**PARALLEL COMPUTING SYSTEM ASSIGNMENT**

**TOPIC: MP AND OMP**

**Submitted by:**

**T Nithya Shree**

# 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]);

}

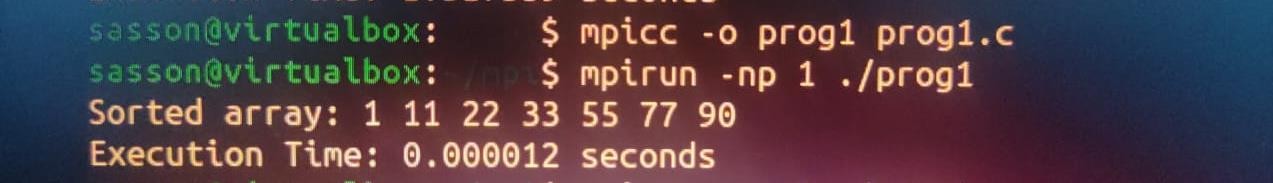
printf("\n");

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]);

}

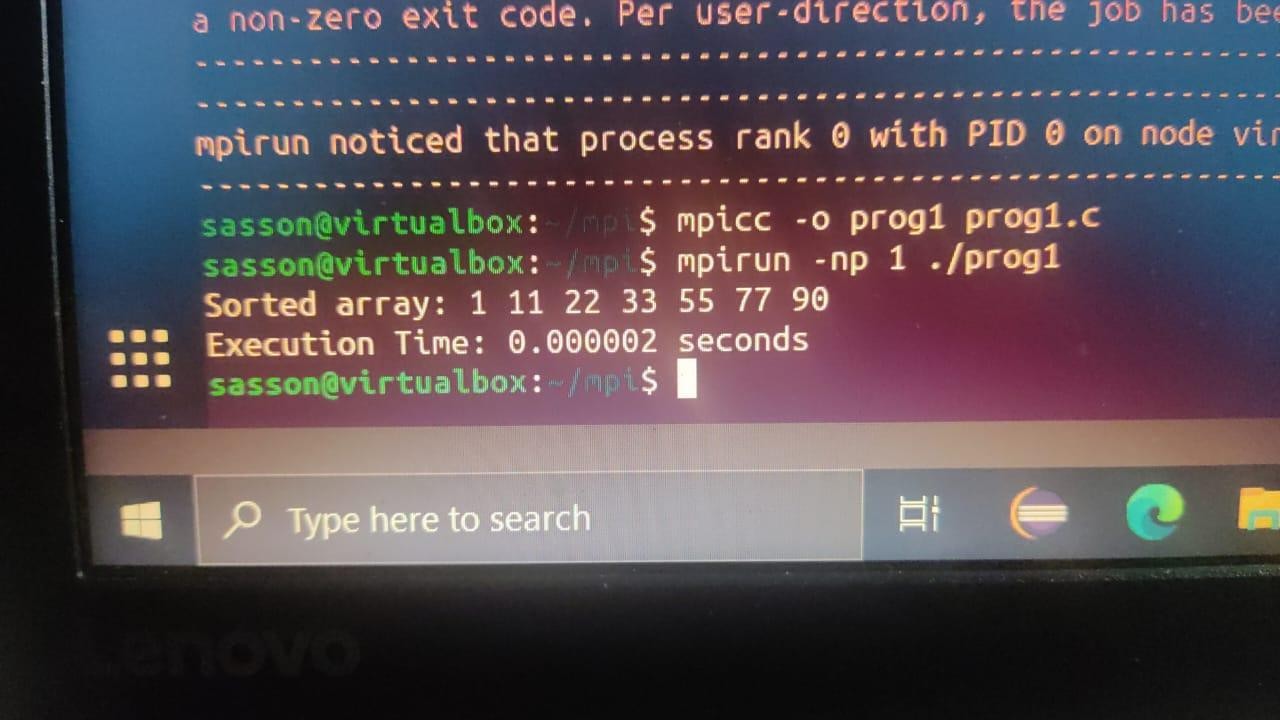
printf("\n");

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]);

