# Homework 1

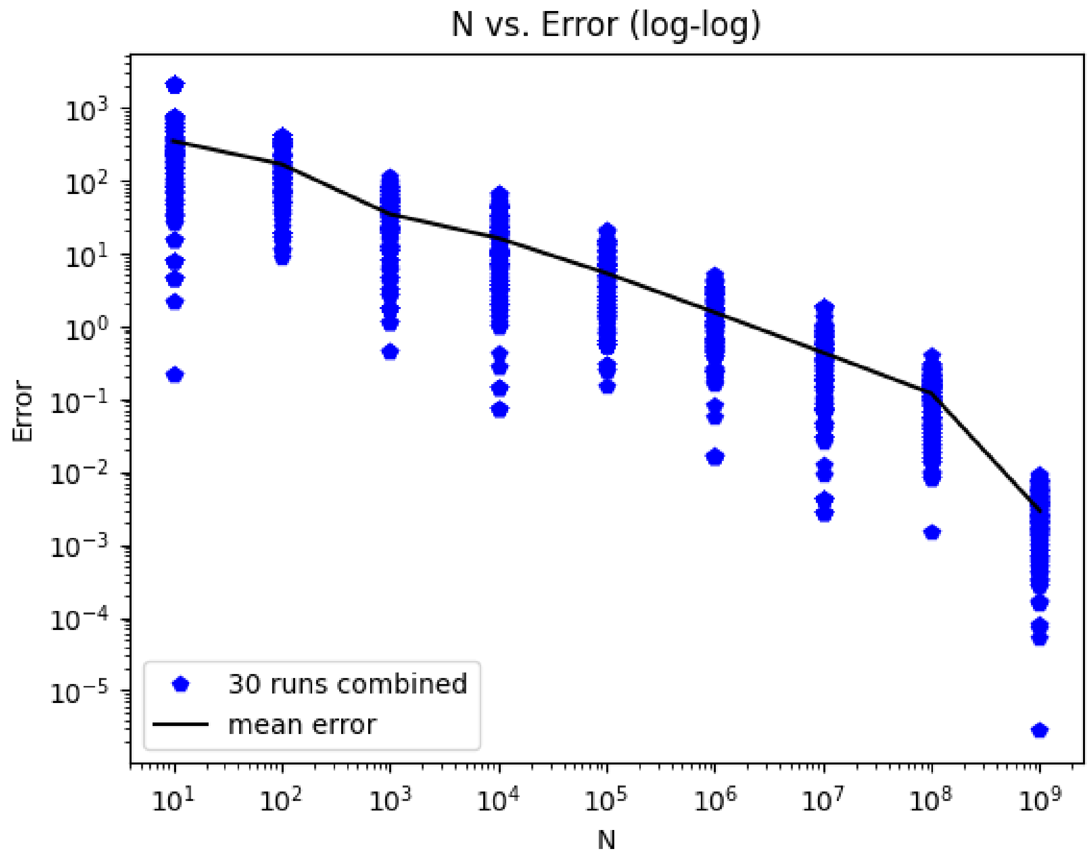
Krut Patel

November 6,2023

## Problem 1

### (a)

We've used the code provided by you for serial 10-D MonteCarlo integration. As seen in previous homework, the plot below show the N values vs mean error for 1000 runs combined. It follows a lone with a slope of approximatly -0.5. As this is a log-log plot, it tells us that the error decreases with rate of $N^{-1/2}$.



## Problem 2

### (a) Code Parallelization

```
omp_set_num_threads(15);
double local_integral = 0.0;

#pragma omp parallel private(x) reduction(+:local_integral)
```

```
  {
    unsigned int seed = (unsigned int)(time(NULL) + omp_get_thread_num());
    // Different seed for each thread

    #pragma omp for
    for(long long int i=1; i<=N; ++i) {

      for(int j=0;j<dim;++j) {
        x[j] = sample_interval(xL,xR,&seed);
      }

      double f_i = func(x,dim);
      local_integral += f_i;
    }
  }

  integral = (V/N)*local_integral;
```

The code block provided above is where we parallelize the for loop. The way seed for random values is given makes sure that each thread makes independent random draws. The code still converges to $2^{10}$ with the same rate as the serial code.

The evidence for multiple threads being used can be found in the data files commited to the repository for problem 3. The first column in the data files is the number of threads being used for each run.

There were a few things that needed to be done in order to get the a good speedup value. A local integral is defined, and inside the for loop, that local integral is calculated by each thread independently. x is also kept private. Also, without the reduction clause, the local integral leads to data races and takes more time to complete the loop then serial code.

### (b)

When we compared the timing values from OMP timer with the Unix program time, we found out that the Unix program time would give a bigger time value then the OMP timer by a factor of $10^{-2}$. It must be due to the fact that Unix program time is an independent program which monitors the executable, and calculates the time. Instead of rand, rand_r is used to generate the random numbers as rand is not thread safe.

