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.

RUNNING LOOPS IN PARALLEL

2.1 DO THIS: Observe the code

An iterative for loop is a remarkably common pattern in all programming, primarily used to perform a calculation N times, often over a set of data containing N elements, using each element in turn inside the for loop. If there are no dependencies between the calculations (i.e. the order of them is not important), then the code inside the loop can be split between forked threads. When doing this, a decision the programmer needs to make is to decide how to partition the work between the threads by answering this question:

- How many and which iterations of the loop will each thread complete on its own?

We refer to this as the ***data decomposition pattern*** because we are decomposing the amount of work to be done (typically on a set of data) across multiple threads. In the following code, this is done in OpenMP using the omp parallel for *pragma* just in front of the for statement (line 10) in the following code.

```
/* parallelLoopEqualChunks.c
        ... illustrates the use of OpenMP's default parallel for loop in which
        threads iterate through equal sized chunks of the index range
        (cache-beneficial when accessing adjacent memory locations).
*
        Joel Adams, Calvin College, November 2009.
*
        Usage: ./parallelLoopEqualChunks [numThreads]
*
        Exercise
        - Compile and run, comparing output to source code
        - try with different numbers of threads, e.g.: 2, 3, 4, 6, 8
*/

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

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if (argc > 1) {
  omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
 int id = omp_get_thread_num();
 printf("Thread %d performed iteration %d\n", id, i);
}

 printf("\n");
 return 0;
}
```

2.2 DO THIS: Edit, Compile and Run the code

First, to create and type or copy past the above code (the one in section 2.1), type the following in a terminal window:

```
nano parallelLoopEqualChunks.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  parallelLoopEqualChunks.c -o pLoop -fopenmp
```

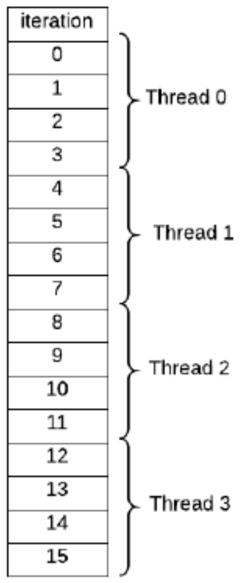
This should create a file called pLoop, which is the executable program.

To run the program, type:

```
./pLoop 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. Replace 4 with other values for the number of threads, or leave off.

When you run it with 4 threads, verify that the behavior is this sort of decomposition of work to threads:



What happens when the number of iterations (16 in this code) is not evenly divisible by the number of threads? Try several cases to be certain how the compiler splits up the work. This type of decomposition is commonly used when accessing data that is stored in consecutive memory locations (such as an array) that might be cached by each thread.

3.2.1 Chunks of work

We've coined this way of dividing the work into consecutive iterations of the loop as working on chunks of the work, or data to be computed, at a time.

3.3 Another way to divide the work

There is an alternative to having each thread do several consecutive iterations of a loop: dole out one iteration of the loop to one thread, the next to the next thread, and so on. When each thread completes its iteration, it can get the next one assigned to it. The assignment is done statically, so that each thread still has equal amounts of work, just not consecutive iterations.

2.2 DO THIS: Edit, Compile and Run the code

First, to create and type or copy past the following code, type the following in a terminal window:

nano parallelLoopChunksOf1.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 parallelLoopChunksOf1.c -o pLoop2 -fopenmp

This should create a file called pLoop2, which is the executable program.

To run the program, type:

```
./pLoop2 4
```

Replace 4 with other values for the number of threads, or leave off.

Now note the difference in how the work of each iteration was assigned.

The code for this example is as follows:

```
/* parallelLoopChunksOf1.c
 * ... illustrates how to make OpenMP map threads to
 *           parallel loop iterations in chunks of size 1
 *           (use when not accesssing memory).
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./parallelLoopChunksOf1 [numThreads]
 *
 * Exercise:
 * 1. Compile and run, comparing output to source code,
 *    and to the output of the 'equal chunks' version.
 * 2. Uncomment the "commented out" code below,
 *    and verify that both loops produce the same output.
 *    The first loop is simpler but more restrictive;
 *    the second loop is more complex but less restrictive.
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if (argc > 1) {
omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for schedule(static,1)
for (int i = 0; i < REPS; i++) {
int id = omp_get_thread_num();
printf("Thread %d performed iteration %d\n", id, i);
}
/*
printf("\n---\n\n");

