**PARALLEL COMPUTING SYSTEM ASSIGNMENT**

**TOPIC: MP AND OMP**

**Submitted by:**

**S Thanga Tamil Selvi**

# BUBBLE SORTING

**BUBBLE SORT USING MPI**

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <sys/time.h>

#define SIZE 7

void swap(int \*xp, int \*yp) { int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

void bubbleSort(int arr[], int n) { for (int i = 0; i < n-1; i++)

for (int j = 0; j < n-i-1; j++) if (arr[j] > arr[j+1])

swap(&arr[j], &arr[j+1]);

}

int main(int argc, char\*\* argv) { int rank, size; MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int \*data;

int localSize = SIZE / size;

data = (int \*)malloc(SIZE \* sizeof(int));

if (rank == 0) {

// Initialize array with specified values

int initialData[SIZE] = {22, 90, 77, 55, 33, 11, 1}; for (int i = 0; i < SIZE; i++)

data[i] = initialData[i];

}

// Scatter the data to all processes

MPI\_Scatter(data, localSize, MPI\_INT, data, localSize, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure the start time double start\_time;

if (rank == 0)

start\_time = MPI\_Wtime();

// Perform local bubble sort bubbleSort(data, localSize);

// Gather the sorted data

MPI\_Gather(data, localSize, MPI\_INT, data, localSize, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure the end time double end\_time;

if (rank == 0) {

end\_time = MPI\_Wtime(); printf("Sorted array: ");

for (int i = 0; i < SIZE; i++) { printf("%d ", data[i]);

}

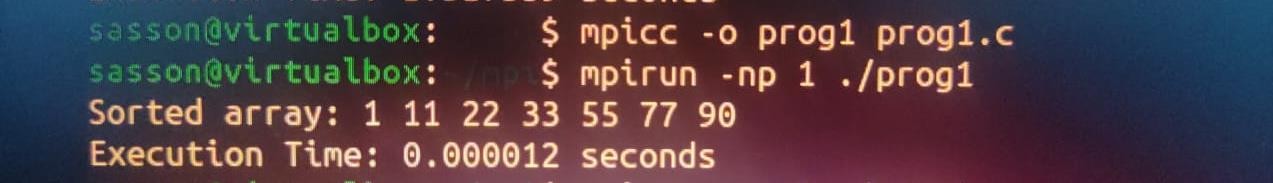
printf("\n");

printf("Execution Time: %f seconds\n", end\_time - start\_time);

}

MPI\_Finalize(); free(data); return 0;

}



# EVEN PHASE ODD PHASE SORTING

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <time.h>

void bubbleSort(int arr[], int n) { int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

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);

int n = 7; // Size of the array

int local\_n = n / size; // Size of each process's local array int arr[n];

// Initialize the array with specific values on the root process if (rank == 0) {

int specificValues[] = {22, 90, 77, 55, 33, 11, 1}; for (int i = 0; i < n; i++) {

arr[i] = specificValues[i];

}

}

int local\_arr[local\_n];

// Scatter the array to all processes

MPI\_Scatter(arr, local\_n, MPI\_INT, local\_arr, local\_n, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double start\_time = MPI\_Wtime();

// Perform local bubble sort bubbleSort(local\_arr, local\_n);

// Gather sorted subarrays back to the root process MPI\_Gather(local\_arr, local\_n, MPI\_INT, arr, local\_n, MPI\_INT, 0,

MPI\_COMM\_WORLD);

// Measure execution time

double end\_time = MPI\_Wtime();

double execution\_time = end\_time - start\_time;

if (rank == 0) {

// Print the sorted array printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

