GETTING STARTED WITH THE RASPBERRY PI AND PARALLEL PROGRAMMING

These instructions assume that you have a Raspberry Pi up and running. 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).

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. You could 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, try listing one of the directories by starting to type the first few letters, like this, and hit tab and see it complete the name of the directory:

ls Dow[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.

2.0 INTEGRATION USING THE TRAPEZOIDAL RULE

2.1 DO THIS: Look at some code

Here we will try a slightly more complex example of a complete program that uses the parallel loop pattern. The following programs attempt to compute a Calculus value, the "trapezoidal approximation of

$$\int_0^{\pi} \sin(x)dx$$

using 2^{20} equal subdivisions." **The answer from this computation should be 2.0.**

Here is a parallel program, called trap-notworking.c, that attempts to do this:

```
1 //The answer from this computation should be 2.0.
2 #include <math.h>
3 #include <stdio.h> // printf()
4 #include <stdlib.h> // atoi()
5 #include <omp.h> // OpenMP
6
7
8 /* Demo program for OpenMP: computes trapezoidal approximation to an integral*/
9
10 const double pi = 3.14159265358979323846264338307 9;
11
12 int main(int argc, char** argv) {
13 /* Variables */
14 double a = 0.0, b = pi; /* limits of integration */;
15 int n = 1048576; /* number of subdivisions = 2^20 */
16 double h = (b - a) / n; /* width of subdivision */
17 double integral; /* accumulates answer */
18 int threadcnt = 1;
19
20 double f(double x);
21
22 /* parse command-line arg for number of threads */
23 if (argc > 1) {
24 threadcnt = atoi(argv[1]);
25 }
26
27 #ifdef _OPENMP
28 omp_set_num_threads( threadcnt );
29 printf("OMP defined, threadct = %d\n", threadcnt);
30 #else
31 printf("OMP not defined");
32 #endif
33
34 integral = (f(a) + f(b))/2.0;
35 int i;
36
37 #pragma omp parallel for private(i) shared (a, n, h, integral)
38 for(i = 1; i < n; i++) {
39 integral += f(a+i*h);
40 }
41
42 integral = integral * h;
43 printf("With %d trapezoids, our estimate of the integral from \n", n);
44 printf("%f to %f is %f\n", a,b,integral);
45 }
46
47 double f(double x) {
48 return sin(x);
49 }
```

The line to pay attention to is line 37, which contains the pragma for parallelizing the loop that is computing the integral. This example is different from others you have seen so far, in that the variables are explicitly declared to be either private (each thread gets its own copy) or shared (all threads use the same copy of the variable in memory).

2.2 Will it work?

As you can guess from the name of the file, this one above has a problem on line 37. Here is the improved version, called trap-working.c:

```
1 //The answer from this computation should be 2.0.
2 #include <math.h>
3 #include <stdio.h> // printf()
4 #include <stdlib.h> // atoi()
5 #include <omp.h> // OpenMP
6
7
8 /* Demo program for OpenMP: computes trapezoidal approximation to an integral*/
9
10 const double pi = 3.14159265358979323846264338307 9;
11
12 int main(int argc, char** argv) {
13 /* Variables */
14 double a = 0.0, b = pi; /* limits of integration */;
15 int n = 1048576; /* number of subdivisions = 2^20 */
16 double h = (b - a) / n; /* width of subdivision */
17 double integral; /* accumulates answer */
18 int threadcnt = 1;
19
20 double f(double x);
21
22 /* parse command-line arg for number of threads */
23 if (argc > 1) {
24 threadcnt = atoi(argv[1]);
25 }
26
27 #ifdef _OPENMP
28 omp_set_num_threads( threadcnt );
29 printf("OMP defined, threadct = %d\n", threadcnt);
30 #else
31 printf("OMP not defined");
32 #endif
33
34 integral = (f(a) + f(b))/2.0;
35 int i;
36
37 #pragma omp parallel for \
38 private(i) shared (a, n, h) reduction(+: integral)
39 for(i = 1; i < n; i++) {
40 integral += f(a+i*h);
41 }
42
43 integral = integral * h;
44 printf("With %d trapezoids, our estimate of the integral from \n", n);
45 printf("%f to %f is %f\n", a,b,integral);
46 }
47
48 double f(double x) {
49 return sin(x);
50 }
```

As with the previous reduction patternlet example, we again have an accumulator variable in the loop, this time called integral. So we once again need to use the reduction clause for that variable.

Comments:

- Make sure no characters follow the backslash character before the end of the first line. This causes the two lines to be treated as a single pragma (useful to avoid long lines).
- The phrase omp parallel for indicates that this is an OpenMP pragma for parallelizing the for loop that follows immediately. The OpenMP system will divide the 2^{20} iterations of that loop up into threadct segments, each of which can be executed in parallel on multiple cores.
- The OpenMP clause num_threads(threadct) specifies the number of threads to use in the parallelization.
- The clauses in the second line indicate whether the variables that appear in the for loop should be shared with the other threads, or should be local private variables used only by a single thread. Here, four of those variables are globally shared by all the threads, and only the loop control variable i is local to each particular thread.

2.2 DO THIS: Edit, Compile and Run the code

First, type or (copy and paste) the following code in a terminal window:

nano trap-notworking.c

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

After you save and exit from nano, you can make the executable program by typing:

gcc trap-notworking.c -o trap-notworking -fopenmp

This should create a file called **trap-notworking**, which is the executable program.