}

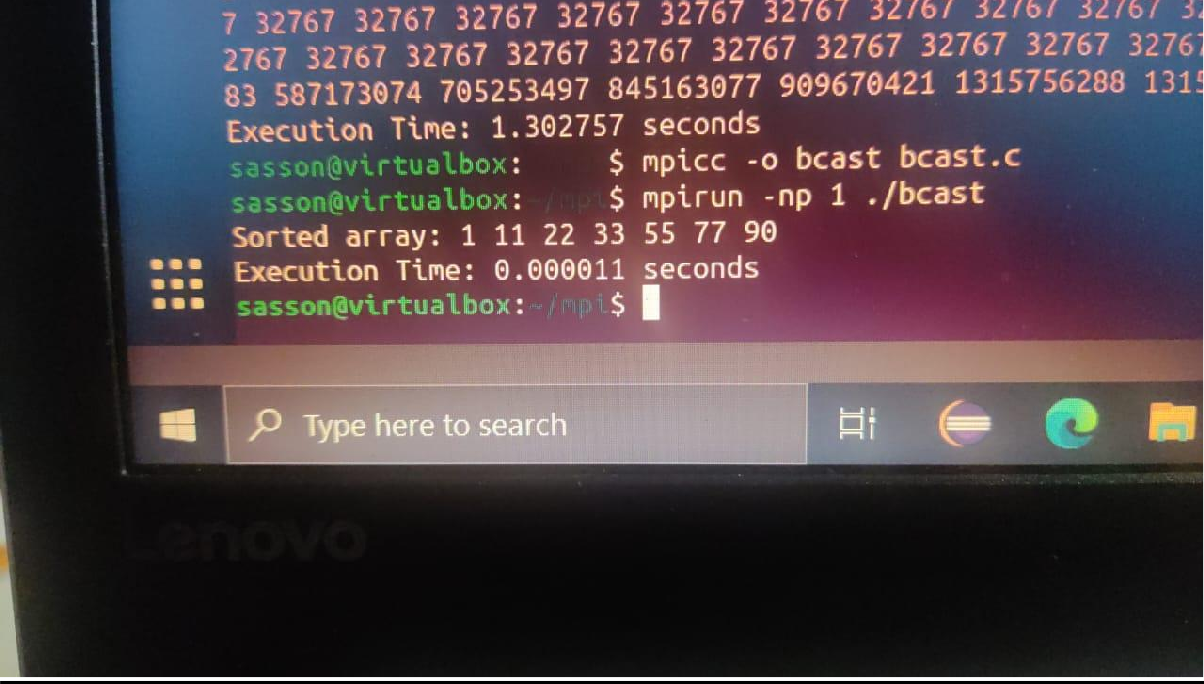
printf("\n");

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]);

}

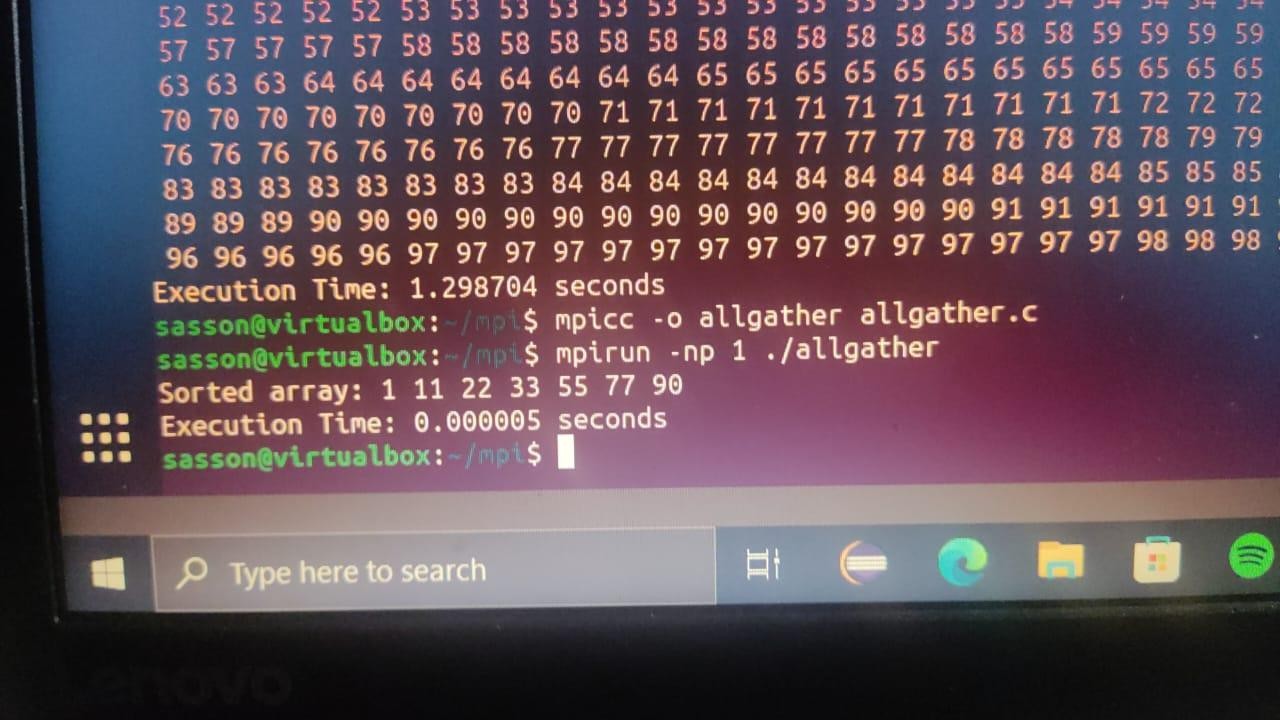
printf("\n");

