**Class:** Final Year (Computer Science and Engineering)

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

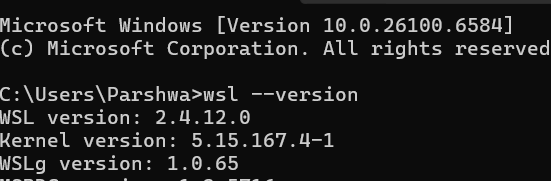
**Practical No. 6**

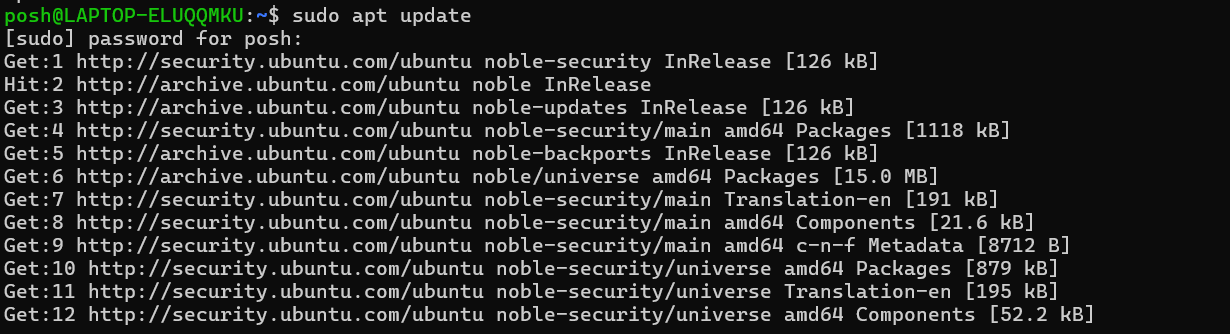
**Exam Seat No: 22510064 – Parshwa Herwade**

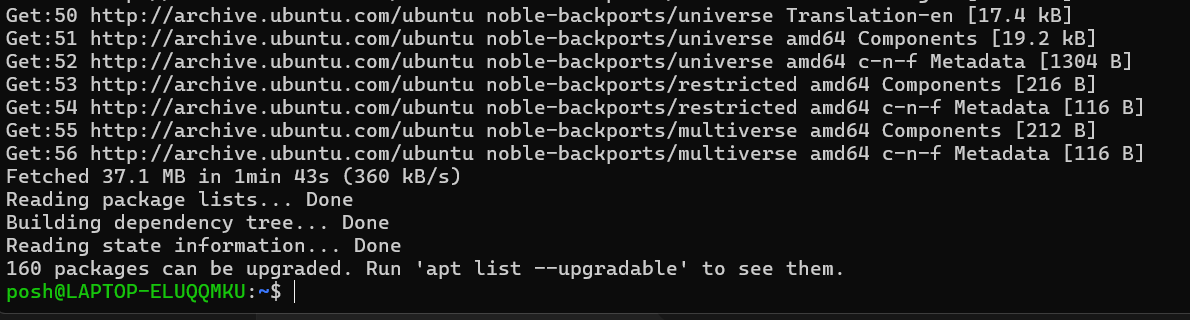
**Github Link:** [**Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub**](https://github.com/parshwa913/Sem-7-Assign/tree/main/HPC%20lab)

