1. WAPP in C/C++ using MPI to compute the sum of all even numbers in an array using scatter and gather operation.

```
#include <mpi.h>
#include <stdio.h>

int is_even(int x) {
    return x % 2 == 0;
}

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get process ID
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get total number of processes

    const int N = 16;                    // Total array size (divisible by size)
    int data[N], local[N / size], local_even_sum = 0;

    if (rank == 0) {
        // Initialize array at root
        printf("Original array:\n");
        for (int i = 0; i < N; i++) {
            data[i] = i + 1;
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    // Scatter data from root to all processes
    MPI_Scatter(data, N / size, MPI_INT, local, N / size, MPI_INT, 0, MPI_COMM_WORLD);
```

```
    // Each process calculates sum of even numbers in its chunk
    for (int i = 0; i < N / size; i++) {
        if (is_even(local[i])) {
            local_even_sum += local[i];
        }
    }

    // Gather local sums at root process
    int gathered_sums[size];
    MPI_Gather(&local_even_sum, 1, MPI_INT, gathered_sums, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        int total_even_sum = 0;
        for (int i = 0; i < size; i++) {
            total_even_sum += gathered_sums[i];
        }
        printf("Sum of even numbers: %d\n", total_even_sum);
    }

    MPI_Finalize();
    return 0;
}
```

✅ Example Output (N = 16)

```
Original array:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Sum of even numbers: 72
```

2 WAPP in C/C++ using MP1 to implement the linear search algorithm.

```
#include<stdio.h>
#include<mpi.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);

   const int N=8;
   int data[8]={2,4,5,6,7,8,12,15};
   int key=12;

   int chunk=N/size;
   int subdata[chunk];

   MPI_Scatter(data, chunk, MPI_INT, subdata, chunk, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&key, 1, MPI_INT, 0, MPI_COMM_WORLD);

   int local_found=0;
   for(int i=0; i<chunk; i++)
   {
      if(subdata[i]==key)
      {
         printf("Key %d is found at %d by processor %d\n", key, rank*chunk+1, rank);
         local_found=1;
      }
   }
```

```
    int found=0;
    MPI_Reduce(&local_found, &found, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0 && found == 0)
    printf("Key %d was not found in any process.\n", key);

    MPI_Finalize();
    return 0;
}
```

Compiling

Compilation is OK

Execution ...

Key 12 is found at 7 by processor 3

3. WAPP in C/C++ using MPI to add two compatible matrices.

```
#include <stdio.h>
#include <mpi.h>

#define N 4  // Rows
#define M 4  // Columns

int main(int argc, char *argv[]) {
   int rank, size;
   MPI_Init(&argc, &argv);                    // Initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);        // Get current process rank
   MPI_Comm_size(MPI_COMM_WORLD, &size);        // Get number of processes

   if (N % size != 0) {
      if (rank == 0)
         printf("Matrix row count (%d) must be divisible by number of processes (%d)\n", N, size);
      MPI_Finalize();
      return 1;
   }

   int rows_per_proc = N / size;

   int A[N][M], B[N][M], C[N][M];          // Full matrices (used only by rank 0)
   int local_A[rows_per_proc][M];          // Local submatrices
   int local_B[rows_per_proc][M];
   int local_C[rows_per_proc][M];

   // Initialize matrices A and B at root
   if (rank == 0) {
      printf("Matrix A:\n");
      for (int i = 0; i < N; i++) {
```

```
        for (int j = 0; j < M; j++) {
            A[i][j] = i + j;
            B[i][j] = (i + 1) * (j + 1);
            printf("%3d ", A[i][j]);
        }
        printf("\n");
    }

    printf("\nMatrix B:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            printf("%3d ", B[i][j]);
        }
        printf("\n");
    }
}

// Scatter rows of A and B to all processes
MPI_Scatter(A, rows_per_proc * M, MPI_INT,
            local_A, rows_per_proc * M, MPI_INT,
            0, MPI_COMM_WORLD);

MPI_Scatter(B, rows_per_proc * M, MPI_INT,
            local_B, rows_per_proc * M, MPI_INT,
            0, MPI_COMM_WORLD);

// Local computation: local_C = local_A + local_B
for (int i = 0; i < rows_per_proc; i++) {
    for (int j = 0; j < M; j++) {
        local_C[i][j] = local_A[i][j] + local_B[i][j];
    }
```

```
    }

    // Gather local_C results back to C in root
    MPI_Gather(local_C, rows_per_proc * M, MPI_INT,
               C, rows_per_proc * M, MPI_INT,
               0, MPI_COMM_WORLD);

    // Display result matrix at root
    if (rank == 0) {
        printf("\nMatrix C = A + B:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                printf("%3d ", C[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();  // Finalize MPI
    return 0;
}
```

