GETTING STARTED WITH THE RASPBERRY PI AND PARALLEL PROGRAMMING

These instructions assume that you have a Raspberry Pi with code to go along with these activities already installed. After starting up your Raspberry Pi, you should have a GUI interface to Raspbian, the Linux variant operating system for these little machines. For this tutorial, you will primarily use the terminal window and type commands into it. You will also use an editor (more on that below).

1.1 DO THIS: The terminal application

Start the terminal window by choosing its icon in the menu bar at the top (it looks like a black square box). Next, you will type this command into that window to get to the directory containing code for this tutorial.

```
cd CSinParallel/RaspberryPiBasics
```

You can type the following command to list what is in this directory:

```
ls
```

You should see three directories listed:

- drugdesign
- integration
- patternlets

The rest of this tutorial will lead you through some code found in the patternlets and integration directories. These are examples that let you explore how we can write code in the C programming language that will use the four cores available on the processor in the Raspberry Pi.

1.2 Heads up: You will also use an editor

In addition to the terminal application, the other tool on the Raspberry Pi that will be useful to you is an editor for slightly changing some of the code examples and seeing what will happen after you re-build them. Look for the editor called Geany in the Programming section of the main Menu in the upper left of your Raspberry Pi's screen. You will use this Geany editor and the Terminal to try out the code examples.

> **Alternative editor**: You could also use an editor that you run in the terminal called nano. You simply type this in the terminal window, along with the file name (examples to follow).

1.3 VERY useful terminal tricks

1.3.1 1. Tab completion of long names

You can complete a command without typing the whole thing by using the Tab key. For example, make sure you are still in the directory /home/pi/CSinParallel/RaspberryPiBasics by typing

```
pwd
```

Then try listing one of the three subdirectories by starting to type the first few letters, like this, and hit tab and see it complete the name of the directory:

```
ls pat[Tab]
```

1.3.2 2. History of commands with up and down arrow

Now that you have typed a few commands, you can get back to previous ones by hitting the up arrow key. If you hit up arrow more than once, you can go back down by hitting the down arrow key, all the way back to the prompt waiting for you to type.

You can get a blank prompt at any time (even after starting to type and changing your mind) by typing control and the u key simultaneously together.

2.1 DO THIS: Look at some code

After following the steps in the terminal window in the previous chapter, you should now be able to do this next in the terminal window at the prompt:

```
cd patternlets/spmd
ls
```

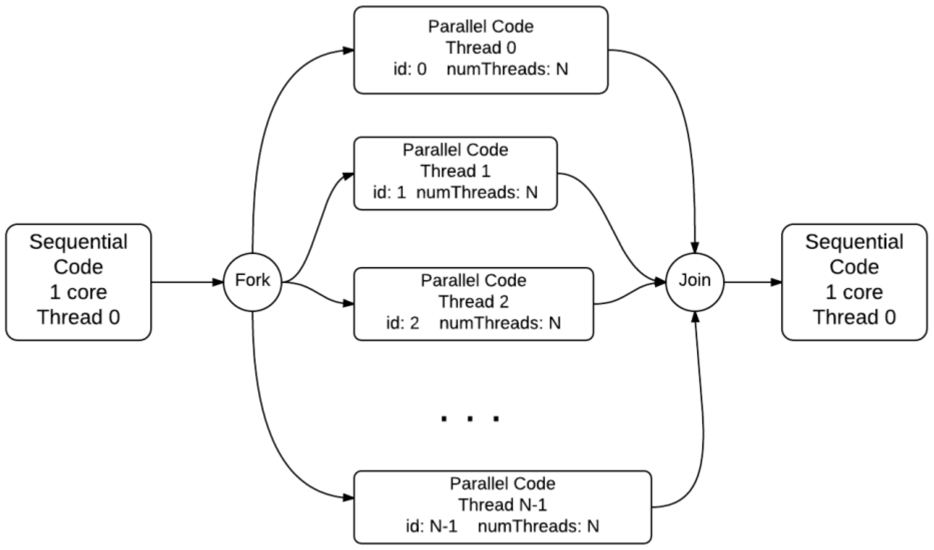
The ls command should show you two files:

- Makefile
- spmd2.c

We will start by examining spmd2.c, a C program that uses special additions to the language that make it easy to run a portion of your program on multiple threads on the different cores of a multicore machine. These additions to the C language are called OpenMP. Here is the code:

```c
/* spmd2.c
 * ... illustrates the SPMD pattern in OpenMP,
 *        using the commandline arguments
 *      to control the number of threads.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./spmd2 [numThreads]
 *
 * Exercise:
 * - Compile & run with no commandline args
 * - Rerun with different commandline args,
 *    until you see a problem with thread ids
 * - Fix the race condition
 *    (if necessary, compare to 02.spmd)
 */

#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;
}
```

