```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <cmath>
#include <mpi.h>

template<typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

template<typename T>
int partition(std::vector<T>& arr, int low, int high) {
    T pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

template<typename T>
void quicksort(std::vector<T>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

template<typename T>
bool is_sorted(const std::vector<T>& arr) {
    for (size_t i = 0; i + 1 < arr.size(); ++i) {
        if (arr[i] > arr[i + 1]) {
            return false;
        }
    }
    return true;
}


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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    long long N = 1000000;
    if (argc > 1) {
```

```
    try {
        N = std::stoll(argv[1]);
    } catch (const std::exception& e) {
        if (rank == 0) {
            std::cerr << "Invalid array size argument. Using default N = "
            << N << std::endl;
        }
    }
}

if (rank == 0) {
    std::cout << "----------------------------------------" << std::endl;
    std::cout << "MPI Quicksort Performance Evaluation (Kaggle Node)" <<
    std::endl;
    std::cout << "Array Size (N): " << N << std::endl;
    std::cout << "MPI Processes Requested: " << size << std::endl;
    std::cout << "----------------------------------------" << std::endl;
}

std::vector<int> data;
std::vector<int> sequential_data;
double sequential_time = 0.0;

if (rank == 0) {
    data.resize(N);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, N * 10);
    for (long long i = 0; i < N; ++i) {
        data[i] = distrib(gen);
    }

    sequential_data = data;

    auto start_seq = std::chrono::high_resolution_clock::now();
    quicksort(sequential_data, 0, N - 1);
    auto end_seq = std::chrono::high_resolution_clock::now();
    sequential_time = std::chrono::duration<double>(end_seq -
    start_seq).count();
    std::cout << "Sequential Time: " << sequential_time << " s" <<
    std::endl;

    if (!is_sorted(sequential_data)) {
         std::cerr << "Sequential sort FAILED!" << std::endl;
    }
}

MPI_Barrier(MPI_COMM_WORLD);
double start_parallel = MPI_Wtime();

int local_n_base = N / size;
int remainder = N % size;
int local_n = (rank < remainder) ? local_n_base + 1 : local_n_base;

std::vector<int> sendcounts(size);
std::vector<int> displs(size);
int current_displ = 0;
for (int i = 0; i < size; ++i) {
```

```
    sendcounts[i] = (i < remainder) ? local_n_base + 1 : local_n_base;
    displs[i] = current_displ;
    current_displ += sendcounts[i];
}

std::vector<int> local_data(local_n);
MPI_Scatterv(rank == 0 ? data.data() : nullptr, sendcounts.data(),
displs.data(), MPI_INT,
             local_data.data(), local_n, MPI_INT, 0, MPI_COMM_WORLD);

std::sort(local_data.begin(), local_data.end());

int num_samples = size > 1 ? size : 1;
std::vector<int> samples;
 samples.reserve(num_samples);
if (local_n > 0) {
   for (int i = 0; i < num_samples; ++i) {
      long long sample_index = static_cast<long long>(i) * local_n /
       num_samples;
       if (sample_index >= local_n) sample_index = local_n -1;
      samples.push_back(local_data[sample_index]);
  }
} else {
     samples.assign(num_samples, 0);
}
if (samples.size() < num_samples) {
    samples.resize(num_samples, samples.empty() ? 0 : samples.back());
}


std::vector<int> gathered_samples;
if (rank == 0) {
    gathered_samples.resize(size * num_samples);
}
MPI_Gather(samples.data(), num_samples, MPI_INT,
           gathered_samples.data(), num_samples, MPI_INT, 0,
           MPI_COMM_WORLD);

std::vector<int> pivots(size - 1);
if (size > 1 && rank == 0) {
     std::sort(gathered_samples.begin(), gathered_samples.end());
     for (int i = 0; i < size - 1; ++i) {
         long long pivot_index = static_cast<long long>(i + 1) *
          num_samples - 1;
          if (pivot_index < 0) pivot_index = 0;
         if (pivot_index >= gathered_samples.size()) pivot_index =
         gathered_samples.size() -1;
         pivots[i] = gathered_samples[pivot_index];
     }
    for (int i = 0; i < size - 2; ++i) {
        if (pivots[i] >= pivots[i+1]) {
            pivots[i+1] = pivots[i] + 1;
        }
    }
}

if (size > 1) {
    MPI_Bcast(pivots.data(), size - 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
