```cpp
///==========================================================================
/// CSCI 176 Program 1
/// Kenneth Willeford
///
///     This program performs parallel partial sums on partitions of an array.
///     Once compiled the program is ran as so...
///         <exe_name> <number_of_thread> <number_of_elements>
///         ie: a.exe 4 400000000   (run with 4 threads and 400000000 elements)
///     I had trouble running the full 500000000 on my system, so I ran with 400000000 instead
///     In addition as I was using a 32 bit compiler I had to use long long integers in order to store my sums.
///==========================================================================
#include <iostream>
#include <cstdlib>
#include <sstream>
#include <pthread.h>
#include <ctime>
using namespace std;

namespace global {
    // contains the amount of threads to run.
    int thread_count;
    // contains the size of the global array.
    int arr_size;
    // contains the size of a given partition in the array.
    int partition_size;
    // the global array itself
    unsigned int* arr;
    // global sum
    long long unsigned int global_sum;
};
// Adds to the global sum, the operation is guarded by a mutex.
void add_to_global_sum(long long unsigned int i);
// Performs a cout command on a string guarded by a mutex.
void cout_semaphore(string);
// Function to call from pthread. Performs a partial sum.
void*perform_partial_sum(void*);
// Retrieves command line arguments and loads them into global variables.
void get_command_line_arguments(char* argv[]);
// Prepares an array to be able to be run in the summation.
void prepare_array();
// Runs the parallel summation.
void run_sums();

int main(int argc, char* argv[]){
    get_command_line_arguments(argv);
    prepare_array();
    run_sums();
}

void*perform_partial_sum(void* v){
    // Get the passed in thread id.
    int thread_id = (int)v;
    // Initialize local sum
    long long unsigned int local_sum = 0;
    // Get the start index based on the partition size and thread id.
    unsigned int start_index = thread_id*global::partition_size;
    // Get the end index based on the partition size and thread id taking into account if it's the last partition.
    unsigned int end_index = ((thread_id+1)*global::partition_size > global::arr_size ? global::arr_size : (thread_id+1)*global::partition_size);
    // Perform the partial sum.
    for(unsigned int i = start_index; i < end_index; i++)
```

```cpp
        local_sum += global::arr[i];
    // Report Local Information
    ostringstream ss;
    ss << "thread_id:" << thread_id << " start_index:" << start_index << " end_index" << end_index << " partial_sum:" << local_sum;
    cout_semaphore(ss.str());
    // Add to the global sum
    add_to_global_sum(local_sum);
}
void cout_semaphore(string s){
    // Shared lock between function calls. Protects cout.
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&lock);
    cout << s << endl;
    pthread_mutex_unlock(&lock);
}
void run_sums(){
    // Begin Benchmark
    clock_t t = clock();
    {
        // Initialize global sum
        global::global_sum = 0;
        // Find partition size
        global::partition_size = global::arr_size/global::thread_count;
        // Launch Threads
        pthread_t threads[global::thread_count];
        for(unsigned int thread_id = 0; thread_id < global::thread_count; thread_id++)
            pthread_create(&threads[thread_id], NULL, perform_partial_sum, (void*)thread_id);
        // Join Threads
        for(unsigned int thread_id = 0; thread_id < global::thread_count; thread_id++)
            pthread_join(threads[thread_id], NULL);
    }
    // End Benchmark
    double seconds = ((float)(clock()-t))/CLOCKS_PER_SEC;
    // Convert global_sum into string and then print data.
    ostringstream ss;
    ss << "global_sum:" << global::global_sum << "time taken:" << seconds << "s";
    cout_semaphore(ss.str());
}
void prepare_array(){
    // Build Array Elements
    for(unsigned int i = 0; i < global::arr_size; i++) global::arr[i] = i+1;
}
void get_command_line_arguments(char* argv[]){
    // Get number of threads to create. If less than 1 or undefined defaults to 1.
    global::thread_count = (atoi(argv[1]) >= 1 ? atoi(argv[1]) : 1);
    // Get size of array to sum. Highest stable array size on my system is along the lines of 476449900 elements. For my tests I'll use 400000000
    // If less than 1000 or undefined defaults to 1000.
    global::arr_size = (atoi(argv[2]) >= 1000 ? atoi(argv[2]) : 1000);
    global::arr = new unsigned int[global::arr_size];
}
void add_to_global_sum(long long unsigned int i){
    // Shared lock between function calls. Protects the sum.
