identifying the components on the Raspberry PI B+

- a. The PI has a quad-core CPU, 1 Gigabyte of RAM, an Ethernet Controller/Port, 4 USB Ports, an HDMI Port, two Power Connectors (One micro-usb, and one circle-shaped one), Camera and Display I/O pins, and some general-purpose pins at the top for other accessories.
- b. The CPU and RAM are located right under the Raspberry Logo.
- c. The Ethernet Controller is right beside the USB Ports.
- d. The HDMI is between the two Power Connectors.
- e. The Display pins are beside the CPU/RAM module, and the Camera is beside the HDMI Port.

2. How many cores does the Raspberry Pi's B+ CPU have?

- a. It is a quad-core CPU, so four cores.

3. List three main differences between X86 (CISC) and ARM Raspberry PI (RISC).

Justify your answer and use your own words (do not copy and paste)

- a. x86 can perform more complex tasks because it has a larger instruction set than ARM. However, the drawback is that x86 might be slower since it has to interpret a larger set of instructions.
- b. ARM only accesses memory with Load and Store operations. It mainly focuses on performing arithmetic operations and calculating values in its registers. Therefore, ARM has more general purpose registers than x86.
- c. x86 uses little-endian to store variables (the least significant bytes get stored first in memory). ARM works with something called "BI-endian" meaning that it can switch between little-endian and big-endian which is basically the opposite of little-endian in the order that bytes are stored in memory.

4. What is the difference between sequential and parallel computation and identify the practical significance of each?

- a. In sequential computation, tasks given to cpu are handled in a series of instructions--those instructions can only be executed one at a time by a single CPU core. The problem with this configuration is that if an instruction takes a very long time to complete (it takes several clock cycles and leaves the CPU in wait state)
- b. In parallel computation, the CPU will try to break the task into steps that can be done in tandem (at the same time). For example, if the task says "Perform a payroll for 4 different users," the CPU might attempt to assign each of its cores to performing one payroll each at the same time. This way, it can complete multiple instructions at once. One disadvantage to Parallel computing, however, is how hot the CPU can get, so proper cooling is absolutely necessary.

5. Identify the basic form of data and task parallelism in computational problem

- a. According to Professor Michael J. Flynn, there are four different ways to categorize parallelism, based on their "Instruction Stream" and "Data Stream."
 - i. Single-Instruction Single Data Stream (SISD) is a standard non-parallel computer with a single line of sequential task handling.

- ii. Single Instruction Multiple Data Stream (SIMD) is a computer with processors that all perform the same instruction at the same time, but can each operate on different data or data sets. One great use for this type of parallel computing is for graphics/image processing since displaying each pixel often requires the same operation (Selecting the RGB component to display) with different values.
- iii. Multiple Instruction Single Data (MISD) is a computer that only has one data stream that is segmented and fed to all of a computer's cores. One good use for this type of parallelism is brute force attacking, since each core could perform a different type of attack (for example, a dictionary attack, or a rainbow table), but each core will require the same data set.
- iv. Multiple Instruction Multiple Data (MIMD) is the category that most supercomputers fall into today. In this type of parallelism, each processor on the system can perform their own set of given instructions and manipulate their own stream of data. This type of parallel computing is probably **very** complex to implement and not all tasks might be able to maximize this system's performance.

6. Explain the differences between processes and threads.

- a. A process and a thread are very different entities:
 - i. A process is sort of an "instance" of a program. You can run multiple processes of the same program. For example, opening up multiple word documents would open up several instances of Microsoft Word. Each process has a unique process ID attached to it.
 - ii. A thread performs a set of tasks for a process. Multiple threads can work together in a single process. A process can manipulate threads via forking/joining them, or assigning threads some different computational tasks and have them share memory.

7. What is OpenMP and what is OpenMP pragmas?

- a. OpenMP is sort of an API that allows sections of a program to run in parallel. The pdf talks about OpenMP for C++. It uses the MIMD design to allow each thread to work on their data and instruction streams.
- b. OpenMP pragmas are directives for a specific language's compiler that tell it to run some of the source code in parallel. Essentially, the purpose of OpenMP pragmas are to reduce the workload of the programmer, since threaded work is done in the background instead of needing to be implemented into the source code (like pthreads).

8. What applications benefit from multi-core?

- a. Database Servers
- b. Multimedia Applications
- c. Running Virtual Machines
- d. Compilers