#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
for (int i = id; i < REPS; i += numThreads) {
printf("Thread %d performed iteration %d\n",
        id, i);
}
}
*/
printf("\n");
return 0;
}
```

3.3.2 Static scheduling

Note that in line 10 in this code we have added a clause to the omp parallel for pragma that is saying that we wish to schedule each thread to do one iteration of the loop in a regular pattern. That is what the keyword static means.

Show and explain the output after running the program.

Note:

3.4 Dynamic scheduling for time-varying tasks

There is also the capability of assigning the threads dynamically, so that when a thread completes, it gets the next iteration or chunk still needed to be done. We do this by changing the word static to dynamic. You can also change the 1 to different sized chunks. This way of dividing, or decomposing the work can be useful when the time each iteration takes to complete its work may vary. Threads that take a short time can pick up the next bit of work while longer threads are still chugging on their piece.

WHEN LOOPS HAVE DEPENDENCIES

Very often loops are used with an accumulator variable to compute a single value from a set of values, such as sum of integers in an array or list. Since this is so common, we can do this in parallel in OpenMP also.

4.1 DO THIS: Look at some code

The code file is called reduction.c and looks like this:

```
/* reduction.c
 * ... illustrates the OpenMP parallel-for loop's reduction clause
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./reduction
 *
 * Exercise:
 * - Compile and run.  Note that correct output is produced.
 * - Uncomment #pragma in function parallelSum(),
 *    but leave its reduction clause commented out
 * - Recompile and rerun.  Note that correct output is NOT produced.
 * - Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()
 * - Recompile and rerun.  Note that correct output is produced again.
 */
#include <stdio.h>   // printf()
#include <omp.h>     // OpenMP
#include <stdlib.h>  // rand()

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
int array[SIZE];

 if (argc > 1) {
  omp_set_num_threads( atoi(argv[1]) );
 }
13.   initialize(array, SIZE);
14.   printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
15.   sequentialSum(array, SIZE),
16.   parallelSum(array, SIZE) );

17.   return 0;
18. }

19. /* fill array with random values */
20. void initialize(int* a, int n) {
21.   int i;
22.   for (i = 0; i < n; i++) {
23.    a[i] = rand() % 1000;
24.   }
25. }

26. /* sum the array sequentially */
27. int sequentialSum(int* a, int n) {
28.   int sum = 0;
29.   int i;
30.  for (i = 0; i < n; i++) {
31.   sum += a[i];
32.  }
33.  return sum;
34. }

35. /* sum the array using multiple threads */
36. int parallelSum(int* a, int n) {
37. int sum = 0;
38. int i;
39. //   #pragma omp parallel for // reduction(+:sum)
40.  for (i = 0; i < n; i++) {
41.   sum += a[i];
42.  }
43.  return sum;
44. }
```

In this example, an array of randomly assigned integers represents a set of shared data (a more realistic program would perform a computation that creates meaningful data values; this is just an example). Note the common sequential code pattern found in the function called sequentialSum in the code below (starting line 26): a for loop is used to sum up all the values in the array.

Next let’s consider how this can be done in parallel with threads. Somehow the threads must implicitly communicate to keep the overall sum updated as each of them works on a portion of the array. In the parallelSum function, line 39 shows a special clause that can be used with the parallel for pragma in OpenMP for this. All values in the array are summed together by using the OpenMP parallel for pragma with the reduction(+:sum) clause on the variable sum, which is computed in line 41.

The plus sign in the pragma redcuction clause indicates the variable sum is being computed by adding values together in the loop.

4.2 DO THIS: Edit, Compile and Run the code

First, to create and type or copy past the following code, type the following in a terminal window:

```
nano reduction.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  reduction.c -o reduction -fopenmp
```

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

To run the program, type:

```
./reduction 4
```

Replace 4 with other values for the number of threads, or leave off.

Show and explain the output after running the program.

4.3 DO: Edit, Compile and Run in Parallel

Now you need an editor to edit the code (Geany program from menu or nano at the terminal command line).

1) To actually run it in parallel, you will need to remove the // comment at the front of line 39. Try removing only the first // and compile it and run it again (as given above). Does the sequentialSum function result (given first) match the parallelSum function's answer?
2) Now try also removing the second // so that the reduction clause is uncommented and compile it run it again. What happens?

4.3.1 Something to think about

Do you have any ideas about why the parallel for pragma without the reduction clause did not produce the correct result?

It is related to why we had to change the declaration of variables in the spmd example so that they were private, with each thread having its own copy. In this parallel for loop example, the reduction variable, or accumulator called sum, needs to be private to each thread as it does its work. The reduction clause in this case makes that happen. Then when each thread is finished the final sum of their individual sums is computed.

In this type of parallel loop, the variable sum is said to have a dependency on what all the threads are doing to compute it.