```cpp
/*6. Assume you have n robots which pick mangoes in a farm.
WAPT calculate the total number of mangoes picked by n robots parallely using MPI.*/

#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);         // Initialize MPI environment

    // Get number of processes and rank of current process
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Initialize random seed and generate random number of mangoes
    srand(static_cast<unsigned>(time(0)) + world_rank);   //srand(world_rank);  //ensures each process generates a diff random no. //or, srand((unsigned int)time(0) + world_rank);
    int mangoes_picked = rand() % 101;  //random number between 0 and 100, representing the number of mangoes picked by the robot

    cout << "Robot " << world_rank << " picked " << mangoes_picked << " mangoes." << endl;

    // Use MPI_Reduce to calculate the total mangoes picked
    int total_mangoes = 0;
    MPI_Reduce(&mangoes_picked, &total_mangoes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Display the total mangoes picked by all robots (only on root process)
    if (world_rank == 0)
        cout << "Total mangoes picked by all robots: " << total_mangoes << endl;

    MPI_Finalize();
    return 0;
}

/*
Robot 2 picked 25 mangoes.
Robot 4 picked 6 mangoes.
Robot 0 picked 15 mangoes.
Robot 1 picked 29 mangoes.
Robot 3 picked 49 mangoes.
Total mangoes picked by all robots: 124
*/


_______________________________________________________________________________________


/*7. Design a program that implements application of
```

```
 MPI Collective Communications.*/

#include <iostream>
#include <mpi.h>
#include <vector>
#include <cstdlib>

using namespace std;

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

    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int n = 10;
    vector<int> local_array(n);
    int local_sum = 0, total_sum = 0;

    srand(static_cast<unsigned>(time(0)) + world_rank);
    for (int i = 0; i < n; ++i) {
        local_array[i] = rand() % 100;
        local_sum += local_array[i];
    }
    cout << "Process " << world_rank << " local array: ";
    for (int i : local_array) cout << i << " ";
    cout << "\nProcess " << world_rank << " local sum: " << local_sum << endl;

    MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Bcast(&total_sum, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (world_rank == 0) {
        double average = static_cast<double>(total_sum) / (n * world_size);
        cout << "Total sum: " << total_sum << endl;
        cout << "Average: " << average << endl;
    }
    MPI_Finalize();
    return 0;
}

/*
$ gedit akhpc7.cpp
$ mpic++ akhpc7.cpp
$ mpirun -np 5 ./a.out

Process 0 local array: 44 81 71 89 74 44 98 41 1 5
Process 0 local sum: 548
Process 3 local array: 62 42 67 68 15 31 40 14 96 2
Process 3 local sum: 437
Process 4 local array: 36 65 76 67 42 93 64 14 68 25
Process 4 local sum: 550
Process 2 local array: 36 8 70 74 74 29 58 11 69 79
Process 2 local sum: 508
Process 1 local array: 13 99 77 4 54 82 64 48 5 52
Process 1 local sum: 498
Total sum: 2541
Average: 50.82
*/

_________________________________________________________________________________________


// 8. Implement Cartesian Virtual Topology in MPI.

#include <iostream>
#include <mpi.h>
#include <vector>
using namespace std;

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

    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int dims[2] = {0, 0};
    MPI_Dims_create(world_size, 2, dims);

    MPI_Comm cart_comm;
    int period[2] = {0, 0}; // Periodicity in both dimensions
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, period, true, &cart_comm);

    int coords[2];
    MPI_Comm_rank(cart_comm, &world_rank);     ///////can comment this lineee
    MPI_Cart_coords(cart_comm, world_rank, 2, coords);

    int north, south, east, west;
    MPI_Cart_shift(cart_comm, 0, 1, &north, &south);
    MPI_Cart_shift(cart_comm, 1, 1, &west, &east);

    int value = world_rank;
    cout << "Process " << world_rank << " at (" << coords[0] << ", " << coords[1] << ") has value: " << value << endl;

    if (north != MPI_PROC_NULL) MPI_Send(&value, 1, MPI_INT, north, 0, cart_comm);
    if (south != MPI_PROC_NULL) MPI_Recv(&value, 1, MPI_INT, south, 0, cart_comm, MPI_STATUS_IGNORE);
    if (west != MPI_PROC_NULL) MPI_Send(&value, 1, MPI_INT, west, 0, cart_comm);
    if (east != MPI_PROC_NULL) MPI_Recv(&value, 1, MPI_INT, east, 0, cart_comm, MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