4. WAPP in C/C++ using MPI to compute the value of PI.

```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>


static inline double rand01(unsigned int *seed)

/* Fast re-entrant RNG: returns uniform double in [0,1) */

{

   return rand_r(seed) / (double)RAND_MAX;

}


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

{

   MPI_Init(&argc, &argv);


   int rank, size;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // my rank

   MPI_Comm_size(MPI_COMM_WORLD, &size);   // #processes


   long long N_total = (argc > 1) ? atoll(argv[1]) : 1000000LL;

   /* Optionally broadcast N_total from rank 0 so all ranks agree */

   MPI_Bcast(&N_total, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);


   /* Divide work: last rank gets the remainder */

   long long N_local = N_total / size;

   if (rank == size - 1) N_local += N_total % size;


   unsigned int seed = (unsigned int)time(NULL) ^ (rank * 0x9e3779b9U);


   long long local_hits = 0;
```

```
    for (long long i = 0; i < N_local; ++i) {
        double x = rand01(&seed);
        double y = rand01(&seed);
        if (x * x + y * y <= 1.0) ++local_hits;
    }

    long long total_hits = 0;
    MPI_Reduce(&local_hits, &total_hits, 1, MPI_LONG_LONG,
               MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_est = 4.0 * (double)total_hits / (double)N_total;
        printf("π ≈ %.12f   (samples = %lld, processes = %d)\n",
               pi_est, N_total, size);
    }

    MPI_Finalize();
    return 0;
}
```

5. WAPP in C/C++ using MPI to compute the sum of all even numbers in an array using scatter and gather operation.

```
#include <stdio.h>

#include <mpi.h>


int is_even(int x) {

   return x % 2 == 0;

}


int main(int argc, char **argv) {

   int rank, size;


   MPI_Init(&argc, &argv);                    // Step 1: Initialize MPI

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);        // Step 2: Get rank

   MPI_Comm_size(MPI_COMM_WORLD, &size);        // Step 3: Get size


   const int N = 16; // Total number of elements (must be divisible by size)

   int data[N];

   int local_data[N / size];


   if (rank == 0) {

      // Initialize array with some numbers

      for (int i = 0; i < N; i++)

         data[i] = i + 1; // Example: 1 to 16

   }


   // Step 4: Scatter the array

   MPI_Scatter(data, N / size, MPI_INT, local_data, N / size, MPI_INT, 0, MPI_COMM_WORLD);


   // Step 5: Compute local sum of even numbers

   int local_even_sum = 0;
```

```
    for (int i = 0; i < N / size; i++) {
        if (is_even(local_data[i]))
            local_even_sum += local_data[i];
    }

    // Step 6: Gather local even sums
    int all_even_sums[size];
    MPI_Gather(&local_even_sum, 1, MPI_INT, all_even_sums, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Step 7: Root calculates final sum
    if (rank == 0) {
        int total_even_sum = 0;
        for (int i = 0; i < size; i++)
            total_even_sum += all_even_sums[i];

        printf("Total sum of even numbers in the array: %d\n", total_even_sum);
    }

    MPI_Finalize(); // Step 8: Finalize MPI
    return 0;
}
```

6. WAPP in C/C++ using MPI to compute find the largest element in the array using broadcast and reduction paradigm.

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

int main(int argc, char *argv[]) {
    int rank, size;
    const int N = 16;
    int data[N];
```

```
int local_max = -1;
int global_max;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Step 1: Initialize the array in root process
if (rank == 0) {
    printf("Input array: ");
    for (int i = 0; i < N; i++) {
        data[i] = rand() % 100;
        printf("%d ", data[i]);
    }
    printf("\n");
}

// Step 2: Broadcast the full array to all processes
MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);

// Step 3: Each process works on a segment
int chunk_size = N / size;
int start = rank * chunk_size;
int end = start + chunk_size;

local_max = data[start];
for (int i = start + 1; i < end; i++) {
    if (data[i] > local_max) {
        local_max = data[i];
    }
}
```

```
    // Step 4: Reduce all local maxima to find the global max
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    // Step 5: Root process prints the result
    if (rank == 0) {
        printf("Largest element in array: %d\n", global_max);
    }

