Chp 6 - Exercise 3 - Individual

•Implement the code for communication with Many.
•Create a purely serial version
•Add the ability to track time it takes to complete
•Test for a large array (around 100 million elements should work)
•Record 5 time runs for the serial approach
•Record 5 time runs for the parallel approach using 2 processors
•Record 5 time runs for the parallel approach using 8 processors

Grading Rubric
____ (2 Point) Code for Communication with Many with update to track time
____ (2 Points) Serial version of the code with ability to track time
____ (3 Points) Data from runs organized in a table with good labels.

```
serial version:
#include <algorithm>
#include <iostream>
#include <memory>
#include <random>
#include <chrono>  // Include chrono for timing

int main() {
  // Define the number of elements
  const int num_elements = 1 << 27;

  // Allocate memory for the data
  auto data_ptr = std::make_unique<int[]>(num_elements);

  // Create random number generator
  std::random_device rd;
  std::mt19937 mt(rd());
  std::uniform_int_distribution dist(1, 1);

  // Create random data
  std::generate(data_ptr.get(), data_ptr.get() + num_elements,
                [&] { return dist(mt); });

  // Start the timer
  auto start = std::chrono::high_resolution_clock::now();

  // Calculate the total sum of the array
  auto result = std::reduce(data_ptr.get(), data_ptr.get() +
```

```
num_elements);

  // Stop the timer
  auto stop = std::chrono::high_resolution_clock::now();

  // Calculate the duration
  auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(stop -
start);

  // Print the result
  std::cout << "Sum: " << result << '\n';

  // Print the time taken
  std::cout << "Time taken: " << duration.count() << "
microseconds\n";

  return 0;
}
```

Output:
sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507626 microseconds

**** this was first test run**

5 runs:

sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507612 microseconds
sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507488 microseconds
sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507731 microseconds
sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507582 microseconds
sa7233@sloop:~/fall2024/HPC$ ./comm_many_serial
Sum: 134217728
Time taken: 507699 microseconds

Parallel version:

```
// An example of sum reduction using MPI
// By: Nick from CoffeeBeforeArch

#include <algorithm>
#include <iostream>
#include <memory>
#include <random>

#include "mpi.h"

using namespace std::chrono;

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

  auto start = high_resolution_clock::now();

  // Initialize MPI
  MPI_Init(&argc, &argv);

  // Get the total number of tasks
  int num_tasks;
  MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

  // Calculate chunk size
  // Assume this divides evenly
  const int num_elements = 1 << 27;
  const int chunk_size = num_elements / num_tasks;

  // Get the task ID
  int task_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &task_id);

  // Create buffer for send (only initialized in rank 0)
  std::unique_ptr<int[]> send_ptr;

  // Generate random numbers from rank 0
  if (task_id == 0) {
    // Allocate memory for send buffer
    send_ptr = std::make_unique<int[]>(num_elements);

    // Create random number generator
```

```
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution dist(1, 1);

    // Create random data
    std::generate(send_ptr.get(), send_ptr.get() +
num_elements,
                  [&] { return dist(mt); });
  }

  // Receive buffer
  auto recv_buffer = std::make_unique<int[]>(chunk_size);

  // Perform the scatter of the data to different threads
  MPI_Scatter(send_ptr.get(), chunk_size, MPI_INT,
recv_buffer.get(),
              chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

  // Calculate partial results in each thread
  auto local_result =
      std::reduce(recv_buffer.get(), recv_buffer.get() +
chunk_size);

  // Perform the reduction
  int global_result;
  MPI_Reduce(&local_result, &global_result, 1, MPI_INT,
MPI_SUM, 0,
             MPI_COMM_WORLD);

  // Print the result from rank 0
  if (task_id == 0) {
    std::cout << global_result << '\n';
  }

  // Finish our MPI work
  MPI_Finalize();

  auto stop = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(stop - start);

  cout << "Time taken by function: "
       << duration.count() << " microseconds" << endl;

  return 0;
```

```
}
```

5 runs 2P:

sa7233@sloop:~/fall2024/HPC$ mpirun -n 2 ./comm_many_parallel
Sum: 134217728
Time taken: 464337 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 2 ./comm_many_parallel
Sum: 134217728
Time taken: 464112 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 2 ./comm_many_parallel
Sum: 134217728
Time taken: 464381 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 2 ./comm_many_parallel
Sum: 134217728
Time taken: 464936 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 2 ./comm_many_parallel
Sum: 134217728
Time taken: 461852 microseconds

5 runs 8 p:

sa7233@sloop:~/fall2024/HPC$ mpirun -n 8 ./comm_many_parallel
Sum: 134217728
Time taken: 180327 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 8 ./comm_many_parallel
Sum: 134217728
Time taken: 179149 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 8 ./comm_many_parallel
Sum: 134217728
Time taken: 179804 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 8 ./comm_many_parallel
Sum: 134217728
Time taken: 179676 microseconds
sa7233@sloop:~/fall2024/HPC$ mpirun -n 8 ./comm_many_parallel
Sum: 134217728
Time taken: 179984 microseconds

Serial vs Parallel computing w/ 2p & 8p					
Serial		Parallel 2-Processors		Parallel 8-Processors	
Run ID	Time taken in microseconds	Run ID	Time taken in microseconds	Run ID	Time taken in microseconds
1	507612	1	464337	1	180327
2	507488	2	464112	2	179149
3	507731	3	464381	3	179804
4	507582	4	464936	4	179676
5	507699	5	461852	5	179984
Average	**507622.4**	**Average**	**463923.6**	**Average**	**179788**

The data shows that as the number of parallel processes increases, the time to process 2^{27} elements significantly decreases. Using 8 parallel processes, the time is reduced by more than 64% compared to the serial execution, demonstrating the clear efficiency of parallel computing.