Raspberry Pi Basics

CSInParallel Project

Sep 11, 2016

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH THE RASPBERRY PI

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 linix 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.

Note: Throughout this document, when you see some text inside a box like the two above and the two below in this chapter, these are commands that you type in the terminal.

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.

CHAPTER

TWO

A SIMPLE PARALLEL PROGRAM

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:

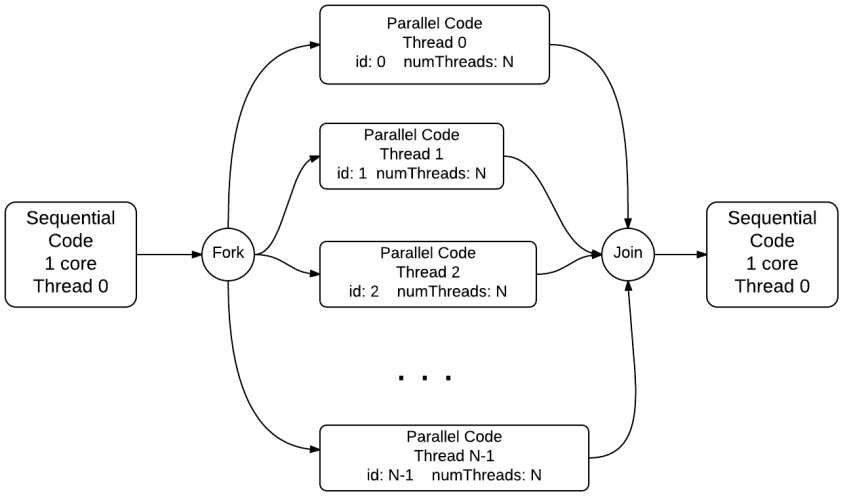
- Makefile
- spmd2.c

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:

```
/* 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]) );
```

```
28      }
29
30      #pragma omp parallel
31      {
32          id = omp_get_thread_num();
33          numThreads = omp_get_num_threads();
34          printf("Hello from thread %d of %d\n", id, numThreads);
35      }
36
37      printf("\n");
38      return 0;
39  }
40
```

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 sub-processes 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:



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 Rasberry 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).

PLEASE REFER TO THE SEPARATE 1-page HANDOUT DESCRIBING THIS PROBLEM, CALLED A RACE CONDITION

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:
```

```
 * - 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>

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;
}
```

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*.

CHAPTER

THREE

RUNNING LOOPS IN PARALLEL

Next we will consider a program that has a loop in it. Since loops appear often our programs, it is useful and natural to try to make use of multiple cores to run the code in the loop in parallel if possible.

3.1 DO THIS: Observe the code

From the spmd directory of the previous example, go to the next example like this (don't forget the tab key to complete the typing for you):

```
cd ../parallelLoop-equalChunks
```

In this directory there are two files:

```
Makefile
parallelLoopEqualChunks.c
```

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 27) in the following code.

```
1  /* parallelLoopEqualChunks.c
2   * ... illustrates the use of OpenMP's default parallel for loop in which
3   *          threads iterate through equal sized chunks of the index range
4   *        (cache-beneficial when accessing adjacent memory locations).
5   *
6   * Joel Adams, Calvin College, November 2009.
7   *
8   * Usage: ./parallelLoopEqualChunks [numThreads]
9   *
10  * Exercise
11  * - Compile and run, comparing output to source code
12  * - try with different numbers of threads, e.g.: 2, 3, 4, 6, 8
13  */
14
15 #include <stdio.h>    // printf()
16 #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;
}

```

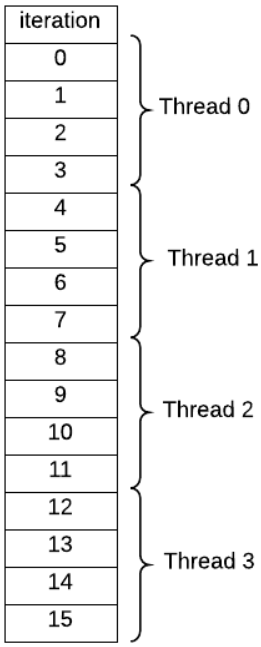
3.2 DO THIS: Build and Run

Try making the program and running it:

```
make
./parallelLoopEqualChunks 4
```

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.

DO THIS: This is illustrated with another example, which you can find by going to another directory:

```
cd ../parallelLoop-chunksOf1
```

3.3.1 DO THIS: Build and Run

Similar to the other examples, you can make the program and run it as follows:

```
make
./parallelLoopChunksOf1 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:

```
1  /* parallelLoopChunksOf1.c
2   * ... illustrates how to make OpenMP map threads to
3   *        parallel loop iterations in chunks of size 1
4   *        (use when not accesssing memory).
5   *
6   * Joel Adams, Calvin College, November 2009.
7   *
8   * Usage: ./parallelLoopChunksOf1 [numThreads]
9   *
10  * Exercise:
11  * 1. Compile and run, comparing output to source code,
12  *    and to the output of the 'equal chunks' version.
13  * 2. Uncomment the "commented out" code below,
14  *    and verify that both loops produce the same output.
15  *    The first loop is simpler but more restrictive;
16  *    the second loop is more complex but less restrictive.
17  */
18
19 #include <stdio.h>
20 #include <omp.h>
21 #include <stdlib.h>
22
```

```
23 int main(int argc, char** argv) {
24     const int REPS = 16;
25 
26     printf("\n");
27     if (argc > 1) {
28         omp_set_num_threads( atoi(argv[1]) );
29     }
30 
31     #pragma omp parallel for schedule(static,1)
32     for (int i = 0; i < REPS; i++) {
33         int id = omp_get_thread_num();
34         printf("Thread %d performed iteration %d\n",
35                  id, i);
36     }
37 
38 /*
39     printf("\n---\n\n");
40 
41     #pragma omp parallel
42     {
43         int id = omp_get_thread_num();
44         int numThreads = omp_get_num_threads();
45         for (int i = id; i < REPS; i += numThreads) {
46             printf("Thread %d performed iteration %d\n",
47                      id, i);
48         }
49     }
50 */
51     printf("\n");
52     return 0;
53 }
54 
```

3.3.2 Static scheduling

Note that in line 31 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.

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.

3.5 Optional Aside: More Programming Patterns

This way of splitting the work in a loop is called the *parallel loop pattern*, and the way that we are using threads to compute over different data being calculated in that loop is called a *data decomposition pattern*.

CHAPTER

FOUR

WHEN LOOPS HAVE DEPENDENCIES

Very often loops are used with an *accumulator* variable to compute a 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

Consider the following code example, which you can get to from the previous example by doing this command:

```
cd ../reduction
```

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]) );
   }
```

```
33
34    initialize(array, SIZE);
35    printf("\nSeq. sum: \t%d\nPar. sum: \t%d\n\n",
36              sequentialSum(array, SIZE),
37              parallelSum(array, SIZE) );
38
39    return 0;
40 }
41
42 /* fill array with random values */
43 void initialize(int* a, int n) {
44    int i;
45    for (i = 0; i < n; i++) {
46       a[i] = rand() % 1000;
47    }
48 }
49
50 /* sum the array sequentially */
51 int sequentialSum(int* a, int n) {
52    int sum = 0;
53    int i;
54    for (i = 0; i < n; i++) {
55       sum += a[i];
56    }
57    return sum;
58 }
59
60 /* sum the array using multiple threads */
61 int parallelSum(int* a, int n) {
62    int sum = 0;
63    int i;
64 //   #pragma omp parallel for // reduction(+:sum)
65    for (i = 0; i < n; i++) {
66       sum += a[i];
67    }
68    return sum;
69 }
70
```

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 51): 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 64 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 66.

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

4.2 DO THIS : Build and Run Sequentially

Do the following to make the program and run it:

```
make
./reduction 4
```

4.3 DO THIS : Edit, Build 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 64. Try removing only the first // and make it and run it again (as given above in the previous section). 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 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.

4.4 Optional Aside: More Patterns

Once threads have performed independent concurrent computations, possibly on some portion of decomposed data such as a sum, it is quite common to then *reduce* those individual computations into one value. This **reduction** pattern is one of a group of patterns called **collective communication** patterns because the threads must somehow work together to create the final desired single value.

CHAPTER

FIVE

INTEGRATION USING THE TRAPEZOIDAL RULE

Your instructors will have an introduction to this chapter before you try it.

5.1 DO THIS: Observe the code

Now we will go to this directory:

```
cd ../../integration
```

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-omp-notworking.c, that attempts to do this:

```
1
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 = 3.14159265358979323846264338327 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).

5.2 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-omp-working.c:

```
1
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 = 3.14159265358979323846264338327 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.

5.3 DO THIS: Build and run each one

To create two programs, one from each code file, again type:

```
make
```

Then run each one to compare the results:

```
./trap-notworking 4
./trap-working 4
```

As with other examples you have done so far, the command-line argument of 4 above indicates use four threads.

5.4 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.