# Gowrish I 2022BCS0155

# LAB 1

### Write OpenMP program in C to parallelize bubble sort

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <stdbool.h>  
#include <time.h>  
#include <omp.h>  
  
// #define LOG(x) printf("%d:[%s = %d]\n", \_\_LINE\_\_, #x, (x))  
#define LOG(x)  
  
#define thread\_no omp\_get\_thread\_num  
  
#define ARR\_SIZE 30000  
#define MAX\_THREADS omp\_get\_max\_threads()  
  
void swap(int\* a, int\* b) {  
 int temp = \*a;  
 \*a = \*b;  
 \*b = temp;  
}  
  
void sequential\_bubble\_sort(int\* arr, int size) {  
 bool sorted = false;  
 while (!sorted) {  
 sorted = true;  
 for (int j = 0; j < size - 1; j++) {  
 if (arr[j] > arr[j + 1]) {  
 swap(arr + j, arr + j + 1);  
 sorted = false;  
 }  
 }  
 }  
}  
  
// Merge two sorted halves of an array  
void merge\_two\_sorted(int\* arr1, int\* arr2, int s1, int s2) {  
 int\* temp = (int\*)malloc((s1 + s2) \* sizeof(int));  
 int i = 0, j = 0, k = 0;  
  
 while (i < s1 && j < s2) {  
 if (arr1[i] <= arr2[j]) {  
 temp[k++] = arr1[i++];  
 } else {  
 temp[k++] = arr2[j++];  
 }  
 }  
  
 while (i < s1) temp[k++] = arr1[i++];  
 while (j < s2) temp[k++] = arr2[j++];  
  
 memcpy(arr1, temp, (s1 + s2) \* sizeof(int));  
 free(temp);  
}  
  
void parallel\_bubble\_sort(int\* arr, int size) {  
 int start, end;  
 int chunk\_size;  
 int remaining;  
  
 #pragma omp parallel shared(arr, size, remaining) private(start, end, chunk\_size)  
 {  
 chunk\_size = (size / MAX\_THREADS);  
 remaining = (size % MAX\_THREADS);  
 start = thread\_no() \* chunk\_size;  
  
 if (thread\_no() == MAX\_THREADS - 1) {  
 chunk\_size += remaining;  
 }  
  
 end = start + chunk\_size - 1;  
  
 // printf("Thread ID %d\t: %d, [%d -- %d] \n", thread\_no(), chunk\_size, start, end);  
 sequential\_bubble\_sort(arr + start, chunk\_size);  
 }  
  
 // printf("---\n");  
  
 // Merging sorted sections in pairs  
 int thread\_count = MAX\_THREADS / 2;  
  
 while (thread\_count > 0) {  
 #pragma omp parallel shared(arr, size, remaining) private(start, chunk\_size)  
 {  
 int thread\_id = thread\_no();  
 if (thread\_id < thread\_count) {  
 chunk\_size = (size / MAX\_THREADS);  
 remaining = (size % MAX\_THREADS);  
  
 start = thread\_id \* 2 \* chunk\_size;  
  
 if (thread\_id == thread\_count - 1) {  
 chunk\_size += remaining;  
 }  
  
 // printf("Merging by Thread ID %d: [%d -- %d] with [%d -- %d]\n", thread\_id, start, start + chunk\_size - 1, start + chunk\_size, start + 2 \* chunk\_size - 1);  
  
 merge\_two\_sorted(arr + start, arr + start + chunk\_size, chunk\_size, chunk\_size);  
 }  
 }  
  
 thread\_count /= 2;  
 }  
}  
  
int main(int argc, char\*\* argv) {  
 srand(time(NULL));  
 int\* arr = malloc(sizeof(int) \* ARR\_SIZE);  
 for (int i = 0; i < ARR\_SIZE; i++) {  
 arr[i] = rand() % 1000;  
 }  
  
#ifdef PARALLEL  
 parallel\_bubble\_sort(arr, ARR\_SIZE);  
#else  
 sequential\_bubble\_sort(arr, ARR\_SIZE);  
#endif  
 free(arr);  
 return 0;  
}

To compile for serial bubble sort

gcc -o main main.c -lm -fopenmp  
time ./main # To measure the time of program execution

real 0m2.600s  
user 0m2.599s  
sys 0m0.000s

To compile for parallel bubble sort

gcc -o main main.c -lm -fopenmp -DPARALLEL  
time ./main # To measure the time of program execution

real 0m0.045s  
user 0m0.506s  
sys 0m0.001s

#### Results

| Execution | Time in ms |
| --- | --- |
| Sequential | 2.600 |
| Parallel | 0.045 |