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]);

}

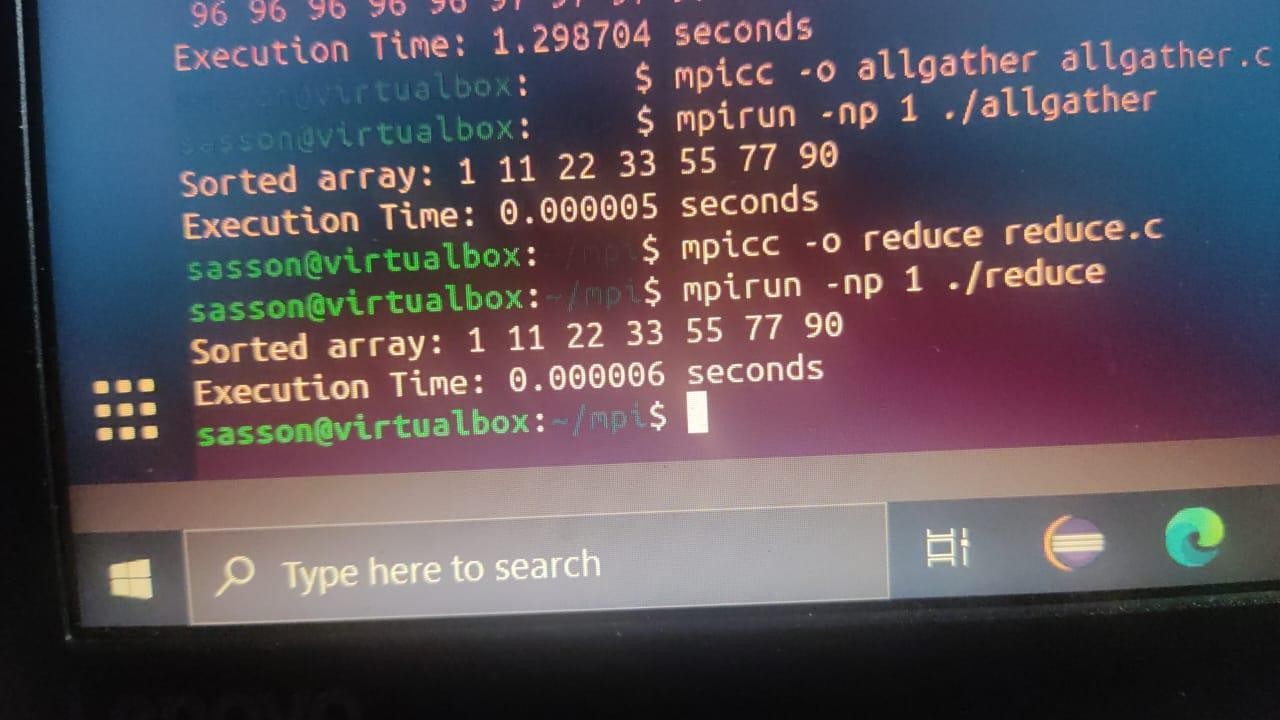
printf("\n");

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]);