```

```
}

/*
$ gedit akhpc8.cpp
$ mpic++ akhpc8.cpp
$ mpirun -np 4 ./a.out

Process 0 at (0, 0) has value: 0
Process 1 at (0, 1) has value: 1
Process 2 at (1, 0) has value: 2
Process 3 at (1, 1) has value: 3
*/

___________________________________________________________________________________________


/*9. Design a MPI program that uses blocking send/receive
routines and non blocking send/receive routines.*/

#include <iostream>
#include <mpi.h>
#include <vector>

using namespace std;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    const int TAG = 0;
    const int DATA_SIZE = 10;
    vector<int> send_data(DATA_SIZE, world_rank);
    vector<int> recv_data(DATA_SIZE);

    if (world_rank == 0) {
        cout << "Process 0 sending data: ";
        for (int i : send_data) cout << i << " ";
        cout << endl;
        MPI_Send(send_data.data(), DATA_SIZE, MPI_INT, 1, TAG, MPI_COMM_WORLD);
    }
    else if (world_rank == 1) {
        MPI_Recv(recv_data.data(), DATA_SIZE, MPI_INT, 0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        cout << "Process 1 received data: ";
        for (int i : recv_data) cout << i << " ";
        cout << endl;
    }

    MPI_Request send_request, recv_request;

```

```cpp
    if (world_rank == 0) {
        MPI_Isend(send_data.data(), DATA_SIZE, MPI_INT, 1, TAG, MPI_COMM_WORLD, &send_request);
        cout << "Process 0 non-blocking send initiated." << endl;
    }
    else if (world_rank == 1) {
        MPI_Irecv(recv_data.data(), DATA_SIZE, MPI_INT, 0, TAG, MPI_COMM_WORLD, &recv_request);
        cout << "Process 1 non-blocking receive initiated." << endl;
    }

    if (world_rank == 0) {
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
        cout << "Process 0 non-blocking send completed." << endl;
    }
    else if (world_rank == 1) {
        MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
        cout << "Process 1 non-blocking receive completed: ";
        for (int i : recv_data) cout << i << " ";
        cout << endl;
    }

    MPI_Finalize();
    return 0;
}

/*
$ mpic++ akhpc9.cpp
$ mpirun -np 9 ./a.out

Process 0 sending data: 0 0 0 0 0 0 0 0 0 0
Process 0 non-blocking send initiated.
Process 0 non-blocking send completed.
Process 1 received data: 0 0 0 0 0 0 0 0 0 0
Process 1 non-blocking receive initiated.
Process 1 non-blocking receive completed: 0 0 0 0 0 0 0 0 0 0
*/

_________________________________________________________________________________________



/* 10. Multiply two square matrices (1000,2000 or 3000 dimensions). Compare
the performance of a sequential and parallel algorithm using open MP.*/

#include <iostream>
#include <vector>
#include <omp.h>  // OpenMP
#include <ctime>

using namespace std;

// Function to multiply two matrices sequentially
```

```cpp
void multiplySequential(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Function to multiply two matrices using OpenMP
void multiplyParallel(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C, int N) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int N; // Matrix size (1000, 2000, 3000)
    cout << "Enter matrix size (e.g., 1000, 2000, 3000): ";
    cin >> N;
    clock_t start, end;

    // Initialize matrices A, B, and C
    vector<vector<int>> A(N, vector<int>(N, 1)); // Matrix A with all elements = 1
    vector<vector<int>> B(N, vector<int>(N, 1)); // Matrix B with all elements = 1
    vector<vector<int>> C(N, vector<int>(N, 0)); // Matrix C to store result

    // Sequential multiplication
    start = clock();           // double start = omp_get_wtime();
    multiplySequential(A, B, C, N);
    end = clock();             // double end = omp_get_wtime();
    double durationSeq = double(end - start) / CLOCKS_PER_SEC;  // double durationSeq = end - start;
    cout << "Time taken for sequential multiplication: " << durationSeq << " seconds" << endl;

    // Parallel multiplication using OpenMP
    start = clock();                   //start = omp_get_wtime();
    multiplyParallel(A, B, C, N);
    end = clock();                     //end = omp_get_wtime();
    double durationPar = double(end - start) / CLOCKS_PER_SEC; //double durationPar = end - start;
    cout << "Time taken for parallel multiplication (OpenMP): " << durationPar << " seconds" << endl;

    return 0;
}
```

```

/*
//OUTPUT:
Enter matrix size (e.g., 1000, 2000, 3000): 1000
Time taken for sequential multiplication: 6.82819 seconds
Time taken for parallel multiplication (OpenMP): 0.62969 seconds
*/
```