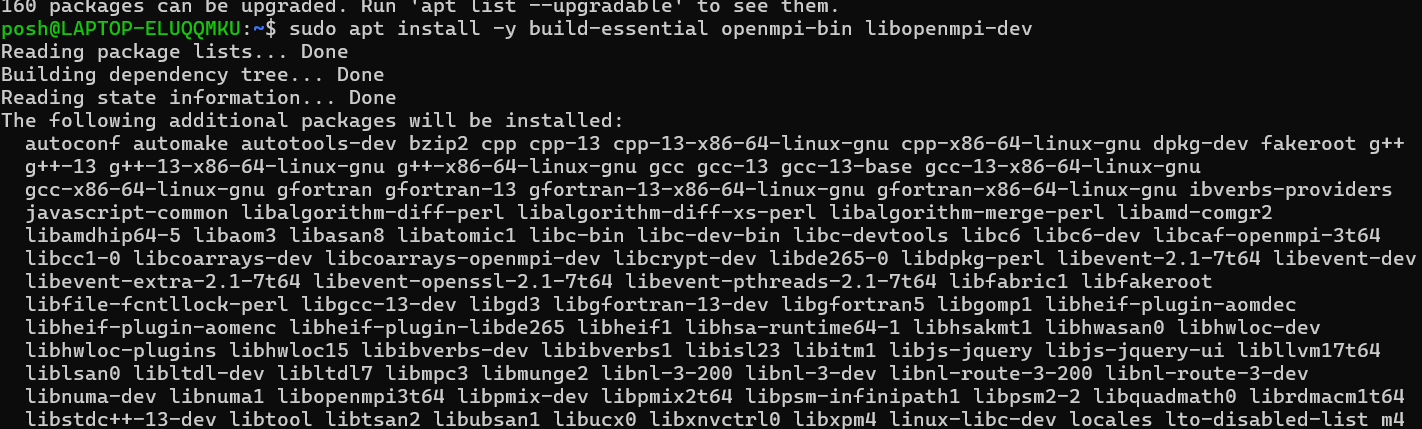
**Title of practical:**

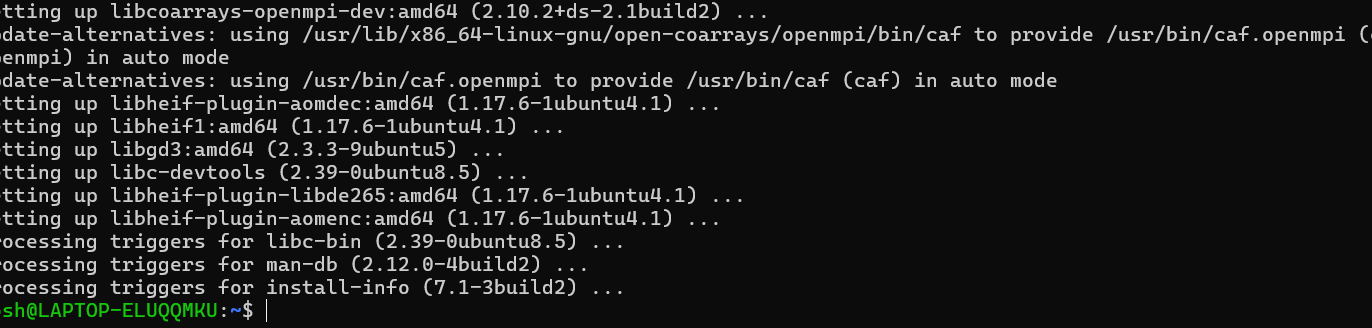
Installation of MPI & Implementation of basic functions of MPI