}

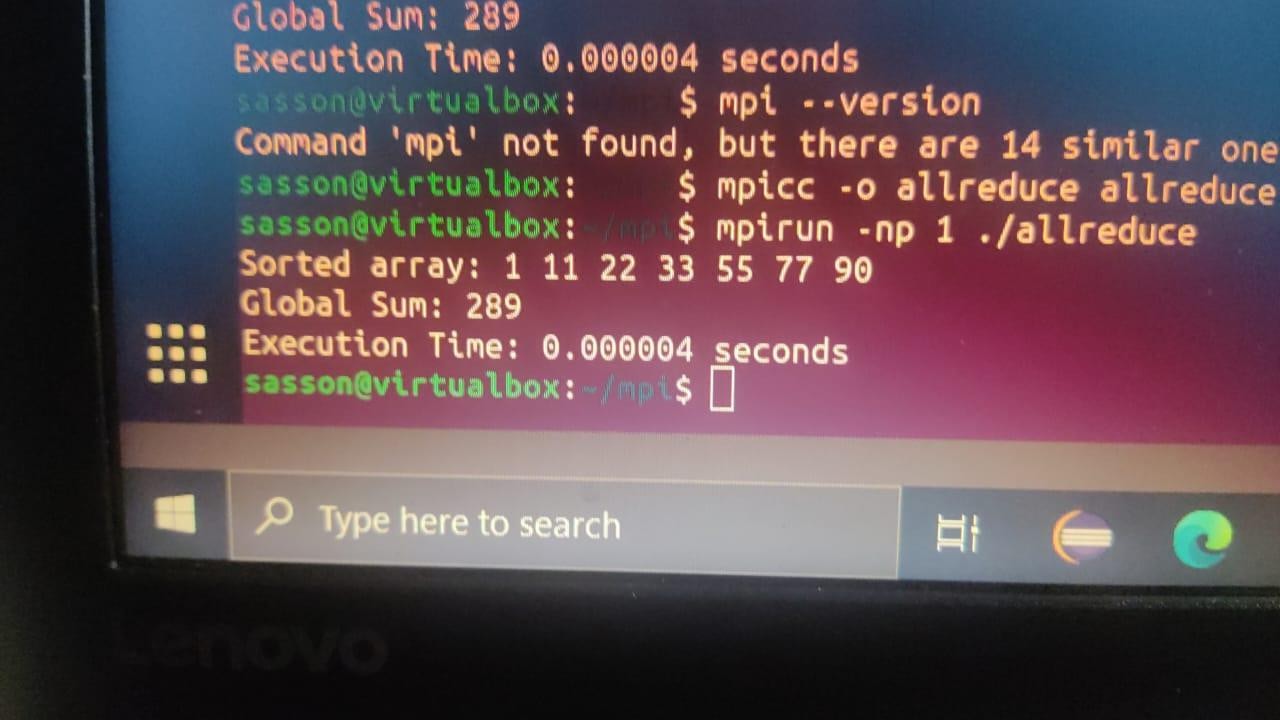
printf("\n");

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

}

MPI\_Finalize(); return 0;

}



# SUMMARY

The choice between MPI\_Bcast, MPI\_Scatter, MPI\_Gather, MPI\_Reduce, or MPI\_Allreduce depends on the specific requirements and characteristics of your MPI program. Each of these MPI collective operations serves a different purpose, and the "best" one depends on the communication pattern and the nature of the data distribution in your application.

## MPI\_Bcast:

* + **Use Case:** When you have a single source (usually the root process) broadcasting data to all other processes.
  + **Bubble Sort Consideration:** If the data to be sorted is the same across all processes, and you only need to distribute the initial array to all processes before sorting.

## MPI\_Scatter:

* + **Use Case:** When you want to distribute different chunks of an array to different processes.
  + **Bubble Sort Consideration:** If you want to split the initial array into chunks, send each chunk to different processes for sorting, and then gather the results.

## MPI\_Gather:

* + **Use Case:** When you want to collect data from different processes onto a single process (often the root process).
  + **Bubble Sort Consideration:** If you sort different chunks of the array on different processes and want to gather the sorted chunks back to a single process for further processing.

## MPI\_Reduce:

* + **Use Case:** When you want to perform a reduction operation (e.g., sum, max) across all processes, typically resulting in a single value.
  + **Bubble Sort Consideration:** If you are interested in a global reduction operation (e.g., finding the sum of all elements in the sorted array).