Now, we do the same for trap-working.c:

nano trap-working.c

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

After you save and exit from nano, you can make the executable program by typing:

gcc trap-working.c -o trap-working -fopenmp

This should create a file called **trap-working**, which is the executable program.

Finally, run each one to compare the results (what did you observe and why): To run the programs, type:

./ trap-notworking 4

./ trap-working 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.

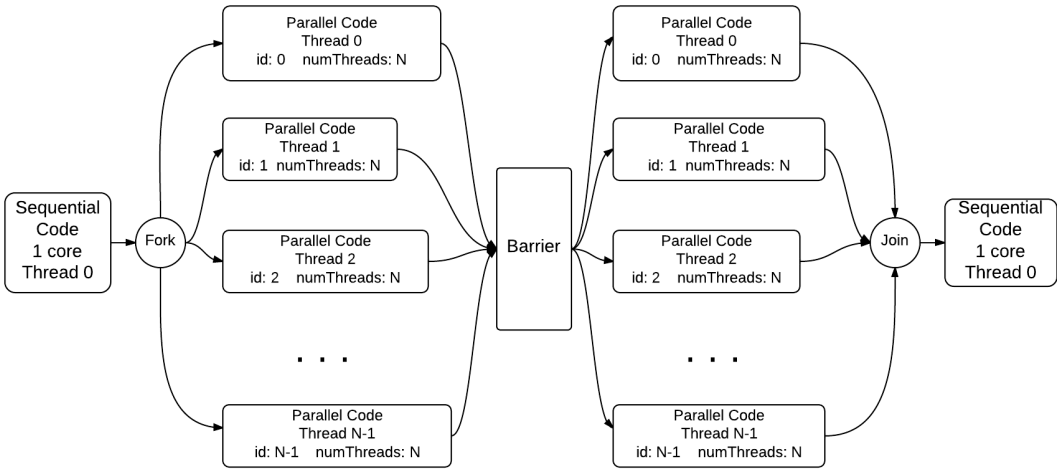
2.3 Data Decomposition: "equal chunks"

Because we did not explicitly add an additional static or dynamic clause to the pragma on lines 37-38 on the working version above, the default behavior of assigning equal amounts of work doing consecutive iterations of the loop was used to decompose the problem onto threads. In this case, since the number of trapezoids used was 1048576, then with 4 threads, thread 0 will do the first 1048576/4 trapezoids, thread 1 the next 1048576/4 trapezoids, and so on.

3.0 Coordination: Synchronization with a Barrier

The barrier pattern is used in parallel programs to ensure that all threads complete a parallel section of code before execution continues. This can be necessary when threads are generating computed data (in an array, for example) that needs to be completed for use in another computation.

Conceptually, the running code is executing like this:



3.1 DO THIS: Look at some code

First, type or (copy and paste) the following code in a terminal window:

nano barrier.c

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

After you save and exit from nano, you can make the executable program by typing:

gcc barrier.c -o barrier -fopenmp

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

Note what happens with and without the commented pragma on line 31.

```
1   /* barrier.c
2    * ... illustrates the use of the OpenMP barrier command,
3    *          using the commandline to control the number of threads...
4    *
5    * Joel Adams, Calvin College, May 2013.
6    *
7    * Usage: ./barrier [numThreads]
8    *
9    * Exercise:
10   * - Compile & run several times, noting interleaving of outputs.
11   * - Uncomment the barrier directive, recompile, rerun,
12   *    and note the change in the outputs.
13   */
14
15  #include <stdio.h>
```

```
16  #include <omp.h>
17  #include <stdlib.h>
18
19  int main(int argc, char** argv) {
20     printf("\n");
21     if (argc > 1) {
22        omp_set_num_threads( atoi(argv[1]) );
23     }
24
25     #pragma omp parallel
26     {
27        int id = omp_get_thread_num();
28        int numThreads = omp_get_num_threads();
29        printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);
30
31  //       #pragma omp barrier
32
33        printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
34     }
35
36     printf("\n");
37     return 0;
38  }
39
```

4.0 Program Structure: The Master-Worker Implementation Strategy

Once you have mastered the notion of fork-join and single-program, multiple data, the next common pattern that programmers use in association with these patterns is to have one thread, called the master, execute one block of code when it forks while the rest of the threads, called workers, execute a different block of code when they fork. This is illustrated in this simple example (useful code would be more complicated).

4.1 DO THIS: Look at some code

First, type or (copy and paste the) following code the following in a terminal window:

nano masterWorker.c

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

After you save and exit from nano, you can make the executable program by typing:

gcc masterWorker.c -o masterWorker -fopenmp

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

Note what happens with and without the commented pragma on line 24.

- re-compile and re-run
- Compare and trace the different executions.

```
1   /* masterWorker.c
2    * ... illustrates the master-worker pattern in OpenMP
3    *
4    * Joel Adams, Calvin College, November 2009.
5    *
6    * Usage: ./masterWorker
```

```
 *
 * Exercise:
 * - Compile and run as is.
 * - Uncomment #pragma directive, re-compile and re-run
 * - Compare and trace the different executions.
 */

#include <stdio.h>   // printf()
#include <stdlib.h>   // atoi()
#include <omp.h>    // OpenMP

int main(int argc, char** argv) {
   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();

      if ( id == 0 ) {  // thread with ID 0 is master
         printf("Greetings from the master, # %d of %d threads\n",
                                    id, numThreads);
      } else {        // threads with IDs > 0 are workers
         printf("Greetings from a worker, # %d of %d threads\n",
                                    id, numThreads);
      }
   }

   printf("\n");

   return 0;
}

```