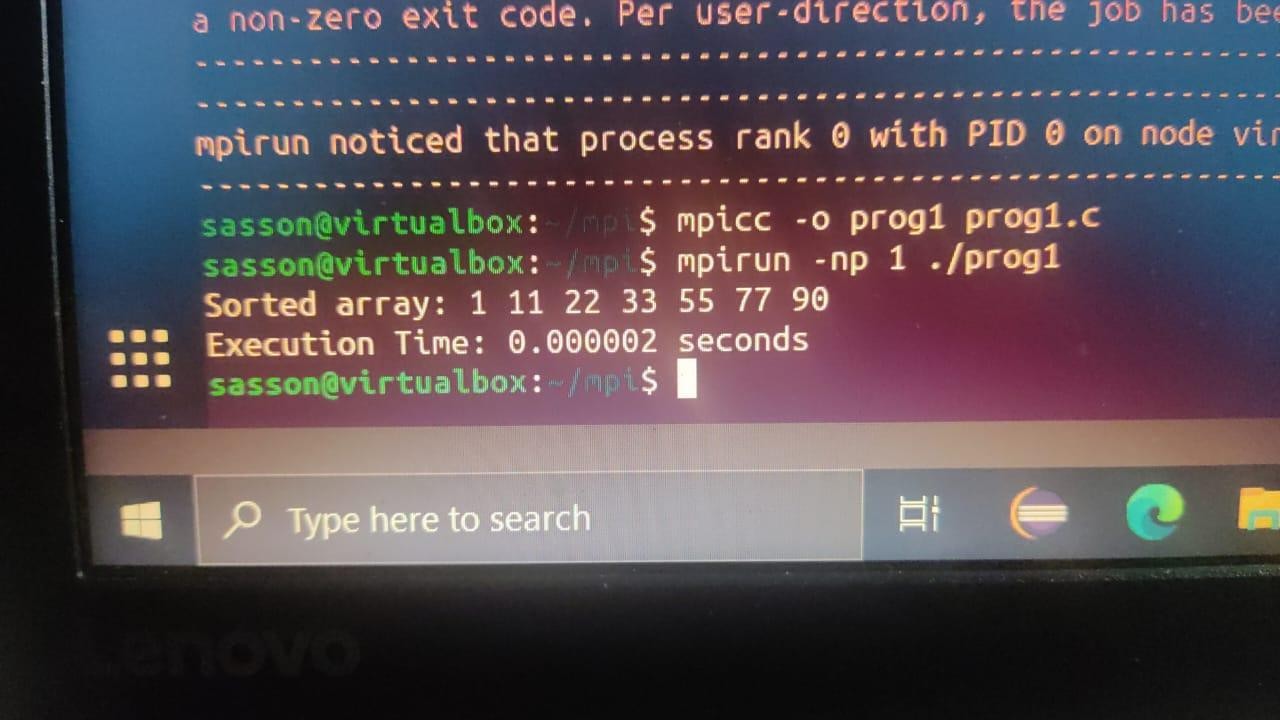
printf("\n");

printf("Execution Time: %f seconds\n", execution\_time);

}

MPI\_Finalize(); return 0;

}



# MPI BROADCAST

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <time.h>

void bubbleSort(int arr[], int n) { int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

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);

int n = 7; // Size of the specific values array int arr[n];

// Initialize the array with specific values on the root process if (rank == 0) {

int specificValues[] = {22, 90, 77, 55, 33, 11, 1}; for (int i = 0; i < n; i++) {

arr[i] = specificValues[i];

}

}

// Broadcast the array to all processes

MPI\_Bcast(arr, n, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double start\_time = MPI\_Wtime();

// Perform parallel bubble sort for (int i = 0; i < n; i++) {

if (i % 2 == 0) {

// Even phase

MPI\_Barrier(MPI\_COMM\_WORLD); // Synchronize processes before even phase

bubbleSort(arr, n);

} else {

// Odd phase

MPI\_Barrier(MPI\_COMM\_WORLD); // Synchronize processes before odd phase bubbleSort(arr, n);

}

}

// Gather sorted subarrays back to the root process

MPI\_Gather(arr, n, MPI\_INT, arr, n, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double end\_time = MPI\_Wtime();

double execution\_time = end\_time - start\_time;

if (rank == 0) {

// Print the sorted array printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