# LAB 2

### Write Bubble sort in sequential and parallel mode

#include <iostream>  
#include <vector>  
#include <chrono>  
#include <cstdio>  
#include <algorithm>  
#include <omp.h>  
  
class Timer {  
 public:  
 void reset();  
 double elapsed() const;  
 void end();  
  
 private:  
 using Clock = std::chrono::high\_resolution\_clock;  
 using Second = std::chrono::duration<double, std::ratio<1> >;  
 std::chrono::time\_point<Clock> curr{ Clock::now() };  
};  
  
void Timer::reset() {  
 curr = Clock::now();  
}  
  
double Timer::elapsed() const {  
 return std::chrono::duration\_cast<Second>(Clock::now() - curr).count();  
}  
  
void Timer::end() {  
 std::cout << "Took " << this->elapsed() << "s" << std::endl;  
 this->reset();  
}  
  
void print\_vector(std::vector<int> arr)  
{  
 for (int x : arr)  
 {  
 std::cout << x << ", ";  
 }  
  
 std::cout << std::endl;  
}  
  
std::vector<int> seq\_bubble\_sort(std::vector<int> arr)  
{  
 while (true)  
 {  
 bool sorted = true;  
 for (int i = 0; i < arr.size() - 1; i++)  
 {  
 bool inOrder = (arr[i] <= arr[i + 1]);  
 if (!inOrder)  
 {  
 int temp = arr[i];  
 arr[i] = arr[i + 1];  
 arr[i + 1] = temp;  
 }  
 sorted &= inOrder;  
 }  
 if (sorted) break;  
 }  
 return arr;  
}  
  
std::vector<int> par\_bubble\_sort(std::vector<int> arr) {  
 bool sorted = false;  
 while (!sorted) {  
 sorted = true;  
  
 // Even phase  
 #pragma omp parallel for reduction(&:sorted)  
 for (int i = 0; i < arr.size() - 1; i += 2) {  
 if (arr[i] > arr[i + 1]) {  
 std::swap(arr[i], arr[i + 1]);  
 sorted = false;  
 }  
 }  
  
 // Odd phase  
 #pragma omp parallel for reduction(&:sorted)  
 for (int i = 1; i < arr.size() - 1; i += 2) {  
 if (arr[i] > arr[i + 1]) {  
 std::swap(arr[i], arr[i + 1]);  
 sorted = false;  
 }  
 }  
 }  
 return arr;  
}  
  
int main()  
{  
 Timer timer;  
 std::vector<int> sizes = {10, 100, 1000, 10000, 50000};  
  
 std::cout << "Using " << omp\_get\_max\_threads() << " Threads" << std::endl;  
  
 for (int i = 0; i < sizes.size(); i++) {  
 int size = sizes[i];  
 std::vector<int> arr(size);  
 for (int i = 0; i < size; i++) {  
 arr[i] = rand() % 100;  
 }  
  
 timer.reset();  
  
 std::vector<int> sorted\_arr = seq\_bubble\_sort(arr);  
 auto t1 = timer.elapsed();  
  
 timer.reset();  
  
 std::vector<int> another\_sorted\_arr = par\_bubble\_sort(arr);  
  
 auto t2 = timer.elapsed();  
  
 printf("[SIZE: %d]\t", size);  
 std::cout << t1 << "\t" << t2 << std::endl;  
 }  
}

#### Output

Using 12 Threads  
[SIZE: 10] 1.412e-06 0.000947931  
[SIZE: 100] 8.7889e-05 0.000698251  
[SIZE: 1000] 0.00771212 0.00669482  
[SIZE: 10000] 0.739259 0.134833  
[SIZE: 50000] 19.1999 5.33304

#### Benchmark

| Size of Array | Sequential in seconds | Parallel in seconds |
| --- | --- | --- |
| 10 | 1.453e-06 | 0.000473912 |
| 100 | 0.000108827 | 0.00086741 |
| 1000 | 0.00776502 | 0.00767423 |
| 10000 | 0.736298 | 0.120619 |
| 50000 | 20.6958 | 3.90075 |

### Calculate Value of Pi

#include <iostream>  
#include <functional>  
#include <cassert>  
#include <chrono>  
#include <cmath>  
  