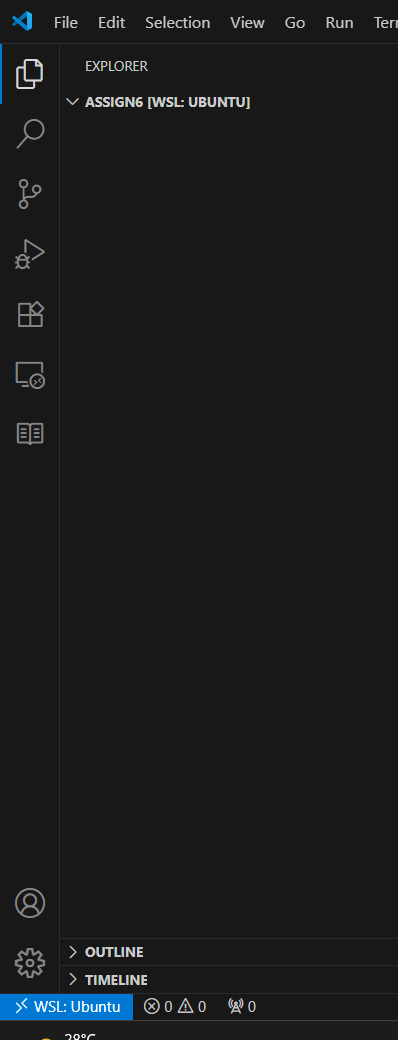
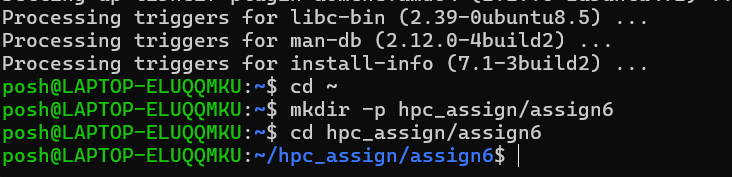
**ANSWER  
  
WSL already installed**











**Problem Statement 1:**

Implement a simple hello world program by setting number of processes equal to 10

#include <mpi.h>

#include <stdio.h>

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);

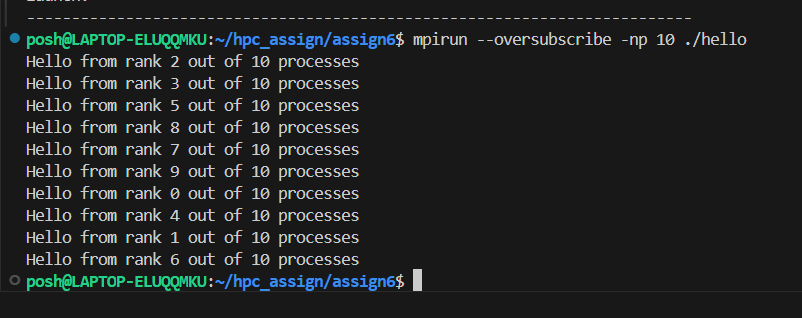
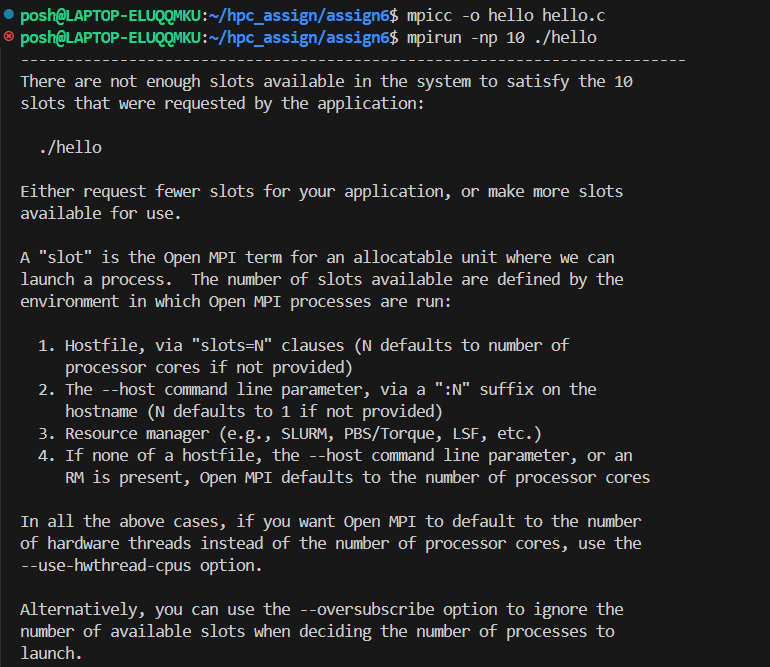
    printf("Hello from rank %d out of %d processes\n", rank, size);