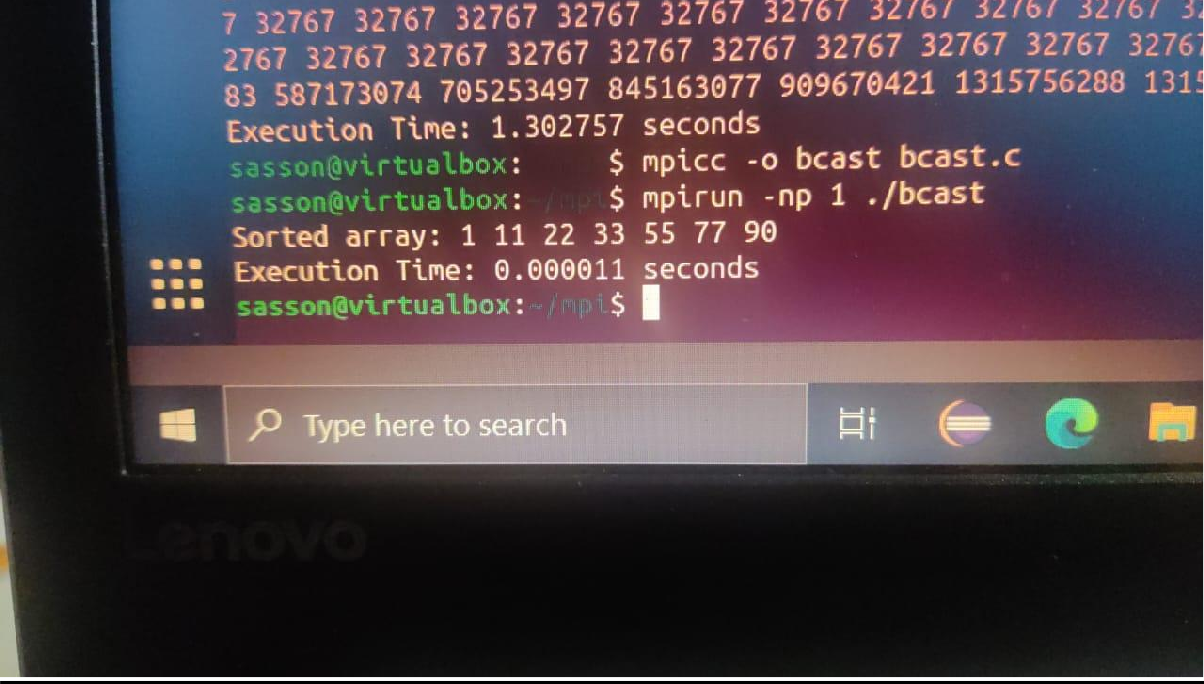
printf("\n");

printf("Execution Time: %f seconds\n", execution\_time);

}

MPI\_Finalize(); return 0;

}



# MPI GATHER

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <time.h>

void bubbleSort(int arr[], int n) { int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

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);

int n = 7; // Size of the array int arr[n];

// Initialize the array with specific values on the root process if (rank == 0) {

int specificValues[] = {22, 90, 77, 55, 33, 11, 1}; for (int i = 0; i < n; i++) {

arr[i] = specificValues[i];

}

}

int local\_size = n / size; // Size of each local array int local\_arr[local\_size];

// Scatter the array to local arrays

MPI\_Scatter(arr, local\_size, MPI\_INT, local\_arr, local\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double start\_time = MPI\_Wtime();

// Perform parallel bubble sort for (int i = 0; i < n; i++) {

if (i % 2 == 0) {

// Even phase bubbleSort(local\_arr, local\_size);

}

MPI\_Barrier(MPI\_COMM\_WORLD); // Synchronize processes before gathering MPI\_Gather(local\_arr, local\_size, MPI\_INT, arr, local\_size, MPI\_INT, 0,

MPI\_COMM\_WORLD);

if (i % 2 == 1) {

// Odd phase bubbleSort(arr, n);

}

MPI\_Barrier(MPI\_COMM\_WORLD); // Synchronize processes before gathering MPI\_Gather(arr, local\_size, MPI\_INT, local\_arr, local\_size, MPI\_INT, 0,

MPI\_COMM\_WORLD);

}

// Measure execution time

double end\_time = MPI\_Wtime();

double execution\_time = end\_time - start\_time;

if (rank == 0) {

// Print the sorted array printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