#include <omp.h>  
  
class Timer {  
 public:  
 void reset() {  
 curr = Clock::now();  
 }  
 double elapsed() const {  
 return std::chrono::duration\_cast<Second>(Clock::now() - curr).count();  
 }  
  
 private:  
 using Clock = std::chrono::high\_resolution\_clock;  
 using Second = std::chrono::duration<double, std::ratio<1> >;  
 std::chrono::time\_point<Clock> curr{ Clock::now() };  
};  
  
double area\_func(double x) {  
 return 4.0 / (1 + x \* x);  
}  
  
double integrate(const std::function<double(double)>& generator, double lower\_limit, double higher\_limit, double dx) {  
 assert(lower\_limit <= higher\_limit);  
  
 double value = 0;  
 double size = higher\_limit - lower\_limit;  
 double rect\_count = (int) (size / dx);  
 for (int i = 0; i < rect\_count; i++) {  
 double x = lower\_limit + i \* dx;  
 value += generator(x) \* dx;  
 }  
  
 return value;  
}  
  
double parallel\_integrate(const std::function<double(double)>& generator, double lower\_limit, double higher\_limit, double dx) {  
 assert(lower\_limit <= higher\_limit);  
  
 double value = 0;  
 double size = higher\_limit - lower\_limit;  
 int rect\_count = (int) (size / dx);  
  
 #pragma omp parallel for reduction(+:value)  
 for (int i = 0; i < rect\_count; i++) {  
 double x = lower\_limit + i \* dx;  
 value += generator(x) \* dx;  
 }  
  
 return value;  
}  
  
int main() {  
 Timer timer;  
 std::vector<double> dx\_values = {1e0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8};  
  
 std::cout << "Using " << omp\_get\_max\_threads() << " Threads" << std::endl;  
  
 for (int i = 0; i < dx\_values.size(); i++) {  
 double dx = dx\_values[i];  
  
 timer.reset();  
 double pi\_seq = integrate(area\_func, 0.0, 1.0, dx);  
 auto t1 = timer.elapsed();  
  
 timer.reset();  
 double pi\_par = parallel\_integrate(area\_func, 0.0, 1.0, dx);  
 auto t2 = timer.elapsed();  
  
 std::cout << dx << "\t" << pi\_seq << "\t" << t1 << "\t" << pi\_par << "\t" << t1 << std::endl;  
 }  
}

#### Output

Using 12 Threads  
1 4 3.61e-07 4 0.000721069  
0.1 3.23993 8.01e-07 3.23993 1.7284e-05  
0.01 3.15158 4.148e-06 3.15158 1.3746e-05  
0.001 3.14259 4.1681e-05 3.14259 1.7564e-05  
0.0001 3.14169 0.000381284 3.14169 0.000129139  
1e-05 3.14158 0.00382948 3.14158 0.000351958  
1e-06 3.14159 0.0289401 3.14159 0.00541477  
1e-07 3.14159 0.27464 3.14159 0.0383328  
1e-08 3.14159 2.73214 3.14159 0.397813

#### Benchmark

| Rectangle width (dx) | PI in sequential | Time for sequential | PI in parallel | Time for parallel |
| --- | --- | --- | --- | --- |
| 1 | 4 | 3.51e-07 | 4 | 0.00216247 |
| 0.1 | 3.23993 | 5.41e-07 | 3.23993 | 0.000155264 |
| 0.01 | 3.15158 | 3.897e-06 | 3.15158 | 0.000317573 |
| 0.001 | 3.14259 | 3.8373e-05 | 3.14259 | 4.6662e-05 |
| 0.0001 | 3.14169 | 0.000381558 | 3.14169 | 8.3504e-05 |
| 1e-05 | 3.14158 | 0.00382509 | 3.14158 | 0.000352621 |
| 1e-06 | 3.14159 | 0.0290141 | 3.14159 | 0.00671049 |
| 1e-07 | 3.14159 | 0.274438 | 3.14159 | 0.00671049 |
| 1e-08 | 3.14159 | 2.74175 | 3.14159 | 0.387032 |

# LAB 3

### Program 1

Write an OpenMP program with C++ that estimates the value of pi (𝜋) using a following function and apply Simpson’s 1/3 rd rule.

#### Simpson’s Rule

The following components are to be shown

1. Write the serial version program to estimate the value of pi (𝜋). Test the result with classical integration value. Calculate the execution time by using the library function omp\_get\_wtime().

