

Praveen Doluweera

### Task 3 Parallel Programming Skills

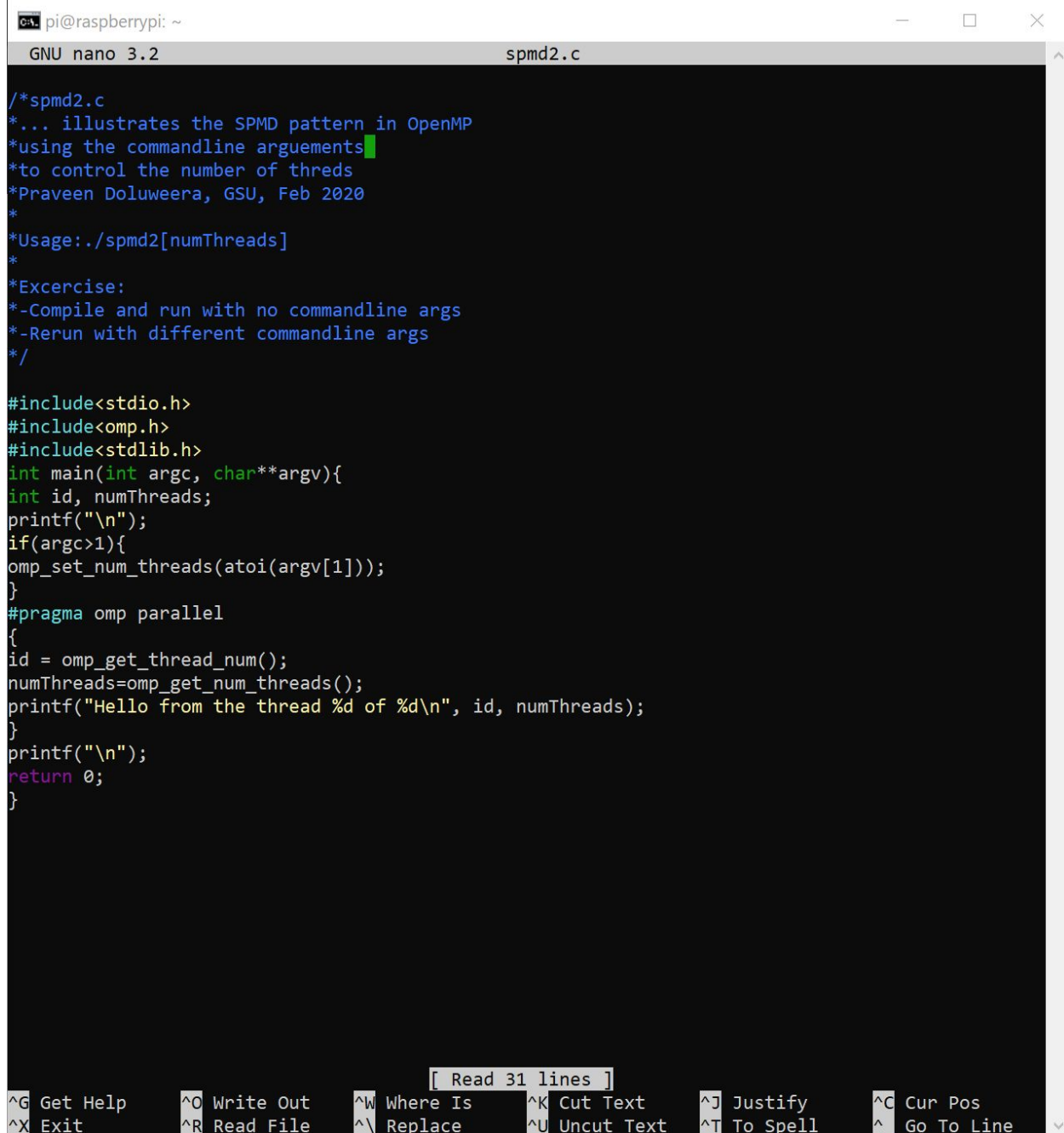
#### a. Foundation

- i. Single Board Computer, quad-core CPU, 1gb ram, power button. i/o (display port, ethernet, hdmi, usb, power input)
- ii. Four
- iii. The first difference between x86 and arm processing is the difference in the ISA. x86 is a CISC (Complex Instruction Set Computing) processor. This allows for more features and a larger instruction set when compared. ARM on the other hand is RISC (Reduced instruction set Computing) which has 100 or less instructions. The instructions operate only on registers which means only the Load/store memory model is used for memory access. Another difference is that in ARM, most instructions can be used for conditional executions. ARM also uses BI-endian format while x86 uses little-endian.
- iv. In sequential computing the problem is broken up into a set of instructions that are meant to be executed one after the other. This means that only one instruction may execute at a time. Sequential computing is significant because it was the start of computing and paved the way for years to come. In parallel computing, there is a simultaneous use of multiple resources to solve a problem. A problem is broken into parts that can be solved concurrently as opposed to sequentially. Instruction from each step can execute simultaneously on different processors to save time. This is significant as it greatly increases the amount of tasks a computer can do.
- v. Data parallelism is a broad category in which the same computation is applied to multiple data items, so that the amount of available parallelism is proportional to the input size. This results in a large amount of potential parallelism. Task parallelism refers to solutions in which parallelism is organized around the functions or tasks that need to be performed rather than around the data itself. In many cases, this does not scale as well as data parallelism.