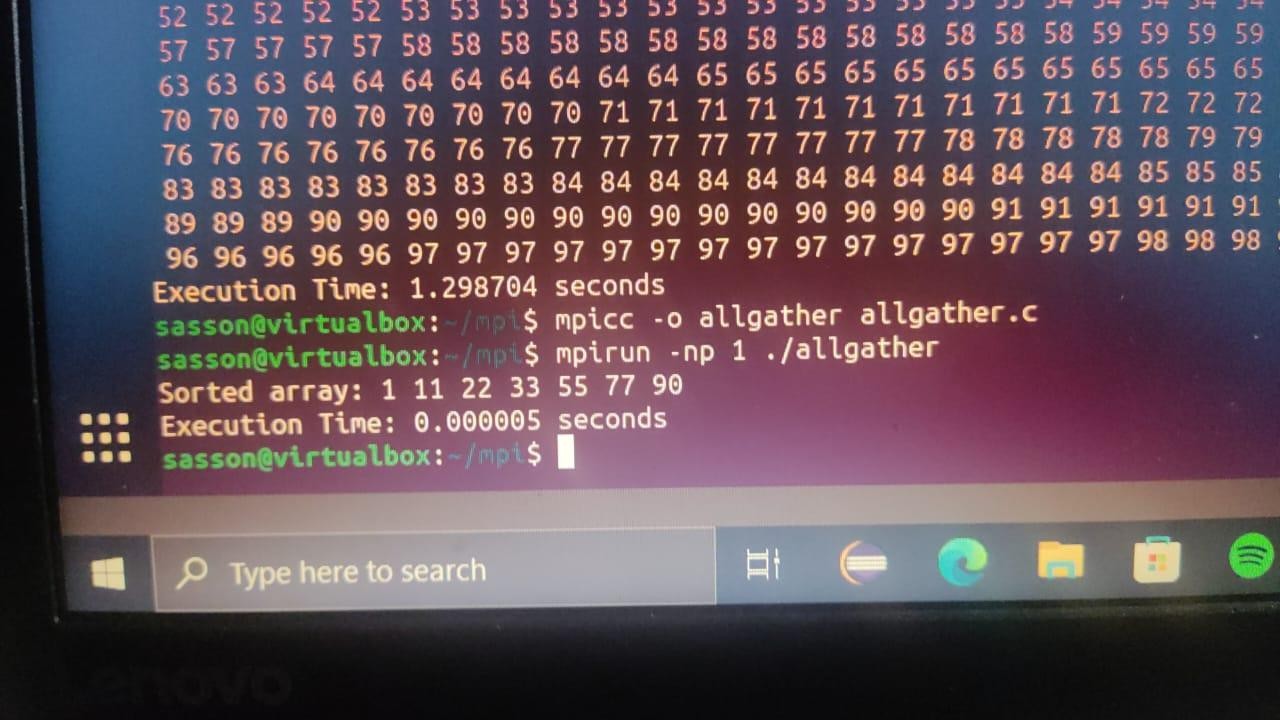
printf("\n");

printf("Execution Time: %f seconds\n", execution\_time);

}

MPI\_Finalize(); return 0;

}



# MPI REDUCE

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <time.h>

void bubbleSort(int arr[], int n) { int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

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);

int n = 7; // Size of the array int arr[n];

// Initialize the array with specific values on the root process if (rank == 0) {

int specificValues[] = {22, 90, 77, 55, 33, 11, 1}; for (int i = 0; i < n; i++) {

arr[i] = specificValues[i];

}

}

int local\_size = n / size; // Size of each local array int local\_arr[local\_size];

// Scatter the array to local arrays

MPI\_Scatter(arr, local\_size, MPI\_INT, local\_arr, local\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double start\_time = MPI\_Wtime();

// Perform parallel bubble sort for (int i = 0; i < n; i++) {

bubbleSort(local\_arr, local\_size);

MPI\_Barrier(MPI\_COMM\_WORLD); // Synchronize processes before reducing MPI\_Reduce(local\_arr, arr, n, MPI\_INT, MPI\_MIN, 0, MPI\_COMM\_WORLD);

}

// Measure execution time

double end\_time = MPI\_Wtime();

double execution\_time = end\_time - start\_time;

if (rank == 0) {

// Print the sorted array printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