## Problem 3

### (a) - Rosenbrock function implementation and proof checking

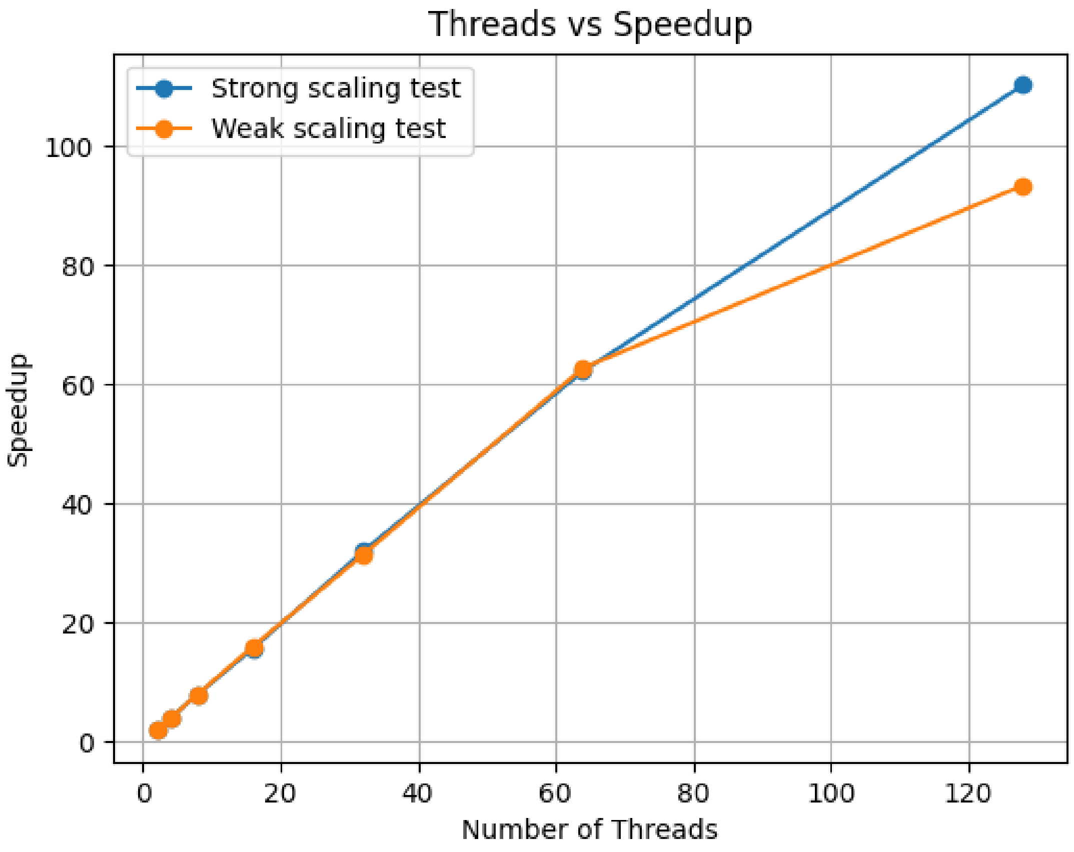
```
#include <stdio.h>
#include <math.h>

double rosenbrock_function(double *x, int dim) {
    double f = 0.0;
    for (int i = 0; i < dim - 1; ++i) {
        f += 100 * pow((x[i + 1] - x[i] * x[i]), 2) + pow(1 - x[i], 2);
    }
    return exp(-f); //Taking the exponential of the negative of the Rosenbrock function
}

int main() {
    const int dim = 10;
    double x[dim];

    //Setting all values of x to 1
    for (int j = 0; j < dim; ++j) {
        x[j] = 1.0;
    }

    //Rosenbrock function for x = {1, 1, ..., 1}
    double integral = rosenbrock_function(x, dim);

    printf("Rosenbrock function value for x = {1, 1, ..., 1}: %1.5e\n", integral);

    return 0;
}
```

The script provided above is the used to test the Rosenbrock function. To check if our code produces correct values, we've set the value of x for all 10 dimensions to be 1. The final integral of the function will give 1 as a result, which this code produces.

Our actual script for Rosenbrock function uses the same methodology, thus confirms that the code should produces correct results.

### (b)

The choices that had to be made in order to get the most speedup is explained at the end of Problem 2 - (a).



The figure above shows the comparison between the speedups of strong scaling test and weak scaling test. The speedup for both the test for the number of threads up to 64 is almost perfect. For 128 threads, weak scaling test shows relatively low speedup. This can be due to the sample size taken. For weak scaling test, N values increases with respect to the number of threads. So, for 128 threads, the sample size for strong scaling test was $10^8$ and for weak scaling test it was $10^{10}$.

## (c)

Our code has the lower and upper limit of the integral set to -1 to 1. I've calculated the the integral for a very large values of N (up to $10^{11}$). Here are the data points from the run.

| Threads | $N$ | Integral | Time (s) |
|---|---|---|---|
| 1 | 1000000000 | 7.33007e-11 | 234.929838 |
| 2 | 2000000000 | 3.74721e-10 | 234.958685 |
| 4 | 4000000000 | 3.63722e-10 | 235.050498 |
| 8 | 8000000000 | 3.74728e-10 | 231.253866 |
| 16 | 16000000000 | 3.74165e-10 | 241.213777 |
| 32 | 32000000000 | 3.69016e-10 | 385.908926 |
| 64 | 64000000000 | 3.69782e-10 | 241.184101 |
| 128 | 128000000000 | 3.73768e-10 | 268.839921 |

We can clearly see that after N $= 2 * 10^9$, the integral is converging to $3.7 * 10^{-10}$, which seems to be the integral's value. We're getting accurate results upto 1 digit of accuracy.

Submitted to - Professor Scott Field