# Computing Lab #7: OpenMP parallelizations

**Due**: Tuesday, November 7, 2017 (midnight)

**Programming language:** Write, compile, and run a Fortran, C or C++ program. Your choice! This lab assumes the C programming.

**Points**: 10

In this lab you will parallelize a 1D trapezoidal integration algorithm with OpenMP. Here is the code to start with:

```c
#include <stdio.h>
#include <stdlib.h>

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

  int     N = atoi( argv[1] );
  double dx = 2./(N-1);
  double integral = 0.0;
  const double pi = 3.141592653589793;
  double x, f;

  for (int i=0;i<N;++i) {
    x = -1. + dx*i; // compute x_i
    f = 1. / (1. + x*x);  // compute f_i = f(x_i)
    integral += dx*f;
  }

  // end points of trapezoidal rule treated specially
  x = -1.0;
  f = 1. / (1. + x*x); // compute f_i = f(x_i)
  integral -= 0.5*dx*f;
  x = 1.0;
  f = 1. / (1. + x*x); // compute f_i = f(x_i)
  integral -= 0.5*dx*f;

  //printf("integral = %f , error = %e\n",integral, integral - pi / 2.);
  printf("%i %f %e\n",N, integral, integral - pi / 2.);

  return 0;
}
```

Before continuing: compile and run this code. Make sure it works.

## I. TASK #1: SIMPLE PARALLELIZATION (4 POINTS)

**Task:** Parallelize the 1D trapezoidal code with a simple strategy.

- **Part a** Parallelize the for-loop. Start by copying the serial code to a new file. Call it trap_omp1.c
    1. Add a line of code to use 4 threads.
    2. Parallelize the for-loop by using the simplest (and wrong) way:

```c
 #pragma omp parallel for
 for (int i=0;i<N;++i) {
   x = -1. + dx*i; // compute x_i
   f = 1. / (1. + x*x);  // compute f_i = f(x_i)
   printf("thread %i, x =%f, f=%f\n",omp_get_thread_num(),x,f); // for diagnostics
   integral += dx*f;
 }
```

3. Compile your code. From here and forever after, you should use a gcc flag to display all warnings: OpenMP is notorious for "silent bugs" which you can catch with the -Wall flag (= warn all)

```
>>> gcc -Wall -std=c99 -fopenmp -o trap_omp1 trap_omp1.c
```

4. Run your code and compare to the serial code... the results should be wrong! They will also give different results each time you run the code.
5. Fix your OpenMP code by making x and f private (to avoid race conditions). Use the private directive we discussed in class.
6. Run your code and compare to the serial code... the results should be wrong!
7. Fix your OpenMP code by making the statement "integral += dx*f;" critical (to prevent multiple threads from accessing it at the same time).
8. Run your code and compare to the serial code... the results should now be the same.

## II. TASK #2: BETTER PARALLELIZATION (4 POINTS)

**Task:** Parallelize the 1D trapezoidal code with a simple strategy.

- **Part a** Parallelize the for-loop. Start by copying the OpenMP code you just wrote to a new file. Call it trap_omp2.c

1. Having all threads use a shared variable "integral" requires coordination: some threads must wait which is wasteful. Lets now use an array for each thread to use. your code should look roughly like

```
double integral_thread[4] = {0}; // initialize all to zero

#pragma omp parallel for private(x,f)
for (int i=0;i<N;++i) {
  x = -1. + dx*i; // compute x_i
  f = 1. / (1. + x*x);  // compute f_i = f(x_i)
  int id = omp_get_thread_num();
  integral_thread[id] += dx*f;
}

double integral = 0;
for(int i=0; i< 4; i++) {
  integral = integral + integral_thread[i];
}

// More code here
```

2. Run your code and compare to the serial code... the results should now be the same.

## III. TASK #3: WHATS FASTER? (2 POINTS)

Run the serial code and both OpenMP parallelizations using lots of points (a large value of N). Whats fastest? Notice that the cputime is probably higher for OpenMP: There is overhead when creating, destroying, and managing threads.

You may be surprised that the two OpenMP parallelizations give very different times; although this isn't so surprising when you recall that critical regions maintain a thread queue which has a lot of overhead. The second parallelization used a trick which might be best called a *reduction* – we'll come back to this later.

That the serial version is probably fastest might be surprising. For this problem we would need to work harder to write a competitive parallel code and/or experiment with a better choice of number of threads.

## IV. UPLOAD YOUR WORK TO GIT

Congratulations! You've completed the lab. Now lets upload your work to git.

Follow the same series of steps as you've done for the last few labs. Place all work for this lab (task 1: OpenMP code, task 2: OpenMP code, task 3: add a very brief description of your timing experiment to a text file.) into a folder called "Lab7" and push it to bitbucket.