- vi. A process is the abstraction of a running program and they do not share memory with each other. A thread is a lightweight process that allows a single executable/process to be decomposed into smaller, independant parts. All threads share a common memory unlike processes. The OS schedules threads on separate cores/CPUS if available.
- vii. OpenMP is an api that supports multi platform shared memory multiprocessing for low level languages like C. OpenMP pragmas are compiler directives that enable the compiler to generate threaded code.
- viii. Database servers, web servers, comilers, multimedia applications.

- ix. A multi core cpu operates more than one process at a time compared to a single core processor. This is better for the following reasons
1. Difficult to make single-core clock frequencies even higher
  2. Solves heat problems
  3. Many new applications are multithreaded
  4. There is a general trend in computer architecture to shift to parallelism.

## b. Parallel Programming

First I typed the given code in the nano editor. Everything in the curly brackets should run simultaneously. This means that everything before line 10 and line 15-17 should run sequentially.



```
pi@raspberrypi: ~  
GNU nano 3.2 spmd2.c  
/*spmd2.c  
*... illustrates the SPMD pattern in OpenMP  
*using the commandline arguments  
*to control the number of threds  
*Praveen Doluweera, GSU, Feb 2020  
*  
*Usage: ./spmd2[numThreads]  
*  
*Excercise:  
*-Compile and 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 the thread %d of %d\n", id, numThreads);  
}  
printf("\n");  
return 0;  
}
```

[ Read 31 lines ]

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

Figure 1 - spmd2 original code

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

Hello from the thread 2 of 4
Hello from the thread 3 of 4
Hello from the thread 2 of 4
Hello from the thread 2 of 4

pi@raspberrypi:~ $ ./spmd2 1

Hello from the thread 0 of 1

pi@raspberrypi:~ $ ./spmd2 2

Hello from the thread 0 of 2
Hello from the thread 1 of 2

pi@raspberrypi:~ $ ./spmd2 3

Hello from the thread 0 of 3
Hello from the thread 0 of 3
Hello from the thread 2 of 3