* **Code**
* #include <iostream>  
  #include <functional>  
  #include <cstdint>  
  #include <cstdio>  
  #include <string>  
  #include <omp.h>  
  #include <iomanip>  
    
  typedef std::function<double(double)> Generator;  
    
  class Timer {  
  public:  
   Timer() {  
   this->reset();  
   }  
    
   void reset() {  
   m\_Time = omp\_get\_wtime();  
   }  
    
   double elapsed() {  
   double now = omp\_get\_wtime();  
   return now - m\_Time;  
   }  
    
   void log(std::string msg) {  
   double time = this->elapsed();  
   std::cout << "[TIMER] "   
   << msg << " : "   
   << time  
   << "s" << std::endl;  
   }  
  private:  
   double m\_Time;  
  };  
    
  double simpson\_rule\_serial(double start, double end, Generator generator, uint32\_t precision)  
  {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   // First Term  
   sum += generator(x(0));  
    
   // Odd sum  
   double odd\_sum = 0.0F;  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   odd\_sum += generator(x(i));  
   }  
   sum += 4.0 \* odd\_sum;  
    
   // Even sum  
   double even\_sum = 0.0F;  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   even\_sum += generator(x(i));  
   }  
   sum += 2.0 \* even\_sum;  
    
   // Last Term  
   sum += generator(x(precision));  
    
   // Final Multiplier  
   sum \*= spacing / 3;  
    
   return sum;  
  }  
    
    
  int identify\_machine()  
  {  
   printf("Detected %d Processor(s)\n", omp\_get\_num\_procs());  
   printf("Max %d Threads(s) Available\n", omp\_get\_max\_threads());  
    
   int thread\_count = -1;  
   while (!(0 < thread\_count && thread\_count <= omp\_get\_max\_threads())) {  
   std::cout << "Enter number of threads to use: ";  
   std::cin >> thread\_count;  
   if (0 < thread\_count && thread\_count <= omp\_get\_max\_threads()) {  
   omp\_set\_num\_threads(thread\_count);  
   break;  
   } else {  
   std::cout << "Invalid thread count entered " << thread\_count << std::endl;  
   }  
   }  
   return thread\_count;  
  }  
    
  int main()  
  {  
   Timer timer;  
   auto pi\_function = [](double x) {  
   return 4 / (double)(1 + x \* x);  
   };  
    
   int thread\_count = identify\_machine();  
    
   printf("Using %d threads...\n", thread\_count);  
    
   uint32\_t precision = 1e8;  
    
   timer.reset();  
   double pi\_serial = simpson\_rule\_serial(0, 1, pi\_function, precision);  
   std::cout << "PI in serial = "   
   << std::setprecision(15)  
   << pi\_serial   
   << std::endl;  
   timer.log("Serial execution took");  
  }
* **Output**
* Detected 12 Processor(s)  
  Max 12 Threads(s) Available  
  Enter number of threads to use: 12  
  Using 12 threads...  
  PI in serial = 3.14159265358944  
  [TIMER] Serial execution took : 2.81654947599964s

1. Write the parallel version program to estimate the same. Test the result with classical integration value and by (a). It includes number of threads involved and the area calculated by which thread number. Calculate the execution time by using the library function omp\_get\_wtime().

* **Code**
* #include <iostream>  
  #include <functional>  
  #include <cstdint>  
  #include <cstdio>  
  #include <string>  
  #include <omp.h>  
  #include <iomanip>  
    
  typedef std::function<double(double)> Generator;  
    
  class Timer {  
  public:  
   Timer() {  
   this->reset();  
   }  
    
   void reset() {  
   m\_Time = omp\_get\_wtime();  
   }  
    
   double elapsed() {  
   double now = omp\_get\_wtime();  
   return now - m\_Time;  
   }  
    
   void log(std::string msg) {  
   double time = this->elapsed();  
   std::cout << "[TIMER] "   
   << msg << " : "   
   << time  
   << "s" << std::endl;  
   }  
  private:  
   double m\_Time;  
  };  
    
  double simpson\_rule\_parallel(double start, double end, Generator generator, uint32\_t precision) {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   // First Term  
   sum += generator(x(0));  
    
   // Odd sum  
   double odd\_sum = 0.0F;  
   double even\_sum = 0.0F;  
    
