# Programming 4 – Threads

### Part 1 – The Thread

Write a program that creates a thread using *default attributes*. Your thread function will take a single integer argument, *n* the number of seconds to sleep. After sleeping for *n* seconds, the thread will print “Hello from *n* seconds in the past”. You will create three threads corresponding to 1, 2, and 3 seconds. Finally, the main thread will wait for each of the threads to finish and display the total elapsed time (in microseconds ) that it took the threads to finish.

You cannot use *global variables* for this program, you will get a zero.

For full credit you need to use proper functional decomposition – this implies the following functions:

* The thread function
* A function to launch the threads
* A function to wait for the threads to finish
* A function to get the micro-seconds elapsed between a start/stop interval
* A function to “click” the stop-watch

$ ./part1

Hello from 1 seconds in the past

Hello from 2 seconds in the past

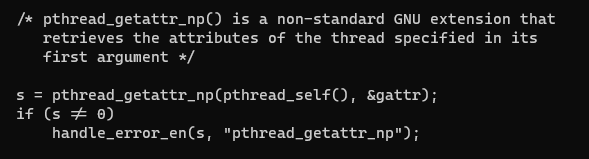
Hello from 3 seconds in the past

It took 3000283 usecs

Helpful tip: It is appropriate to use functions, watch out for *race conditions*! So, when you pass arguments to the sleeper function you’ll need to remember what the *lifetime* of a variable is – you have to pass a pointer, but if the function that “owns” the original value goes out of scope, the value of that pointer can change! The trick? Keep the variable that owns the pointer somewhere that has lifetime longer than the threads do.

## Part 2 – Thread Attributes

Thread attributes are used to configure the thread’s properties. Read the pthread\_attr\_init man page and use it for inspiration. Your job is to create a thread function that will display its stack size. The thread function should use the GNU extension described in the manual page to get its own attributes. Everything you need to know for this in the man page. Pay attention to the man page for this function too – it needs a #define before the #include to “turn on” this extract function.



Next, create a thread invoking your thread function using default arguments, wait for it to finish. Then, create a user-level thread with a 128KB stack and call your thread function, and wait for it to finish. You can let the thread library find its own memory.

Use your stop-watch function in the previous step to time the difference between creating a user and kernel thread.

$ ./part2

My stack is XXXXXXX

Default thread XXXX usecs

My stack is XXXXXX

Small stack thread XXXXX usecs

## Part 3 – Threads Working Together

This exercise will explore using threads to accelerate computations, in this case, points in Euclidean 3D space. You will use threads to accelerate the process. Ultimately, you will need to create an “answer” struct that contains:

typedef struct {

int num\_points; // number of points

point\_t \*points; // an array of points in 3-space (x,y,z)

double \*distances; // a num\_points^2 array of the distances between points

double average\_distance; // the average distance between all points

point\_t \*centroid; // the center of all the points

} answers\_t;

### Instructions

Start with a non-threaded version of the program. Follow these basic steps:

1. Get the number of points from the command line: strtod(argv[1],0), print an error and exit if the command line does not contain the right number of arguments.
2. Create a “compute” function that accepts the number of points and will return an answers struct.
3. Create the array of points using *malloc* or *calloc*
4. Initialize the random number generator. This is important: before generating your first random number call the function srand48(0) to seed the random number generator. This will intentionally bork the random process – it will always generate the same sequence of numbers. If you run your program 100 times, you’ll get the same answers every time!
5. Fill the points array with random numbers from drand48(); which will give *double* values in the range [0..1).; again, because we borked the seed, these won’t truly be random, but it is helpful for debugging.
6. Create the *distances* array using *malloc* or *calloc*. If there are *n* points, there will be distances. This actually represents a 2D array, so that array coordinates will be found at .
7. Fill the *distances* array with the distance between each point.
8. Next, compute the *average distance* using
9. Next, compute the *centroid* where ,
10. Fill the *answers* struct above, and return that value.
11. Add your stop-watch functions from parts 1 / 2.
12. Display the results:

./part3 5500

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 1283280 usecs

For the record, yes, it took 1.2 seconds to compute 5500 points on an overclocked Ryzen 3 5800. That gets us to the real “meat” of the assignment.

Before proceeding, it will be really really really really if you take a moment and use *functional decomposition*. Since this has been a struggle for many of you, I will share the names of the functions in my solution and the approximate complexity of each function. You do not have to do it my way – but I am desperately trying to get you to make your coding experience more productive and efficient, and it will pay dividends with the next step.

* *void init\_point(point\_t \*points) -* initialize a single point using drand48(), 3 lines of code.
* *void init\_all\_points(point\_t \*points, int num\_points) -* initialize all points, 3 lines of code.
* *point\_t \*create\_points(int num\_points)*create the points array, seed the random number generator, and initialize all the points, 4 lines of code.
* *double \*create\_distances(int num\_points)* – create the distances array, 2 lines of code
* *double distance(const point\_t \*a, const point\_t \*b)* – return the Euclidean distance between two points, 5 lines of code.
* *double \*compute\_all\_distances(const point\_t \*points, int num\_points)* – create the distance array, compute all the distances, and return it, 5 lines of code.
* *double compute\_average(const double \*dists, int num\_points)* – compute the average of all of the distances in the array, 4 lines of code.
* *point\_t \*compute\_centroid(const point\_t \*points, int num\_points*) – compute the (x,y,z) centroid, allocate a point from the heap, and store the x,y,z value in the point and return, 10 lines of code
* *answers\_t \*compute(int num\_points)* – Perform all of the above calculations, 11 lines of code.
* *void tick(struct timeval \*start)* – save the time of day into start, 1 line of code.
* *long tock(struct timeval start)* – get the current time and compute the usecs difference between start and now, 5 lines of code.
* *int main(int argc, char \*\*argv)* – get the command line argument for num points, get the start time, call compute, get the end-time, print the results, free all of the allocated junk, 15 lines of code.