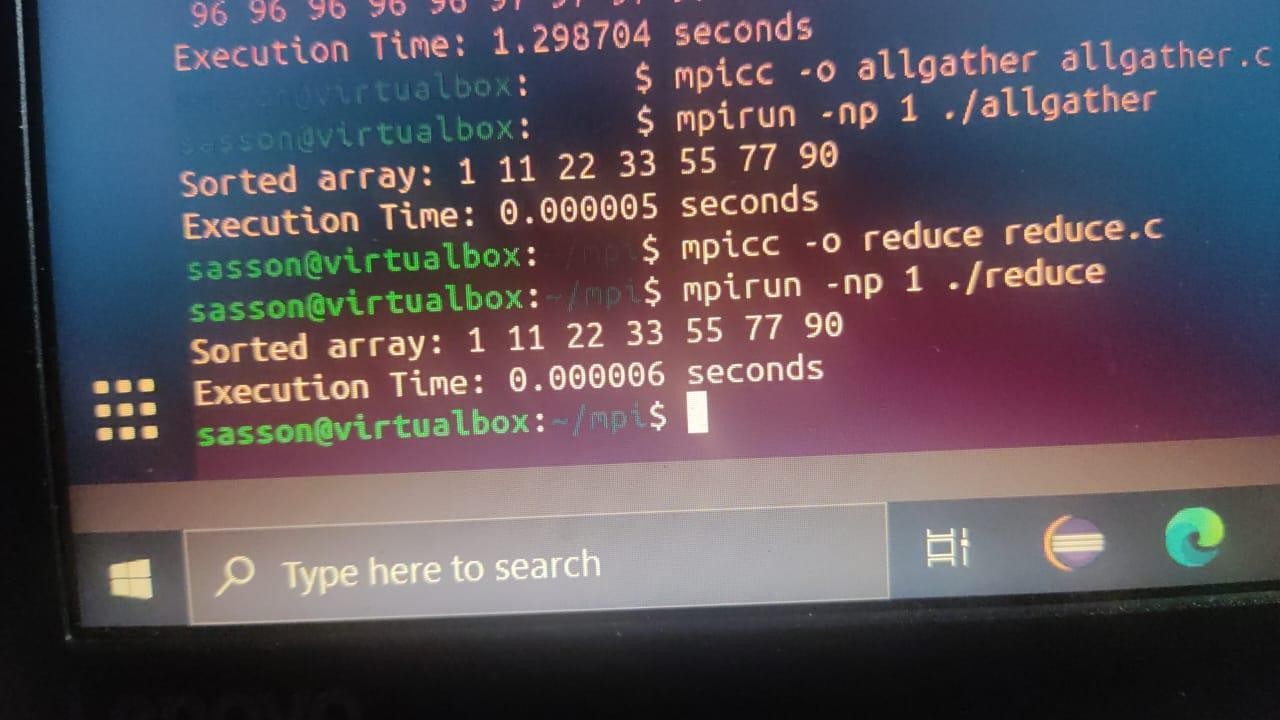
printf("\n");

printf("Execution Time: %f seconds\n", execution\_time);

}

MPI\_Finalize();

}



# MPI ALL REDUCE

#include <stdio.h> #include <stdlib.h> #include <mpi.h> #include <time.h>

void bubbleSort(int arr[], int n) { int temp;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) { if (arr[j] > arr[j + 1]) {

temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

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);

int n = 7; // Size of the array

int local\_size = n / size; // Size of each local array

int local\_arr[local\_size]; int arr[n];

if (rank == 0) {

// Initialize the array with specific values on the root process int specificValues[] = {22, 90, 77, 55, 33, 11, 1};

for (int i = 0; i < n; i++) { arr[i] = specificValues[i];

}

}

// Broadcast the array to all processes

MPI\_Bcast(arr, n, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Scatter the array to local arrays

MPI\_Scatter(arr, local\_size, MPI\_INT, local\_arr, local\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Measure execution time

double start\_time = MPI\_Wtime();

// Perform parallel bubble sort for (int i = 0; i < n; i++) {

bubbleSort(local\_arr, local\_size);

// Allgather to ensure correct ordering for the next iteration MPI\_Allgather(local\_arr, local\_size, MPI\_INT, arr, local\_size, MPI\_INT,

MPI\_COMM\_WORLD);

}

// Calculate local sum int local\_sum = 0;

for (int i = 0; i < local\_size; i++) { local\_sum += local\_arr[i];

}

// Allreduce local sums to find the global sum int global\_sum;

MPI\_Allreduce(&local\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD);

// Measure execution time

double end\_time = MPI\_Wtime();

double execution\_time = end\_time - start\_time;

if (rank == 0) {

// Print the sorted array printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