}

std::vector<std::vector<int>> partitions(size);
std::vector<int> send_counts_alltoall(size, 0);
if (size > 1) {
    for(int val : local_data) {
         int dest_rank = 0;
         while(dest_rank < size - 1 && val >= pivots[dest_rank]) {
             dest_rank++;
         }
         partitions[dest_rank].push_back(val);
         send_counts_alltoall[dest_rank]++;
     }
} else {
     partitions[0] = local_data;
     send_counts_alltoall[0] = local_data.size();
}


std::vector<int> recv_counts_alltoall(size);
MPI_Alltoall(send_counts_alltoall.data(), 1, MPI_INT,
             recv_counts_alltoall.data(), 1, MPI_INT, MPI_COMM_WORLD);

std::vector<int> send_displs_alltoall(size, 0);
std::vector<int> recv_displs_alltoall(size, 0);
int total_send_size = 0;
int total_recv_size = 0;
for (int i = 0; i < size; ++i) {
    send_displs_alltoall[i] = total_send_size;
    recv_displs_alltoall[i] = total_recv_size;
    total_send_size += send_counts_alltoall[i];
    total_recv_size += recv_counts_alltoall[i];
}

std::vector<int> send_buffer_alltoall(total_send_size);
if (size > 1) {
   int current_pos = 0;
   for(int i=0; i<size; ++i) {
        if (send_counts_alltoall[i] > 0) {
            std::copy(partitions[i].begin(), partitions[i].end(),
            send_buffer_alltoall.begin() + current_pos);
            current_pos += send_counts_alltoall[i];
        }
   }
} else {
    send_buffer_alltoall = partitions[0];
}

std::vector<int> recv_buffer_alltoall(total_recv_size);
MPI_Alltoallv(send_buffer_alltoall.data(), send_counts_alltoall.data(),
              send_displs_alltoall.data(), MPI_INT,
              recv_buffer_alltoall.data(), recv_counts_alltoall.data(),
              recv_displs_alltoall.data(), MPI_INT, MPI_COMM_WORLD);

std::sort(recv_buffer_alltoall.begin(), recv_buffer_alltoall.end());

std::vector<int> final_recv_counts(size);
std::vector<int> final_displs(size);
```

```
    int final_local_size = recv_buffer_alltoall.size();
    MPI_Gather(&final_local_size, 1, MPI_INT, final_recv_counts.data(), 1,
    MPI_INT, 0, MPI_COMM_WORLD);

    std::vector<int> final_data;
    if (rank == 0) {
        final_displs[0] = 0;
        long long total_final_size = final_recv_counts[0];
        for (int i = 1; i < size; ++i) {
            final_displs[i] = final_displs[i - 1] + final_recv_counts[i - 1];
            total_final_size += final_recv_counts[i];
        }
         if (total_final_size != N) {
            std::cerr << "WARNING: Final calculated size mismatch! " <<
            total_final_size << " != " << N << ". Resizing result vector." <<
            std::endl;
            final_data.resize(total_final_size);
         } else {
             final_data.resize(N);
         }
    }

    MPI_Gatherv(recv_buffer_alltoall.data(), final_local_size, MPI_INT,
                final_data.data(), final_recv_counts.data(),
                final_displs.data(), MPI_INT,
                0, MPI_COMM_WORLD);


    MPI_Barrier(MPI_COMM_WORLD);
    double end_parallel = MPI_Wtime();
    double parallel_time = end_parallel - start_parallel;

    if (rank == 0) {
        std::cout << "Parallel Time:   " << parallel_time << " s" << std::endl;

         if (!is_sorted(final_data)) {
             std::cerr << "Parallel sort FAILED!" << std::endl;
         } else {
             std::cout << "Parallel sort Verified." << std::endl;
         }

        std::cout << "----------------------------------------" << std::endl;
        if (parallel_time > 1e-9 && sequential_time > 1e-9) {
            double speedup = sequential_time / parallel_time;
            std::cout << "Speedup:         " << speedup << "x" << std::endl;
        } else {
            std::cout << "Speedup:         N/A (Sequential or Parallel time too
            small)" << std::endl;
        }
        std::cout << "----------------------------------------" << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Output

```
----------------------------------------
MPI Quicksort Performance Evaluation (Kaggle Node)
Array Size (N): 2000000
MPI Processes Requested: 8
----------------------------------------
Sequential Time: 0.53989 s
Parallel Time:   0.238298 s
Parallel sort Verified.
----------------------------------------
Speedup:         2.26561x
----------------------------------
```