    MPI\_Finalize();

    return 0;

}

**Screenshots:**



**Information 1:**

**Theory/Info:**

**Every MPI program begins with MPI\_Init and ends with MPI\_Finalize.**

**MPI\_Comm\_rank gives the rank of the process (unique ID).**

**MPI\_Comm\_size gives the total number of processes.**

**Purpose: To understand process identification in MPI and parallel execution.**

**Demonstration: Each process prints its rank and the total number of processes.**

**Key Concepts: Process initialization, basic communication, parallel output.**

**Conclusion:**

**Verified that multiple processes are launched and each process has a unique rank.**

**Problem Statement 2:**

Implement a program to display rank and communicator group of five processes

#include <mpi.h>

#include <stdio.h>

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

    MPI\_Init(&argc, &argv);

    int world\_rank, world\_size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

    int color = (world\_rank < (world\_size+1)/2) ? 0 : 1;

    MPI\_Comm newcomm;

    MPI\_Comm\_split(MPI\_COMM\_WORLD, color, world\_rank, &newcomm);

    int new\_rank, new\_size;

    MPI\_Comm\_rank(newcomm, &new\_rank);

    MPI\_Comm\_size(newcomm, &new\_size);

    printf("World rank %d/%d -> color=%d -> newcomm rank %d/%d\n",

           world\_rank, world\_size, color, new\_rank, new\_size);

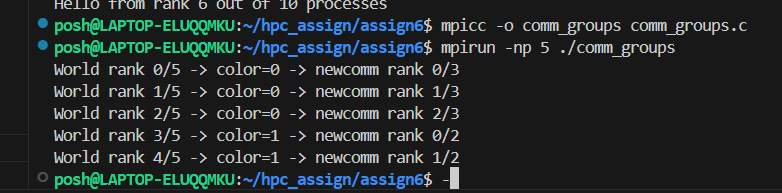
    MPI\_Comm\_free(&newcomm);

    MPI\_Finalize();

    return 0;

}

**Screenshots:**



**Topic:** Communicator Splitting and Subgroups  
**Theory/Info:**

* MPI allows splitting the global communicator (MPI\_COMM\_WORLD) into sub-communicators using MPI\_Comm\_split.
* Processes can be divided into groups based on rank, color, or other criteria.
* Purpose: Learn group communication and logical partitioning of processes.  
  **Demonstration:** Processes are divided into two groups; each process prints its original rank and its new rank in the subgroup.  
  **Key Concepts:** Process grouping, sub-communicators, parallel task partitioning.  
  **Conclusion:**
* Demonstrates how MPI sub-communicators can divide work among process groups.
* Helps design programs with group-specific communication patterns.

**Q3: Implement a MPI program to give an example of Deadlock.**

**Program and screenshots**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define BIGSIZE 10000000

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

    int rank, size;

    int \*buffer = NULL;

    MPI\_Init(&argc, &argv);

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

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

    if (size != 2) {

        if (rank == 0) {

            printf("Run this program with exactly 2 processes!\n");

        }

        MPI\_Finalize();

        return 0;

    }

    buffer = (int\*)malloc(BIGSIZE \* sizeof(int));

    if (rank == 0) {

        MPI\_Send(buffer, BIGSIZE, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

        MPI\_Recv(buffer, BIGSIZE, MPI\_INT, 1, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Process 0 finished exchange\n");

    } else if (rank == 1) {

        MPI\_Send(buffer, BIGSIZE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

        MPI\_Recv(buffer, BIGSIZE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Process 1 finished exchange\n");

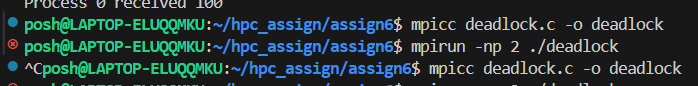
    }

    free(buffer);

    MPI\_Finalize();

    return 0;