Line number 30 in the above code is how with OpenMP a programmer can direct that the next block of code within the curly braces should be run simultaneously, or concurrently, on a given number of separate threads, or small subprocesses of the original program. When the program is run, at the point of the curly brace on line 31, new threads get forked , each with their own id, and run separate copies of the code inside the block between the braces. Conceptually, this looks like this:



Sequential Code 1 core Thread 0
Fork
Parallel Code Thread 0 id: 0 numThreads: N
Parallel Code Thread 1 id: 1 numThreads: N
Parallel Code Thread 2 id: 2 numThreads: N
. . .
Parallel Code Thread N-1 id: N-1 numThreads: N
Join
Sequential Code 1 core Thread 0

The code in main up until line 30 is run in one thread on one core, the forking of separate threads to run the code between lines 31 and 35 is shown in the middle of the diagram. The final last couple of lines of code are run back in the single thread 0 after all the threads have completed and join back to the main thread.

2.2 DO THIS: Build and Run the code

In the terminal window, you can make the executable program by typing:

```
make
```

This should create a file called spmd2, which is the executable program.

To run the program, type:

```
./spmd2 4
```

The 4 is called a command-line argument that indicates how many threads to fork. Since we have a 4-core processor on the Raspberry Pi, it is natural to try 4 threads. You can also try 1, 2, 3, or more than 4. Try it and observe the results!

2.3 CHECK IT: Is it correct?

Did it seem to work? Try running it several times with 4 threads (remember the up arrow key will help make this easier) – something should be amiss. Can you figure it out?

The order in which the threads finish and print is not the problem- that happens naturally because there is no guarantee for when a thread will finish (that is illustrated in the diagram above).

What is not good is when you see a thread id number appearing more than once. Each thread is given its own unique id. This code has a problem that often happens with these kinds of parallel programs. The Pi's processor may have multiple cores, but those cores share one bank of memory in the machine. Because of this, we need to be a bit more careful about declaring our variables. When we declare a variable outside of the block that will be forked and run in parallel on separate threads, all threads share that variable's memory location. This is a problem with this code, since we want each thread to keep track of its id separately (and write it to memory on line 31).

2.4 DO THIS: Fix the code

Try opening the Geany editor from the menu under Programming. Then use it to open this file and edit it as given just below here:

CSinParallel/RaspberryPiBasics/patternlets/spmd/spmd2.c

Alternative editor: If you wish you can also try the simple text editor called nano by typing this:

nano spmd2.c

Control-w writes the file; control-x exits the editor.

The exact way that the code should look is shown in the listing below.

2.4.1 The steps to fix the code are:

1. comment line 23 with a // at the very front of the line
2. fix lines 32 and 33 to be full variable declarations (see listing below)

When you make this change, and save it, you are now saying in the code that each thread will now have its own private copy of the variables named id and numThreads.

Try building the code with make and running it again several times, using 4 or more threads. Each time you should now get unique thread numbers printed out (but not necessarily in order).

Your changed code needs to look like this:

```
1  /* spmd2.c
2   * ... illustrates the SPMD pattern in OpenMP,
3   *        using the commandline arguments
4   *      to control the number of threads.
5   *
6   * Joel Adams, Calvin College, November 2009.
7   *
8   * Usage: ./spmd2 [numThreads]
9   *
10  * Exercise:
```

```c
11  * - Compile & run with no commandline args
12  * - Rerun with different commandline args,
13  *    until you see a problem with thread ids
14  * - Fix the race condition
15  *    (if necessary, compare to 02.spmd)
16  */
17
18 #include <stdio.h>
19 #include <omp.h>
20
21 int main(int argc, char** argv) {
22 //    int id, numThreads;
23
24     printf("\n");
25     if (argc > 1) {
26         omp_set_num_threads( atoi(argv[1]) );
27     }
28
29     #pragma omp parallel
30     {
31         int id = omp_get_thread_num();
32         int numThreads = omp_get_num_threads();
33         printf("Hello from thread %d of %d\n", id, numThreads);
34     }
35
36     printf("\n");
37     return 0;
38 }
39
```

2.5 Optional Aside: Patterns in programs

This really small program illustrates a couple of standard ways that programmers writing parallel programs design their code. These standard, tried and true methods that are effective for good programmers are referred to as patterns in programs. This code illustrates the fork-join programming pattern for parallel programs, and is built into OpenMP through the use of the pragma on line 30. The other pattern illustrated is called single program, multiple data (thus the name spmd2 for this program). We see here one program code file that can at times run parts of the code separately on different data values stored in memory. This is also built into OpenMP and saves the hassle of writing more than one program file for the code to run on separate threads.

Since this code is very small yet illustrates these patterns, we have coined it a patternlet.