   #pragma omp parallel shared(precision, sum)  
   {  
   #pragma omp for reduction( + : odd\_sum )  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   odd\_sum += generator(x(i));  
   }  
    
   #pragma omp for reduction( + : even\_sum )  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   even\_sum += generator(x(i));  
   }  
   }  
    
   sum += 4.0 \* odd\_sum + 2.0 \* even\_sum;  
    
   // Last Term  
   sum += generator(x(precision));  
    
   // Final Multiplier  
   sum \*= spacing / 3;  
    
   return sum;  
  }  
    
  double simpson\_rule\_serial(double start, double end, Generator generator, uint32\_t precision)  
  {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   // First Term  
   sum += generator(x(0));  
    
   // Odd sum  
   double odd\_sum = 0.0F;  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   odd\_sum += generator(x(i));  
   }  
   sum += 4.0 \* odd\_sum;  
    
   // Even sum  
   double even\_sum = 0.0F;  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   even\_sum += generator(x(i));  
   }  
   sum += 2.0 \* even\_sum;  
    
   // Last Term  
   sum += generator(x(precision));  
    
   // Final Multiplier  
   sum \*= spacing / 3;  
    
   return sum;  
  }  
    
    
  int identify\_machine()  
  {  
   printf("Detected %d Processor(s)\n", omp\_get\_num\_procs());  
   printf("Max %d Threads(s) Available\n", omp\_get\_max\_threads());  
    
   int thread\_count = -1;  
   while (!(0 < thread\_count && thread\_count <= omp\_get\_max\_threads())) {  
   std::cout << "Enter number of threads to use: ";  
   std::cin >> thread\_count;  
   if (0 < thread\_count && thread\_count <= omp\_get\_max\_threads()) {  
   omp\_set\_num\_threads(thread\_count);  
   break;  
   } else {  
   std::cout << "Invalid thread count entered " << thread\_count << std::endl;  
   }  
   }  
   return thread\_count;  
  }  
    
  int main()  
  {  
   Timer timer;  
   auto pi\_function = [](double x) {  
   return 4 / (double)(1 + x \* x);  
   };  
    
   int thread\_count = identify\_machine();  
    
   printf("Using %d threads...\n", thread\_count);  
    
   uint32\_t precision = 1e8;  
    
   timer.reset();  
   double pi\_serial = simpson\_rule\_serial(0, 1, pi\_function, precision);  
   std::cout << "PI in serial = "   
   << std::setprecision(15)  
   << pi\_serial   
   << std::endl;  
   timer.log("Serial execution took");  
    
   timer.reset();  
   double pi\_parallel = simpson\_rule\_parallel(0, 1, pi\_function, precision);  
   std::cout << "PI in parallel = "   
   << std::setprecision(15)  
   << pi\_parallel   
   << std::endl;  
   timer.log("Parallel execution took");  
  }
* **Output**
* Detected 12 Processor(s)  
  Max 12 Threads(s) Available  
  Enter number of threads to use: 5  
  Using 5 threads...  
  PI in serial = 3.14159265358944  
  [TIMER] Serial execution took : 2.75547154200103s  
  PI in parallel = 3.14159265358966  
  [TIMER] Parallel execution took : 0.62808093700005s

1. Identify the line of statement which leads the race condition. Race condition occurs when the multiple threads accessing a shared variable. If it exists how will you handle this problem? Use appropriate OpenMP clauses such as critical, atomic, ordered, Sections and find the solution. Test the result with classical integration value and by (a) and (b). Calculate the execution time for critical, atomic, ordered, Sections clauses by using the library function omp\_get\_wtime().

* double simpson\_rule\_parallel\_critical(double start, double end, Generator generator, uint32\_t precision) {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   #pragma omp parallel shared(precision, sum)  
   {  
   #pragma omp for  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp critical  
   {  
   sum += 4.0 \* temp;  
   }  
   }  
    
   #pragma omp for  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp critical  
   {  
   sum += 2.0 \* temp;  
   }  
   }  
   }  
    
   sum += generator(x(0)) + generator(x(precision));  
   sum \*= spacing / 3.0;  
    
   return sum;  
  }  
    
  double simpson\_rule\_parallel\_atomic(double start, double end, Generator generator, uint32\_t precision) {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   #pragma omp parallel shared(precision, sum)  
   {  
   #pragma omp for  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp atomic  
   sum += 4.0 \* temp;  
   }  
    