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&lock);
    global::global_sum += i;
    pthread_mutex_unlock(&lock);
}
```

```
g++ parallel-sums.cpp -o parallel-sums.exe
parallel-sums 1 400000000 > output1.txt
parallel-sums 2 400000000 > output2.txt
parallel-sums 4 400000000 > output4.txt
parallel-sums 8 400000000 > output8.txt
```

```
thread_id:0 start_index:0 end_index400000000 partial_sum:80000000200000000
global_sum:80000000200000000time taken:2.065s
```

```
thread_id:0 start_index:0 end_index200000000 partial_sum:20000000100000000
thread_id:1 start_index:200000000 end_index400000000 partial_sum:60000000100000000
global_sum:80000000200000000time taken:0.83s
```

```
thread_id:0 start_index:0 end_index100000000 partial_sum:5000000050000000
thread_id:2 start_index:200000000 end_index300000000 partial_sum:25000000050000000
thread_id:1 start_index:100000000 end_index200000000 partial_sum:15000000050000000
thread_id:3 start_index:300000000 end_index400000000 partial_sum:35000000050000000
global_sum:80000000200000000time taken:0.723s
```

```
thread_id:1 start_index:50000000 end_index100000000 partial_sum:3750000025000000
thread_id:2 start_index:100000000 end_index150000000 partial_sum:6250000025000000
thread_id:4 start_index:200000000 end_index250000000 partial_sum:11250000025000000
thread_id:3 start_index:150000000 end_index200000000 partial_sum:8750000025000000
thread_id:5 start_index:250000000 end_index300000000 partial_sum:13750000025000000
thread_id:0 start_index:0 end_index50000000 partial_sum:1250000025000000
thread_id:6 start_index:300000000 end_index350000000 partial_sum:16250000025000000
thread_id:7 start_index:350000000 end_index400000000 partial_sum:18750000025000000
global_sum:80000000200000000time taken:0.725s
```

```cpp
///////////////////////////////
//// Park -- this is a C++ version of the Pthread Hello program
////
//// compile and run:
//// $> g++ -o Hello Hello.cpp -lpthread
//// $>./Hello 4
////   //4 is the number of threads to create - any
///////////////////////////////

#include <iostream>
#include <cstdlib> //for atoi()
#include <pthread.h>
using namespace std;

//globals --accessible to all threads
int thread_count;       //for command line arg

///==================================================================
/// Global Mutex    <ADDED>
///==================================================================
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
///==================================================================
void *Hello(void* rank); //prototype for a Thread function

///////////////////
int main(int argc, char* argv[]){
  long thread_id; //long for type conversion [long<-->void*] for 64 bit system

  thread_count = atoi(argv[1]); //tot number of threads - from command line
  pthread_t myThreads[thread_count]; //define threads

  //creates a certain number of threads
  for(thread_id = 0; thread_id < thread_count; thread_id++)
     pthread_create(&myThreads[thread_id], NULL, Hello, (void*)thread_id);
///==================================================================
/// Protected cout  <ADDED>
///==================================================================
  pthread_mutex_lock(&lock);
  cout<<"Hello from the main thread"<<endl;
  pthread_mutex_unlock(&lock);
///==================================================================
  //wait until all threads finish
  for(thread_id = 0; thread_id < thread_count; thread_id++)
     pthread_join(myThreads[thread_id], NULL);

  return 0;
}//main

/////////////////slave function
void *Hello(void* rank) {
  int my_rank = (long)rank; //rank is void* type, so can cast to (long) type only;
///==================================================================
/// Protected cout  <ADDED>
///==================================================================
  pthread_mutex_lock(&lock);
  cout<<"Hello from thread_"<<my_rank<<" of "<<thread_count<<endl;
  pthread_mutex_unlock(&lock);
///==================================================================
  return NULL;
}//Hello
```