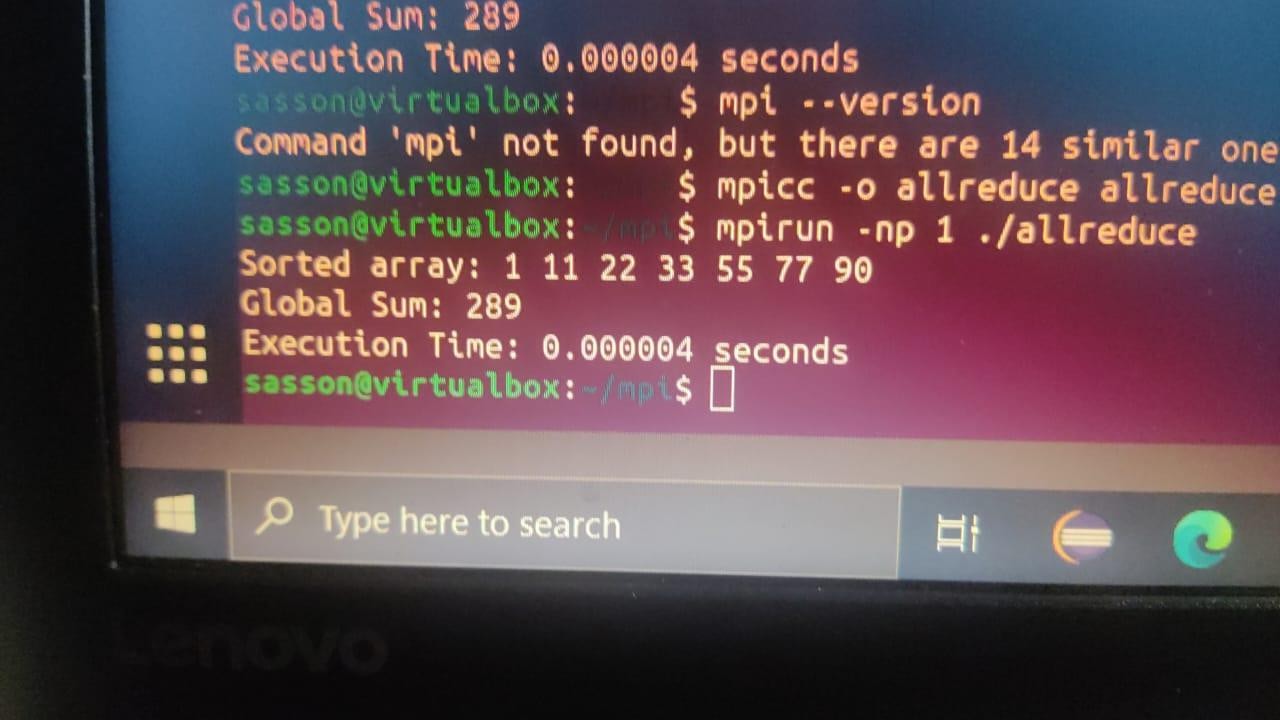
printf("\n");

printf("Global Sum: %d\n", global\_sum); printf("Execution Time: %f seconds\n", execution\_time);

}

MPI\_Finalize(); return 0;

}



**SELECTION**

**The selection among MPI\_Bcast, MPI\_Scatter, MPI\_Gather, MPI\_Reduce, or MPI\_Allreduce significantly relies on the unique requirements and behavior exhibited by your MPI program. Each MPI collective operation serves a specific purpose, and the ideal choice is contingent upon the communication pattern and the manner in which data is distributed in your application.**

**MPI\_Bcast: Primarily used when a solitary source, usually the root process, must disseminate data to all other processes. Bubble Sort Consideration: If the data intended for sorting remains consistent across all processes, MPI\_Bcast efficiently distributes the original array to all processes prior to sorting.**

**MPI\_Scatter: Suitable for scenarios requiring the distribution of distinct segments of an array to different processes. Bubble Sort Consideration: When the initial array needs division into segments, each transmitted to various processes for sorting, and subsequent aggregation of the results is necessary.**

**MPI\_Gather: Valuable for accumulating data from multiple processes onto a singular process, often designated as the root process. Bubble Sort Consideration: Relevant when different segments of the array undergo sorting on separate processes, and the sorted segments need to be assembled on a single process for subsequent actions.**

**MPI\_Reduce: Utilized for executing a reduction operation across all processes, typically culminating in a solitary value. Bubble Sort Consideration: If aiming for a comprehensive reduction operation (e.g., determining the sum of all elements in the sorted array) across multiple processes.**

**MPI\_Allreduce: Deployed for executing a reduction operation across all processes, with each process obtaining the resultant value. Bubble Sort Consideration: When each process necessitates possessing the outcome of a comprehensive reduction operation without an additional step to gather the results.**

**In an MPI Bubble Sort implementation, the communication pattern plays a pivotal role. A combination of MPI\_Scatter and MPI\_Gather might be appropriate for scenarios demanding data distribution and independent local sorting. For global reduction operations, MPI\_Allreduce or MPI\_Reduce might present more suitable options.**