   #pragma omp for  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp atomic  
   sum += 2.0 \* temp;  
   }  
   }  
    
   sum += generator(x(0)) + generator(x(precision));  
   sum \*= spacing / 3.0;  
    
   return sum;  
  }  
    
  double simpson\_rule\_parallel\_ordered(double start, double end, Generator generator, uint32\_t precision) {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   #pragma omp parallel shared(precision, sum)  
   {  
   #pragma omp for ordered  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp ordered  
   sum += 4.0 \* temp;  
   }  
    
   #pragma omp for ordered  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   double temp = generator(x(i));  
   #pragma omp ordered  
   sum += 2.0 \* temp;  
   }  
   }  
    
   sum += generator(x(0)) + generator(x(precision));  
   sum \*= spacing / 3.0;  
    
   return sum;  
  }  
    
  double simpson\_rule\_parallel(double start, double end, Generator generator, uint32\_t precision) {  
   double spacing = ((end - start) / (double)precision);  
   auto x = [start, spacing](uint32\_t index) {  
   return start + index \* spacing;  
   };  
    
   double sum = 0.0F;  
    
   // First Term  
   sum += generator(x(0));  
    
   // Odd sum  
   double odd\_sum = 0.0F;  
   double even\_sum = 0.0F;  
    
   #pragma omp parallel shared(precision, sum)  
   {  
   #pragma omp for reduction( + : odd\_sum )  
   for (uint32\_t i = 1; i <= precision - 1; i += 2) {  
   odd\_sum += generator(x(i));  
   }  
    
   #pragma omp for reduction( + : even\_sum )  
   for (uint32\_t i = 2; i <= precision - 2; i += 2) {  
   even\_sum += generator(x(i));  
   }  
   }  
    
   sum += 4.0 \* odd\_sum + 2.0 \* even\_sum;  
    
   // Last Term  
   sum += generator(x(precision));  
    
   // Final Multiplier  
   sum \*= spacing / 3;  
    
   return sum;  
  }
* **Output**
* Detected 12 Processor(s)  
  Max 12 Threads(s) Available  
  Enter number of threads to use: 10  
  Using 10 threads...  
  PI in serial = 3.14159265358944  
  [TIMER] Serial execution took : 2.77507608699852s  
  PI in parallel (critical) = 3.14159265359004  
  [TIMER] Parallel (critical) execution took : 14.4866989049988s  
  PI in parallel (atomic) = 3.14159265358992  
  [TIMER] Parallel (atomic) execution took : 3.20636564099914s  
  PI in parallel (ordered) = 3.14159265358995  
  [TIMER] Parallel (ordered) execution took : 3.4326813610005s  
  PI in parallel (reduction) = 3.14159265358981  
  [TIMER] Parallel (reduction) execution took : 0.608498552999663s

#### Results

| Execution Type | Calculated PI value | Time Taken in seconds |
| --- | --- | --- |
| Serial | 3.14159265358944 | 2.66738621900004 |
| Parallel (Critical) | 3.14159265359018 | 19.2036709250006 |
| Parallel (Atomic) | 3.14159265359035 | 3.82193632400049 |
| Parallel (Ordered) | 3.14159265358995 | 3.47913229900041 |
| Parallel (Reduction) | 3.14159265358984 | 0.745230704000278 |

### Program 2

Write an openMP program with C++ that illustrates the following OpenMP clause with its various types.

schedule clause: It allows to specify how the iterations of the loop should be scheduled, i.e., allocated to threads. The various types of schedule are as follows.

* Write an openMP program with C++ that calculate the sum of the first N natural numbers using for loop. (Serial Version) Try the following on parallel version of the code.
* **Code**
* #include <iostream>  
  #include <omp.h>  
    
  class Timer {  
  public:  
   Timer() {  
   this->reset();  
   }  
    
   void reset() {  
   m\_Time = omp\_get\_wtime();  
   }  
    
   double elapsed() {  
   double now = omp\_get\_wtime();  
   return now - m\_Time;  
   }  
    
   void log(std::string msg) {  
   double time = this->elapsed();  
   std::cout << "[TIMER] "   
   << msg << " : "   
   << time  
   << "s" << std::endl;  
   }  
  private:  
   double m\_Time;  
  };  
    
  uint64\_t sum\_all(uint64\_t n) {  
   uint64\_t sum = 0;  
   for (uint64\_t i = 0; i < n; i++) {  
   sum += i;  
   }  
   return sum;  
  }  
    
  int main() {  
   Timer timer;  
    
