Code

1. Serial Execution (PA3.cpp)

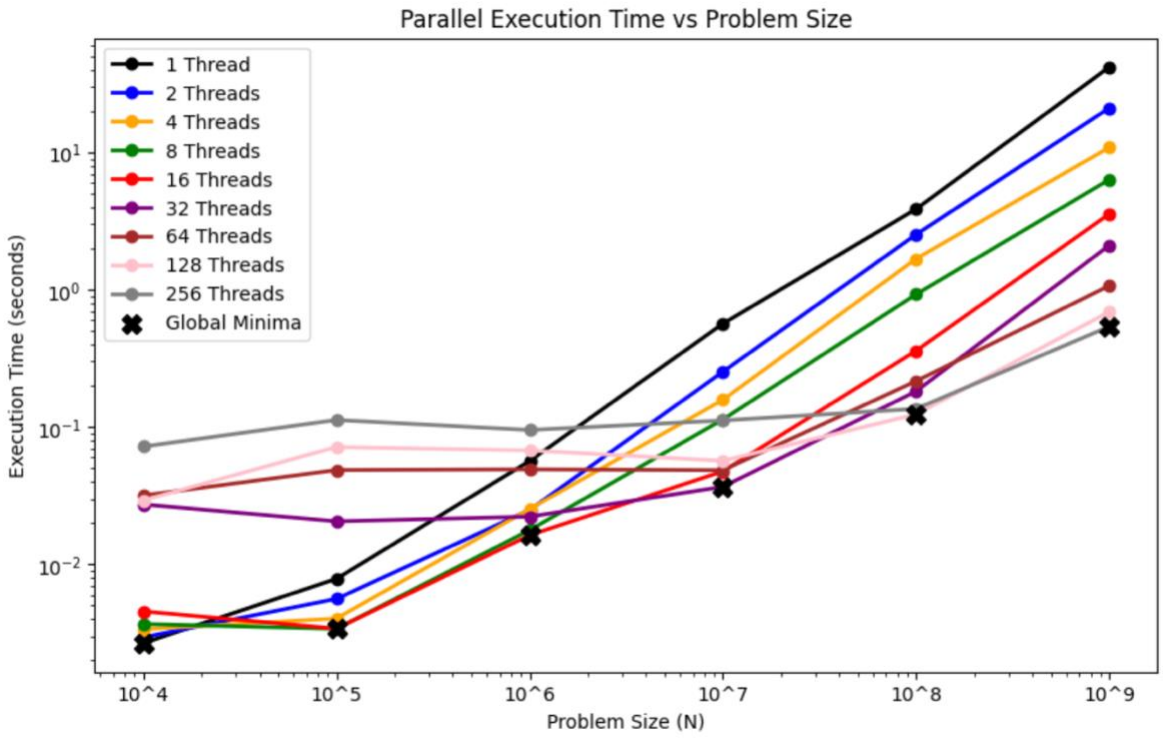
```
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;
double function(double x){
  return acos(cos(x)/(1+2*cos(x)));
}
double findIntegral(int N, double b, double a){
    double h = (b - a) / N;
    double sum = function(a) + function(b);
    double oddSum = 0.0, evenSum = 0.0;

    for (int i = 1; i < N; i += 2) { // Odd indices
        double x = a + i * h;
        oddSum += function(x);
    }

    for (int i = 2; i < N; i += 2) { // Even indices
        double x = a + i * h;
        evenSum += function(x);
    }

    return (h / 3) * (sum + 4 * oddSum + 2 * evenSum);
}

int main(int argc, char* argv[]){
  int N = stoi(argv[1]);
  double b = M_PI/2;
  double a = 0;
  auto start = chrono::high_resolution_clock::now();
  double approx = findIntegral(N,b,a);
  cout<< "Approc " << approx << endl;
  double result = (5*(M_PI)*(M_PI)) / 24;
  double diff = abs(result - approx);
  cout << " Diff " << diff << endl;
  auto end = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed = end - start;
  cout << "Serial Execution Time for N=" << N << " " << elapsed.count() << " sec\n";
  return 0;
          }
```

2. Parallel Execution (PA3_parallel.cpp)

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <vector>
#include <algorithm>
#include <omp.h>
#include <cmath>
using namespace std;

double function(double x){
        return acos(cos(x)/(1+2*cos(x)));
}

double simpsons_method_omp(double a, double b, int N,int numThreads)
{
   double h = (b - a) / N;
   double sum = function(a) + function(b);

   double oddSum = 0.0, evenSum = 0.0;

   #pragma omp parallel for num_threads(numThreads) reduction(+:oddSum)
   for (int i = 1; i < N; i += 2) { // Odd indices
      double x = a + i * h;
      oddSum += function(x);
   }

   #pragma omp parallel for num_threads(numThreads) reduction(+:evenSum)
   for (int i = 2; i < N; i += 2) { // Even indices
      double x = a + i * h;
      evenSum += function(x);
   }

   return (h / 3) * (sum + 4 * oddSum + 2 * evenSum);
}


int main(int argc, char* argv[]){
  int N = stoi(argv[1]);
  int num_threads = stoi(argv[2]);
  double b = M_PI/2;
  double a = 0;
  auto start = chrono::high_resolution_clock::now();
  double approx = simpsons_method_omp(a,b,N,num_threads);
```

```
  cout<< "Approximate " << approx << endl;
  double result = (5*(M_PI)*(M_PI)) / 24;
  double diff = abs(result - approx);
  cout << " Diff " << diff << endl;
  auto end = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed = end - start;

    cout << "Optimized Parallel Execution Time for N=" << N << " with " << num_threads
         << " threads: " << elapsed.count() << " sec\n";

    return 0;
}
```

PLOT (Partition vs Run Time)



DISCUSSION

As evidenced in graph with the Global Minima symbol, different number of partitions have different optimal number of threads.

Here is a table representing that –

Partitions (N)	Optimal Time	Optimal Thread Count (n)
10^4	0.00266876	1
10^5	0.00339317	8
10^6	0.0163029	16
10^7	0.0367181	32
10^8	0.123891	128
10^9	0.539737	256

From the table, we can deduce as for power of 10 magnitude increase, the optimal Thread count increases by a magnitude of at least 2. As we move on to bigger partitions, 10^10 and further, it is safe to assume that 256 (highest thread number) will be the optimal thread count. Serial calculation is universally slower than all other parallel execution threads by 10^7 partitions.

Accuracy – All my executions with all combinations of N*n (partitions * threadCount) my error is <= 1e-14. After dumping my execution results I visualized this table to better understand the accuracy of threadCount and Partition combinations.

Partitions (N)	Max Error
10^4	1.33227e-15
10^5	5.77316e-15
10^6	3.4639e-14
10^7	4.4639e-14
10^8	4.30767e-14
10^9	4.35207e-14

As we can see, max error for each partition (N) is within the defined threshold of 10^-14.