# BUBBLE SORT PROGRAM IN OPEN MP

#include <stdio.h> #include <stdlib.h> #include <omp.h>

void bubbleSort(int arr[], int n) { int temp;

#pragma omp parallel for for (int i = 0; i < n - 1; i++) {

#pragma omp parallel for shared(arr) private(temp) for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) { #pragma omp critical

{

temp = arr[j];

arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

}

int main() {

int n = 10; // Adjust the size of the array as needed int arr[] = {64, 34, 25, 12, 22, 11, 90, 88, 75, 50};

printf("Original array: "); for (int i = 0; i < n; i++) { printf("%d ", arr[i]);

}

printf("\n");

// Perform parallel bubble sort bubbleSort(arr, n);

printf("Sorted array: "); for (int i = 0; i < n; i++) {

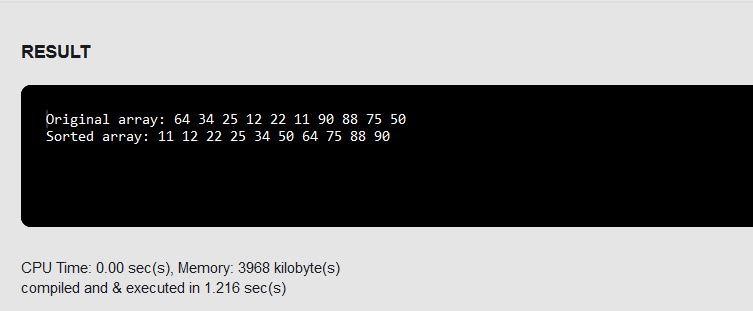
printf("%d ", arr[i]);

}

printf("\n"); return 0;

}

OUTPUT:



# EVEN PHASE AND ODD PHASE BUBBLE SORTING

#include <stdio.h> #include <stdlib.h> #include <omp.h>

void oddEvenBubbleSort(int arr[], int n) { int temp;

int sorted = 0;

while (!sorted) { sorted = 1;

// Odd phase

#pragma omp parallel for shared(arr) private(temp) reduction(&&:sorted) for (int i = 1; i < n - 1; i += 2) {

if (arr[i] > arr[i + 1]) { temp = arr[i];

arr[i] = arr[i + 1]; arr[i + 1] = temp;

sorted = 0; // Set sorted to false if a swap occurred

}

}

// Even phase

#pragma omp parallel for shared(arr) private(temp) reduction(&&:sorted) for (int i = 0; i < n - 1; i += 2) {

if (arr[i] > arr[i + 1]) { temp = arr[i];

arr[i] = arr[i + 1]; arr[i + 1] = temp;

sorted = 0; // Set sorted to false if a swap occurred

}

}

// Ensure all threads have finished their work before checking 'sorted' #pragma omp barrier

}

}

int main() {

int n = 10; // Adjust the size of the array as needed int arr[] = {64, 34, 25, 12, 22, 11, 90, 88, 75, 50};

printf("Original array: "); for (int i = 0; i < n; i++) { printf("%d ", arr[i]);

}

printf("\n");

// Perform odd-even phase parallel bubble sort oddEvenBubbleSort(arr, n);

printf("Sorted array: "); for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}



Critical Section:

Code 1 implements a singular critical section encompassing the entire sorting operation, guaranteeing exclusive access for a single thread to execute the sorting process, thereby preventing race conditions. This safeguards the integrity of shared resources.

1.In contrast, Code 2 introduces a nested parallel region with a critical section nested within the inner loop. Such nesting may result in inefficiencies and might not be indispensable for the particular sorting algorithm in use.

2.A critical section refers to a specific code segment that mandates execution by only one thread or process at any given time to avert race conditions and ensure the coherence of shared resources.

3.Barrier in Code 2: Code 2 features #pragma omp barrier after each phase of the sorting algorithm, a characteristic absent in Code 1. These barriers ascertain that all threads conclude their tasks within the current phase before proceeding to the subsequent phase. Nonetheless, this approach of employing barriers could potentially constrain parallelism.

4.Private Variables: Both codes incorporate the usage of private variables (e.g., temp) to circumvent data races. Employing private variables is crucial in a parallel environment to ensure accuracy and consistency in execution.

Suggestion: Opting for a single critical section to safeguard the complete sorting operation, similar to the approach in Code 1, is typically more efficient than implementing nested parallelism alongside barriers. The critical section ensures mutual exclusion while allowing for better parallelism. You may further optimize the parallelization based on the specific requirements of your application and the characteristics of the sorting algorithm.