}

****

**Process didn’t complete   
Forced to stop the code**

**DEADLOCK**

**FIXED CODE:**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define BIGSIZE 10000000   // 10 million ints ~ 40 MB

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

    int rank, size;

    int \*sendbuf = NULL;

    int \*recvbuf = NULL;

    MPI\_Init(&argc, &argv);

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

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

    if (size != 2) {

        if (rank == 0) {

            printf("Run this program with exactly 2 processes!\n");

        }

        MPI\_Finalize();

        return 0;

    }

    sendbuf = (int\*)malloc(BIGSIZE \* sizeof(int));

    recvbuf = (int\*)malloc(BIGSIZE \* sizeof(int));

    for (int i = 0; i < BIGSIZE; i++) {

        sendbuf[i] = rank;   // fill with process rank

    }

    // Safe exchange using MPI\_Sendrecv

    MPI\_Sendrecv(sendbuf, BIGSIZE, MPI\_INT, 1 - rank, 0,

                 recvbuf, BIGSIZE, MPI\_INT, 1 - rank, 0,

                 MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

    printf("Process %d successfully exchanged data with process %d\n",

           rank, 1 - rank);

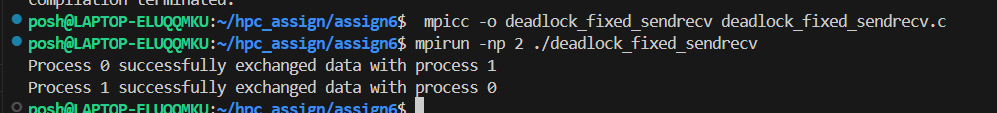
    free(sendbuf);

    free(recvbuf);

    MPI\_Finalize();

    return 0;

}

****

**Theory/Info:**

* **A deadlock occurs when two or more processes wait indefinitely for each other to send/receive messages.**
* **Using blocking sends (MPI\_Send), if all processes try to send simultaneously without a matching receive, the program hangs.**
* **Purpose: Learn the importance of ordering communication operations to avoid deadlocks.  
  Demonstration: Processes attempting to send large messages to each other simultaneously will hang.  
  Key Concepts: Deadlock conditions, blocking communication, synchronization issues.  
  Conclusion:**
* **Demonstrates the consequences of improper communication ordering.**
* **Highlights the need for careful design in parallel programs to avoid deadlocks.**

**Topic: MPI\_Sendrecv  
Theory/Info:**

* **MPI\_Sendrecv allows a process to simultaneously send and receive a message safely.**
* **This prevents deadlocks because each process posts a send and receive in one atomic call.**
* **Purpose: Learn safe data exchange patterns in MPI to avoid deadlocks.  
  Demonstration: Two processes exchange messages successfully and print confirmation.  
  Key Concepts: Synchronous communication, deadlock prevention, atomic send-receive.  
  Conclusion:**
* **Confirms that deadlocks can be prevented using proper MPI functions.**
* **Demonstrates reliable communication even with simultaneous data exchange.**

**Q4. Implement blocking MPI send & receive to demonstrate Nearest neighbor exchange of data in a ring topology.**

**Program and screenshots**

#include <mpi.h>

#include <stdio.h>

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 my\_value;

    if (rank == 0) {

        printf("Enter an integer value (rank 0): ");

        fflush(stdout);

        scanf("%d", &my\_value);

    }

    MPI\_Bcast(&my\_value, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    int left = (rank - 1 + size) % size;

    int right = (rank + 1) % size;

    int recv\_value = -1;

    if (rank % 2 == 0) {

        MPI\_Send(&my\_value, 1, MPI\_INT, right, 0, MPI\_COMM\_WORLD);

        MPI\_Recv(&recv\_value, 1, MPI\_INT, left, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

    } else {

        MPI\_Recv(&recv\_value, 1, MPI\_INT, left, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Send(&my\_value, 1, MPI\_INT, right, 0, MPI\_COMM\_WORLD);