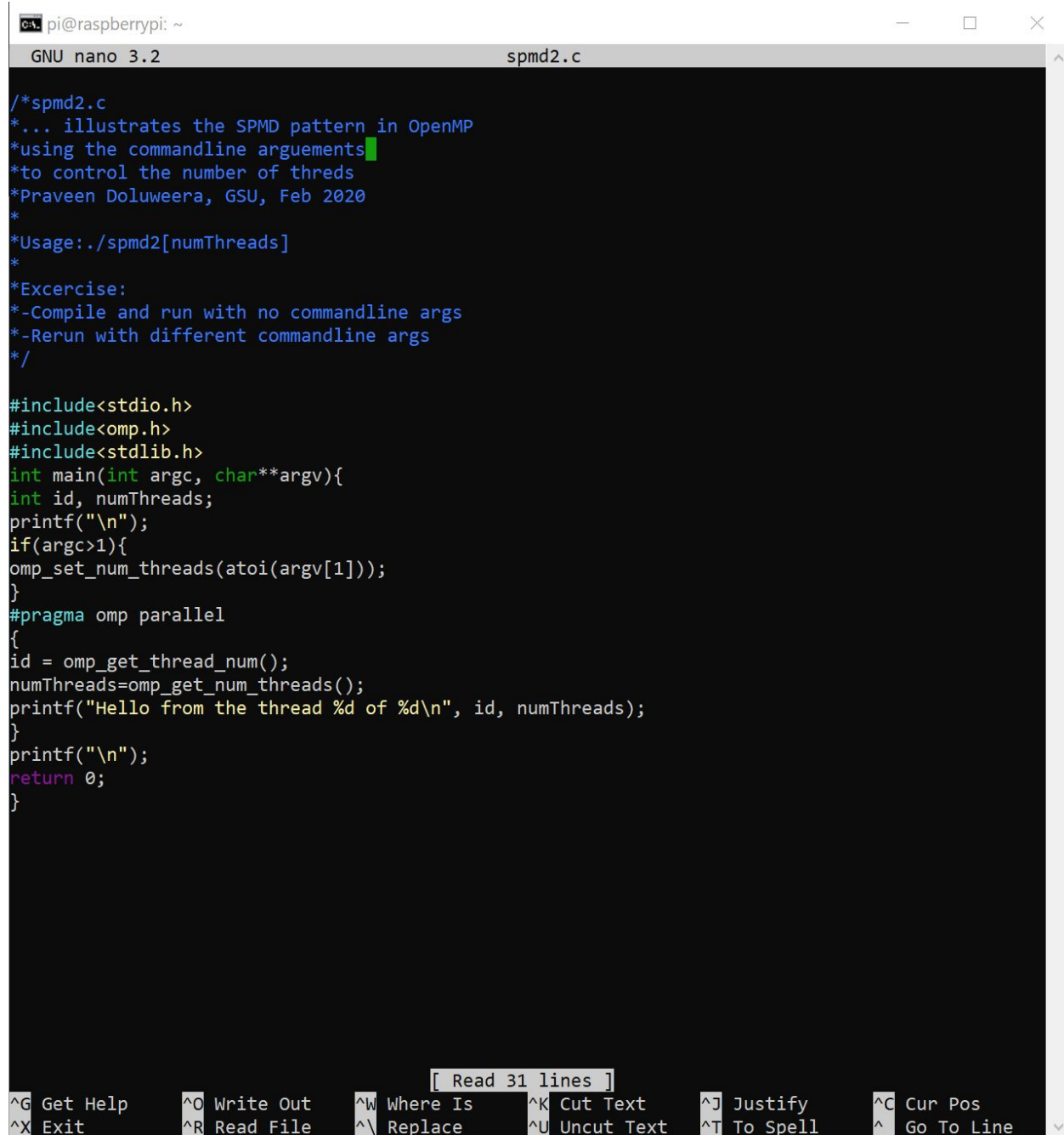
pi@raspberrypi:~ $ ./spmd2 4

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

pi@raspberrypi:~ $
```

Figure 2- Output of the original spmd2 code

The code was then found to have errors that need to be fixed. The output of the code has the same repeated numbers instead of having a unique ID that is associated with each thread. To fix this the code was updated below in Figure 3



```
pi@raspberrypi: ~
GNU nano 3.2 spmd2.c

/*spmd2.c
*... illustrates the SPMD pattern in OpenMP
*using the commandline arguments
*to control the number of threds
*Praveen Doluweera, GSU, Feb 2020
*
*Usage: ./spmd2[numThreads]
*
*Excercise:
*-Compile and 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 the thread %d of %d\n", id, numThreads);
}
printf("\n");
return 0;
}

[ Read 31 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
```

Figure 3 - Updated spmd2 code

The output of this code is shown below in Figure 4. As you can see, each thread has a unique number.

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

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

pi@raspberrypi:~ $ ./spmd2 3

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

pi@raspberrypi:~ $ ./spmd2 2

Hello from the thread 0 of 2
Hello from the thread 1 of 2

pi@raspberrypi:~ $ ./spmd2 1

Hello from the thread 0 of 1

pi@raspberrypi:~ $ ./spmd2 5

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

pi@raspberrypi:~ $ ./spmd2 6

Hello from the thread 1 of 6
Hello from the thread 4 of 6
Hello from the thread 0 of 6
Hello from the thread 3 of 6
Hello from the thread 2 of 6
Hello from the thread 5 of 6

pi@raspberrypi:~ $
```

Figure 4 - Updated output of spmd2