   timer.reset();  
   sum\_all(1e9);  
   timer.log("Serial execution took ");  
  }
* **Output**
* [TIMER] Serial execution took : 1.41841s
* schedule (static), schedule (static, C) where C – number of chunks to tasks. Each chunk contains C contiguous iterations.
* uint64\_t sum\_all\_static(uint64\_t n) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(static)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }  
    
  uint64\_t sum\_all\_static\_chunk(uint64\_t n, int chunk\_size) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(static, chunk\_size)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }
* schedule (dynamic), schedule (dynamic, C)
* uint64\_t sum\_all\_dynamic(uint64\_t n) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(dynamic)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }  
    
  uint64\_t sum\_all\_dynamic\_chunk(uint64\_t n, int chunk\_size) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(dynamic, chunk\_size)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }
* schedule (guided), schedule (guided, C)
* uint64\_t sum\_all\_guided(uint64\_t n) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(guided)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }  
    
  uint64\_t sum\_all\_guided\_chunk(uint64\_t n, int chunk\_size) {  
   uint64\_t sum = 0;  
   #pragma omp parallel for reduction(+:sum) schedule(guided, chunk\_size)  
   for (uint64\_t i = 0; i < n; ++i) {  
   sum += i;  
   }  
   return sum;  
  }

**Final Code**

#include <iostream>  
#include <omp.h>  
#include <cstdint>  
  
class Timer {  
public:  
 Timer() {  
 this->reset();  
 }  
  
 void reset() {  
 m\_Time = omp\_get\_wtime();  
 }  
  
 double elapsed() {  
 double now = omp\_get\_wtime();  
 return now - m\_Time;  
 }  
  
 void log(std::string msg) {  
 double time = this->elapsed();  
 std::cout << "[TIMER] "   
 << msg << " : "   
 << time  
 << "s" << std::endl;  
 }  
private:  
 double m\_Time;  
};  
  
uint64\_t sum\_all\_serial(uint64\_t n) {  
 uint64\_t sum = 0;  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_static(uint64\_t n) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(static)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_static\_chunk(uint64\_t n, int chunk\_size) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(static, chunk\_size)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_dynamic(uint64\_t n) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(dynamic)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_dynamic\_chunk(uint64\_t n, int chunk\_size) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(dynamic, chunk\_size)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_guided(uint64\_t n) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(guided)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
uint64\_t sum\_all\_guided\_chunk(uint64\_t n, int chunk\_size) {  
 uint64\_t sum = 0;  
 #pragma omp parallel for reduction(+:sum) schedule(guided, chunk\_size)  
 for (uint64\_t i = 0; i < n; ++i) {  
 sum += i;  
 }  
 return sum;  
}  
  
int main() {  
 uint64\_t n = 1e9;  
 Timer timer;  
  
 // Serial Execution  
 timer.reset();  
 sum\_all\_serial(n);  
 timer.log("Serial execution took");  
  
 // Static Schedule  
 timer.reset();  
 sum\_all\_static(n);  
 timer.log("Static schedule took");  
  
 // Static Schedule with Chunk Size  
 int chunk\_size = 1000;  
 timer.reset();  
 sum\_all\_static\_chunk(n, chunk\_size);  
 timer.log("Static schedule with chunk size took");  
  
 // Dynamic Schedule  
 timer.reset();  
 sum\_all\_dynamic(n);  
 timer.log("Dynamic schedule took");  
  
 // Dynamic Schedule with Chunk Size  
 timer.reset();  
 sum\_all\_dynamic\_chunk(n, chunk\_size);  
 timer.log("Dynamic schedule with chunk size took");  
  
 // Guided Schedule  
 timer.reset();  
 sum\_all\_guided(n);  
 timer.log("Guided schedule took");  
  
 // Guided Schedule with Chunk Size  
 timer.reset();  
 sum\_all\_guided\_chunk(n, chunk\_size);  
 timer.log("Guided schedule with chunk size took");  
  
 return 0;  
}

[TIMER] Serial execution took : 1.17382s  
[TIMER] Static schedule took : 0.160601s  
[TIMER] Static schedule with chunk size took : 0.165052s  
[TIMER] Dynamic schedule took : 10.0197s  
[TIMER] Dynamic schedule with chunk size took : 0.189753s  
[TIMER] Guided schedule took : 0.192321s  
[TIMER] Guided schedule with chunk size took : 0.184942s