    }

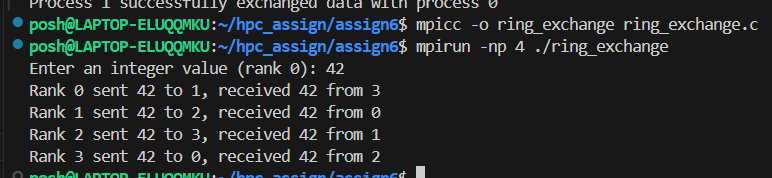
    printf("Rank %d sent %d to %d, received %d from %d\n",

           rank, my\_value, right, recv\_value, left);

    MPI\_Finalize();

    return 0;

}

****

**Theory/Info:**

* **In a ring topology, each process communicates only with its neighbors (left and right).**
* **MPI allows designing custom topologies for structured communication.**
* **Purpose: Understand neighbor communication patterns, blocking sends, and data propagation in a ring.  
  Demonstration: Each process sends its data to the next process and receives from the previous one; output shows correct exchange.  
  Key Concepts: Process topologies, nearest neighbor communication, synchronization in a structured network.  
  Conclusion:**
* **Validates communication in a ring structure is correctly implemented.**
* **Useful for algorithms like token passing, pipeline computations, or distributed workflows.**

**Q5. Write a MPI program to find the sum of all the elements of an array A of size**

**n. Elements of an array can be divided into two equals groups. The first [n/2]**

**elements are added by the first process, P0, and last [n/2] elements the by second process, P1. The two sums then are added to get the final result.**

**Program and screenshots**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

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);

    if (size != 2) {

        if (rank == 0) printf("Run this program with exactly 2 processes!\n");

        MPI\_Finalize();

        return 0;

    }

    int n;

    int \*arr = NULL;

    if (rank == 0) {

        printf("Enter total number of elements: ");

        fflush(stdout);

        scanf("%d", &n);

        arr = (int\*)malloc(n \* sizeof(int));

        printf("Enter %d integers: ", n);

        for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    }

    MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    int half = n / 2;

    int remaining = n - half;

    int local\_sum = 0;

    if (rank == 0) {

        for (int i = 0; i < half; i++) local\_sum += arr[i];

        MPI\_Send(arr + half, remaining, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

        int sum1;

        MPI\_Recv(&sum1, 1, MPI\_INT, 1, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Final sum = %d (P0=%d + P1=%d)\n", local\_sum + sum1, local\_sum, sum1);

        free(arr);

    } else if (rank == 1) {

        int \*subarr = (int\*)malloc(remaining \* sizeof(int));

        MPI\_Recv(subarr, remaining, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        for (int i = 0; i < remaining; i++) local\_sum += subarr[i];

        MPI\_Send(&local\_sum, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

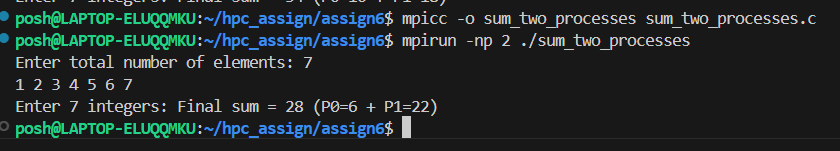
        free(subarr);

    }

    MPI\_Finalize();

    return 0;

}

****

**Theory/Info:**

* Large arrays can be split across multiple processes for parallel computation.
* Each process computes a partial sum and sends it to a master process to compute the final sum.
* Purpose: Learn data partitioning, communication of partial results, and parallel reduction.  
  **Demonstration:**
  + Process 0 sums the first half of the array.
  + Process 1 sums the second half.
  + Partial sums are combined at process 0 to get the total.  
    **Key Concepts:** Data parallelism, reduction operations, inter-process communication.  
    **Conclusion:**
* Shows the effectiveness of parallel computation in reducing workload.
* Confirms correct summation of array elements across multiple processes.