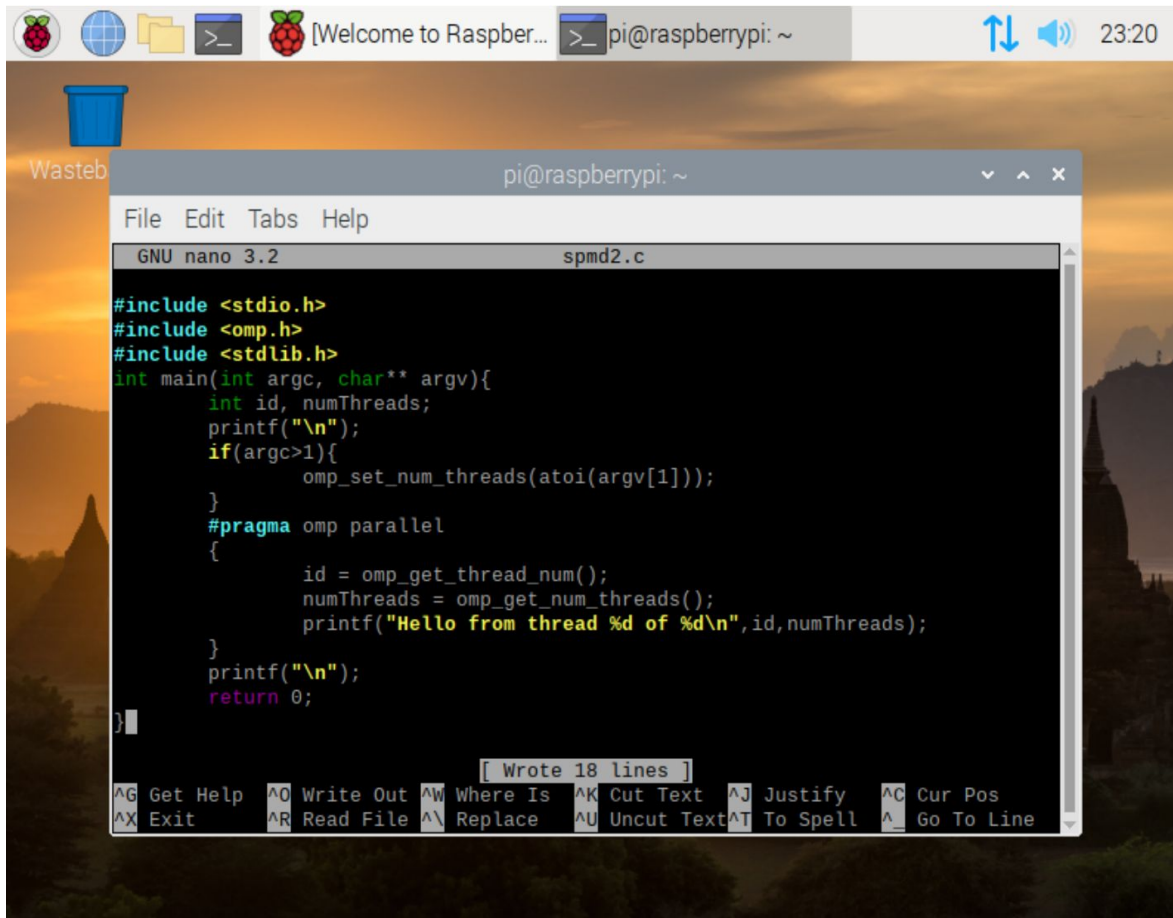
9. Why Multicore?

- a. It is difficult to increase the performance of one core by increasing its clock frequency. It is much easier to add more cores for more performance gains.

- b. It is highly supported by the majority of consumer desktop operating systems, including Windows, Linux, and Mac OS.
- c. Most computer architecture is shifting towards parallel computing design.
- d. One drawback would be that because of the communications between cores, the circuits inside of the cpu are pipelined densely, very close to each other. This means multi-core CPUs will generate a lot of heat, and also that their design is extremely complex compared to a single-core CPU.

Part 2 - Parallel Programming Basics

The first task was to open up the terminal just like last assignment, and create a file using the nano command.



```

GNU nano 3.2      spsmd2.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
    {
        id = omp_get_thread_num();
        numThreads = omp_get_num_threads();
        printf("Hello from thread %d of %d\n",id,numThreads);
    }
    printf("\n");
    return 0;
}
  
```

[Wrote 18 lines]

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
 ^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

This is a c file. Some of the notable features:

1. directives at the top `#include` are like importing packages. in square brackets are some header files that contain important commands like `omp_set_num_threads()`.
2. a main function that returns an int, the standard for a C program. The int and char* array will probably come from input from the terminal.
3. on line 10, there is a very strange piece of code, **`#pragma omp parallel`**. According to the pdf, this line of code assigns a new thread to whatever is inside of the braces after it. This means that segment of code should be executed in parallel with the main function.

After making the C file in nano, I compiled it using gcc, but with some extra operands (-o apparently means to put the result into the operand after it. -fopenmp probably has something to do with invoking OpenMP's pragmas) - gcc spmd2.c -o spmd2 -fopenmp.

Running the new executable using: ./spmd2 4, I added the 4 operand meaning I am passing the integer 4 to the main function. The if condition on line 7 holds true, and we set the number of threads to 4.