## MPI\_Allreduce:

* + **Use Case:** When you want to perform a reduction operation across all processes, and each process receives the result.
  + **Bubble Sort Consideration:** If you want all processes to have the result of a global reduction operation without the need for an additional gather step.

For a Bubble Sort program in MPI, the communication pattern is crucial. If you are distributing the data and performing independent local sorting, a combination of MPI\_Scatter and MPI\_Gather might be suitable. If you are interested in a global reduction operation, then MPI\_Allreduce or MPI\_Reduce might be appropriate.

# 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++) {

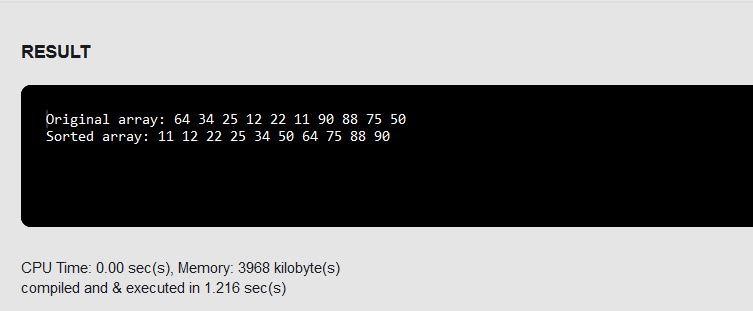
printf("%d ", arr[i]);

}

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:

* + In Code 1, the entire sorting operation is protected by a single critical section. This ensures that only one thread at a time can execute the sorting operation, preventing race conditions.
  + In Code 2, there is a nested parallel region with a critical section inside the inner loop. This can lead to inefficiencies and might not be necessary for this specific sorting algorithm.

critical section -is a region of code that must be executed by only one thread or process at a time, in order to prevent race conditions and ensure the integrity of the shared resource.

## Barrier in Code 2:

* + Code 2 includes #pragma omp barrier after each phase of the sorting algorithm. This is not present in Code 1. The barrier ensures that all threads complete their work in the current phase before proceeding to the next phase. However, using barriers in this way might limit parallelism.

## Private Variables:

* + Both codes use private variables (temp) to avoid data races. The use of private variables is essential to ensure correctness in a parallel setting.

**Recommendation:** It's generally more efficient to use a single critical section to protect the entire sorting operation (as in Code 1) rather than introducing nested parallelism and 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();

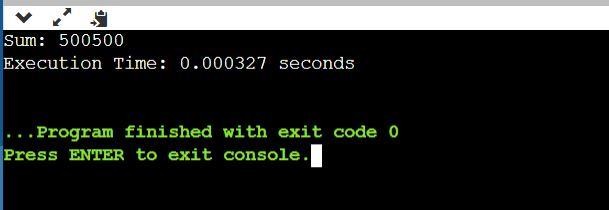
double execution\_time = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

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;

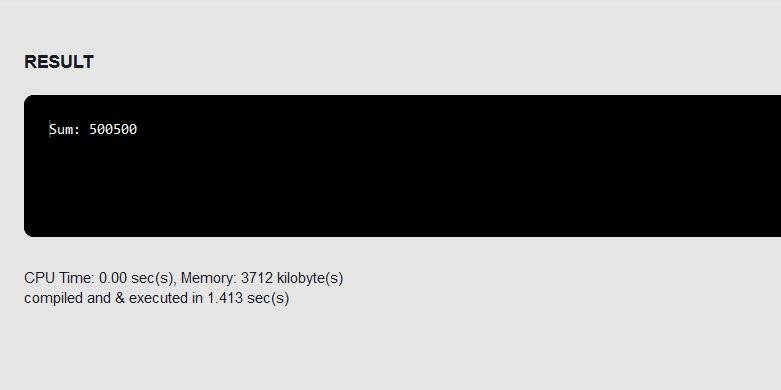
}

}

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

return 0;

}



## Replace Pthreads functions with OpenMP directives:

* + The pthread\_create and pthread\_join functions are replaced with OpenMP directives for parallel regions and loops (#pragma omp parallel and #pragma omp for).
  + The pthread\_exit call at the end of the Pthreads version is not needed in the OpenMP version.

## Private variables:

* + In the OpenMP version, the private variable t is explicitly declared within the parallel directive.

## Critical section:

* + The critical section, where the global sum is updated, is achieved using #pragma omp critical in OpenMP.