### Threads!

There are several logical places to start multi-threading this program. The places I would start are anywhere we have loops: initializing all points, computing all distances, computing the average, and computing the centroid.

The logical place to start is to determine which parts of the program take the longest. Do you remember when I asked you about “profiling” a program? Did you wonder why? Its for today, right here, right now. I added the “-pg” argument to cflags, reran the program, and found:

|  |  |
| --- | --- |
| Function | % time |
| Compute All Distances | 76% |
| Compute Average | 29% |
| Initialize All Points | 0% |
| Computing Centroid | 0% |

So, the only *speed-up* I would expect is from the compute all distances and compute average. So, lets make this happen. How do we take a *for loop* and make it parallel? It *depends*. Sorry, that’s one of the worst bits of droll humor I’ve put into an assignment in years. We have to look for any *loop dependencies* that would prevent one iteration of the loop from happening before another one. If there are dependences, then we have to wait for synchronization methods in the next chapter.

In this case, we *should* have for loop were every computation is done independently of the rest. In fact, if we had an infinite number of processors, we could do all of the loop statements at once! This is *embarrallel*.

You probably won’t read this far until after you’ve written your code, so I’ll share my function here…

double \* compute\_all\_distances(const point\_t \*points, int num\_points)

{

double \*dists = create\_distances(num\_points);

for (int r = 0; r < num\_points; r++)

{

for (int c= 0; c < num\_points; c++)

{

dists[r \* num\_points + c] = distance(&points[r], &points[c]);

}

}

return dists;

}

In theory, we could:

1. Create a new thread for every single pair of points – this would require threads.
2. Create a new thread for every row – each thread is responsible for all of the columns in its row – this would require *n* threads
3. Create a new thread for clumps of rows – e.g. with 8 processors, we could have each thread do *n/8* rows, to balance the workload between them
4. Create *p* threads, where each *p* is responsible for a “patch” of the distances – a small grid of the larger 2D array – this way, each processor has localized access to the data and we should increase the cache hit rates on each processor and reduce hardware contention. However, the *size* of the patch is hardware dependent.

There are certainly other approaches we could do, but these are just a few. The *easiest* one in my mind is the *#3, create a new thread for clumps of rows*.

### Instructions

Modify your compute function so that it still creates the distance array, but creates *p* threads (a parameter which is the number of threads to use), where each thread is responsible for some chunk of rows, for example, with 100 rows and p=1, the thread is responsible for rows 0…99. With 100 rows and p=2, one thread handles rows 0..49, and the other 50..99.

Remember, you can only pass one argument to a thread function, so you’ll want to create a struct that holds all of the information each thread needs to get started.

Congratulations on reading all the instructions first. I looked at what the function needed for its computation, and then made a struct:

typedef struct {

const point\_t \*points;

int num\_points;

double \*dists;

int start\_row;

int end\_row;

} thrd\_dist\_args\_t;

So, once you have all of the information the thread function needs in a single struct, you can write the thread function:

void \*compute\_distance\_thread(void \*rawargs)

{

thrd\_dist\_args\_t \*args = (thrd\_dist\_args\_t \*)rawargs;

for (int r = args->start\_row; r < args->end\_row; r++)

{

for (int c = 0; c < args->num\_points; c++)

{

args->dists[r \* args->num\_points + c] = distance(&args->points[r], &args->points[c]);

}

}

}

So, this new function looks a lot like the original, but its been setup for threads work. Next, to minimize the impact on the rest of your program, we go back to the original “compute\_all\_distances”, and add a parameter to the function:

double \* compute\_all\_distances(const point\_t \*points, int num\_points, ***int num\_threads***)

Modify this function now. You need to:

* Create an array of *num­\_threads* thread arguments
* Create an array of *num\_threads* pthread\_t
* Create and initialize a set of pthread attributes
* Create a for loop that sets the thread arguments
* Create a for loop that creates the threads (giving the thread argument)
* Create a for loop that does the pthread\_join on each thread.

### Number of Threads

The number of threads and the number of points will be passed as a command line arguments to your program.

### Results

You should see results similar to the following (your computer’s hardware will play a huge role here). Things to watch out for: If your time isn’t going down, then make sure you are dividing up the work correctly. The first time I ran this, my times were all the same. Turns out I started every work unit at 0. The other thing is that the average is very sensitive to small changes in results. I was getting averages that were close but not the same for some thread numbers – it was an integer division / truncation thing, fixed that by making the last work unit have the last rows. Finally, I was still getting a rounding error, and it was because I was ending one working on row *r*, and then starting the next work unit on the same row *r*.

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 1319296 usecs using 1 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 706717 usecs using 2 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 479169 usecs using 4 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 327905 usecs using 6 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 284594 usecs using 8 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 270783 usecs using 10 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 243001 usecs using 12 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 231168 usecs using 14 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 231663 usecs using 16 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 256972 usecs using 18 threads

Centroid: (0.504,0.498,0.499), Average: (0.661)

It took 234410 usecs using 20 threads

So, in the end, we took this computation down to 234,410 microseconds, for a speed up of 5.6281! But, wait, that’s not the *linear* speed up we hoped for!!!