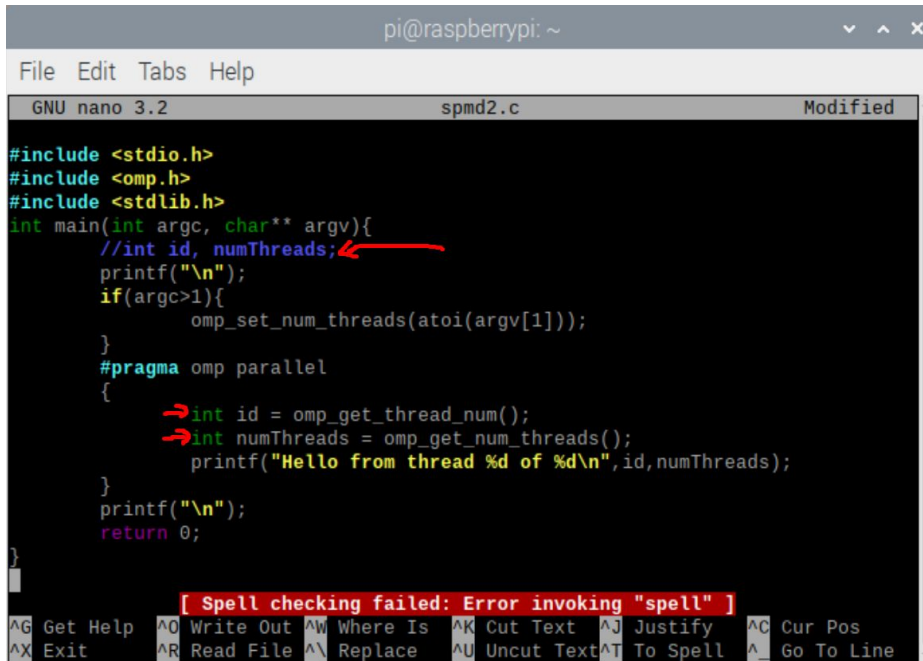
```
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 0 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 1 of 4

pi@raspberrypi:~ $ ./spmd2 10

Hello from thread 0 of 10
Hello from thread 7 of 10
Hello from thread 8 of 10
Hello from thread 8 of 10
Hello from thread 6 of 10
Hello from thread 5 of 10
Hello from thread 4 of 10
Hello from thread 3 of 10
Hello from thread 3 of 10
Hello from thread 1 of 10
```

The order of these print statements seem to be random, which means that the threads do not necessarily work at the same pace. The problem here is how there are some duplicates. It has to do with the variables "id" probably. Because it is declared on line 5, each thread shares the same variable, and they are all modifying the same value. In other words, each thread is sharing the same memory address. So to fix this, I commented out line 5 and declared them entirely on lines 12 and 13. This means each thread should initialize their own "id" and "numThreads".

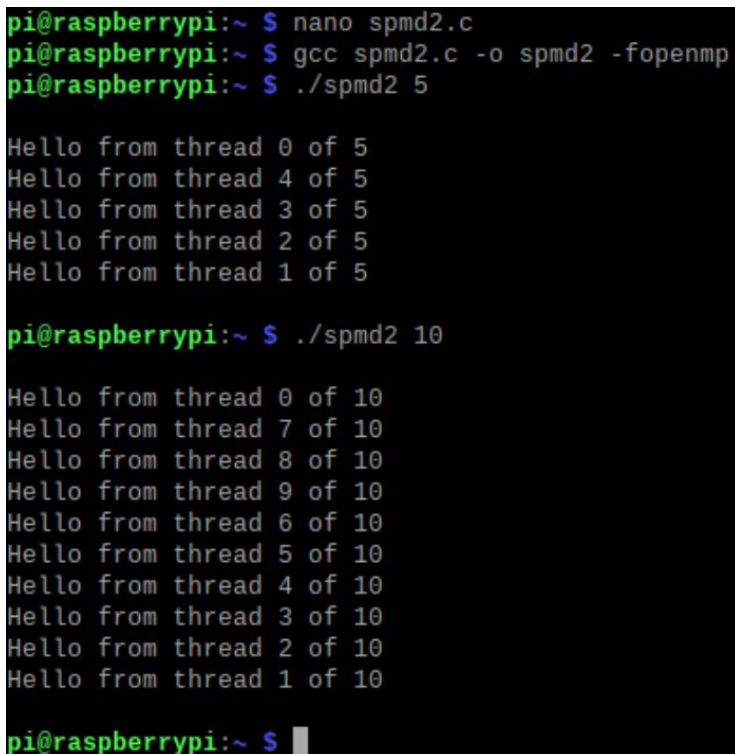


```
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2      spmd2.c      Modified

#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;
}

[ Spell checking failed: Error invoking "spell" ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell  ^_ Go To Line
```

Result:



```
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 5

Hello from thread 0 of 5
Hello from thread 4 of 5
Hello from thread 3 of 5
Hello from thread 2 of 5
Hello from thread 1 of 5

pi@raspberrypi:~ $ ./spmd2 10

Hello from thread 0 of 10
Hello from thread 7 of 10
Hello from thread 8 of 10
Hello from thread 9 of 10
Hello from thread 6 of 10
Hello from thread 5 of 10
Hello from thread 4 of 10
Hello from thread 3 of 10
Hello from thread 2 of 10
Hello from thread 1 of 10

pi@raspberrypi:~ $
```

The order seems random still (as expected), but the issue of int “id” not working properly has been fixed.