# PTHREAD PROGRAM TO OPEN MP:

**PTHREAD**

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <time.h>

#define NUM\_THREADS 4

#define ARRAY\_SIZE 1000

int array[ARRAY\_SIZE]; int sum = 0;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; // Declare mutex

void \*computeSum(void \*threadID) { long id = (long)threadID;

int localSum = 0;

for (int i = id \* (ARRAY\_SIZE / NUM\_THREADS); i < (id + 1) \* (ARRAY\_SIZE / NUM\_THREADS); ++i) {

localSum += array[i];

}

// Critical section pthread\_mutex\_lock(&mutex); sum += localSum; pthread\_mutex\_unlock(&mutex);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS]; long t;

// Initialize array

for (int i = 0; i < ARRAY\_SIZE; ++i) { array[i] = i + 1;

}

// Measure execution time clock\_t start\_time = clock();

// Create threads

for (t = 0; t < NUM\_THREADS; ++t) {

pthread\_create(&threads[t], NULL, computeSum, (void \*)t);

}

// Join threads

for (t = 0; t < NUM\_THREADS; ++t) {

pthread\_join(threads[t], NULL);

}

// Measure execution time clock\_t end\_time = clock();

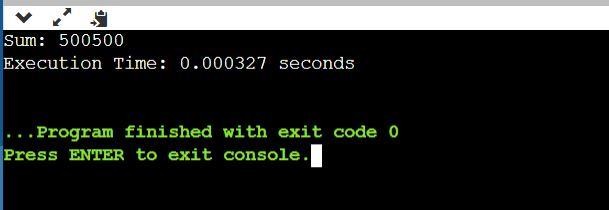
double execution\_time = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("Sum: %d\n", sum);

printf("Execution Time: %f seconds\n", execution\_time);

pthread\_exit(NULL);

}



# OPEN MP

#include <stdio.h> #include <stdlib.h> #include <omp.h>

#define NUM\_THREADS 4

#define ARRAY\_SIZE 1000

int array[ARRAY\_SIZE]; int sum = 0;

int main() { long t;

// Initialize array

for (int i = 0; i < ARRAY\_SIZE; ++i) { array[i] = i + 1;

}

// Parallel region with OpenMP

#pragma omp parallel num\_threads(NUM\_THREADS) private(t)

{

int localSum = 0;

// Each thread computes its local sum #pragma omp for

for (int i = 0; i < ARRAY\_SIZE; ++i) { localSum += array[i];

}

// Critical section to update the global sum #pragma omp critical

{

sum += localSum;

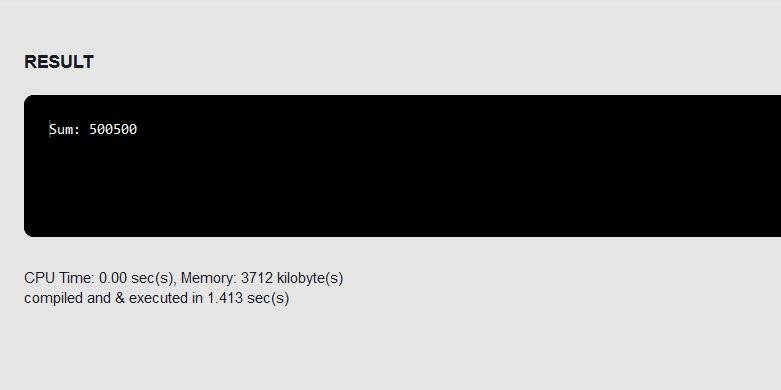
}

}

printf("Sum: %d\n", sum);

return 0;

}



1.Substituting Pthreads functions with OpenMP directives:

Replacing pthread\_create and pthread\_join functions with OpenMP directives involves utilizing #pragma omp parallel and #pragma omp for for parallel regions and loop execution, respectively.

Unlike the Pthreads version, the OpenMP version doesn't require a pthread\_exit call at the conclusion of the execution.

2.Private variables:

In the OpenMP version, the variable 't' is explicitly declared as private within the parallel directive to ensure its scope remains confined to individual threads.

3.Critical section:

In OpenMP, establishing a critical section, specifically where the global sum is updated, involves using #pragma omp critical to ensure exclusive access to this section, preventing race conditions when multiple threads access shared resources simultaneously.