    MPI_Finalize();
    return 0;
}
```

7. WAPP in C/C++ using MPI to compute the product of two compatible matrices.

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

#define M 4  // Rows of A and C
#define N 4  // Columns of A and Rows of B
#define P 4  // Columns of B and C

int main(int argc, char *argv[]) {
    int rank, size;
    int A[M][N], B[N][P], C[M][P];
    int local_A[M/4][N], local_C[M/4][P];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (M % size != 0) {
        if (rank == 0)
            printf("Matrix row size M must be divisible by number of processes.\n");
        MPI_Finalize();
        return -1;
    }

    int rows_per_process = M / size;

    // Initialize matrices A and B in the root process
    if (rank == 0) {
        printf("Matrix A:\n");
        for (int i = 0; i < M; i++) {
```

```
            for (int j = 0; j < N; j++) {
                A[i][j] = i + j;
                printf("%3d ", A[i][j]);
            }
            printf("\n");
        }

        printf("Matrix B:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < P; j++) {
                B[i][j] = i * j;
                printf("%3d ", B[i][j]);
            }
            printf("\n");
        }
    }

    // Scatter rows of A
    MPI_Scatter(A, rows_per_process * N, MPI_INT, local_A, rows_per_process * N, MPI_INT, 0,
MPI_COMM_WORLD);

    // Broadcast matrix B to all processes
    MPI_Bcast(B, N * P, MPI_INT, 0, MPI_COMM_WORLD);

    // Local computation of matrix multiplication
    for (int i = 0; i < rows_per_process; i++) {
        for (int j = 0; j < P; j++) {
            local_C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                local_C[i][j] += local_A[i][k] * B[k][j];
            }
```

```
        }
    }

    // Gather results into matrix C
    MPI_Gather(local_C, rows_per_process * P, MPI_INT, C, rows_per_process * P, MPI_INT, 0,
MPI_COMM_WORLD);

    // Display result in root
    if (rank == 0) {
        printf("Matrix C (Product):\n");
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < P; j++) {
                printf("%4d ", C[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

8. WAPP in C/C++ using MPI to compute the sum of elements in an array using scatter and reduction operation.

```c
#include <mpi.h>

#include <stdio.h>


#define N 16  // Total number of elements (should be divisible by number of processes)


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

   int rank, size;

   int data[N];        // Full array (used only by root)

   int local[N];       // Over-allocated to maximum possible size


   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Comm_size(MPI_COMM_WORLD, &size);


   if (N % size != 0 && rank == 0) {

      printf("Error: Array size N = %d is not divisible by number of processes = %d\n", N, size);

      MPI_Abort(MPI_COMM_WORLD, 1);

   }


   int local_n = N / size;


   // Initialize array only on root process

   if (rank == 0) {

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

         data[i] = i + 1;  // Example: 1, 2, ..., N

      }

   }


   // Scatter the array to all processes
```

```
MPI_Scatter(
    data,        // send buffer (root only)
    local_n,     // number of elements sent to each process
    MPI_INT,
    local,       // receive buffer (each process)
    local_n,     // number of elements to receive
    MPI_INT,
    0,           // root
    MPI_COMM_WORLD
);
// Each process calculates the sum of its local part
int local_sum = 0;
for (int i = 0; i < local_n; i++) {
    local_sum += local[i];
}
// Reduce all local sums to a single global sum at root
int global_sum = 0;
MPI_Reduce(
    &local_sum,     // send buffer
    &global_sum,    // receive buffer (only on root)
    1,
    MPI_INT,
    MPI_SUM,
    0,
    MPI_COMM_WORLD
);
// Print the result at root
if (rank == 0) {
    printf("Sum of array elements = %d\n", global_sum);
}
MPI_Finalize(); return 0; }
```

9. WAPP in C/C++ using MPI to compute the sum of all prime numbers in an array using broadcast.

```
#include <mpi.h>

#include <stdio.h>

#include <math.h>


int is_prime(int num) {

   if (num < 2) return 0;

   for (int i = 2; i <= sqrt(num); i++)

      if (num % i == 0) return 0;

   return 1;

}


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

   int rank, size;

   const int N = 16;

   int data[N] = {5, 7, 8, 9, 11, 13, 15, 17, 4, 6, 19, 23, 28, 29, 31, 33};


   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Comm_size(MPI_COMM_WORLD, &size);


   // Broadcast the entire array to all processes

   MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);


   // Determine local portion to process

   int local_sum = 0;

   for (int i = rank; i < N; i += size) {

      if (is_prime(data[i]))

         local_sum += data[i];

   }
```

```
    int global_sum = 0;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
        printf("Sum of prime numbers = %d\n", global_sum);

    MPI_Finalize();
    return 0;
}
```

10. WAPP in C/C++ using MPI to find and print the primes numbers in an array

```c
#include <mpi.h>
#include <stdio.h>

int is_prime(int n) {
   if (n <= 1) return 0;
   for (int i = 2; i * i <= n; ++i)
      if (n % i == 0) return 0;
   return 1;
}

int main(int argc, char *argv[]) {
   int rank, size;
   const int N = 16;
   int data[N] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17};

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int chunk = N / size;
   int local[chunk];

   MPI_Scatter(data, chunk, MPI_INT, local, chunk, MPI_INT, 0, MPI_COMM_WORLD);

   int local_primes[chunk];
   int local_count = 0;

   for (int i = 0; i < chunk; i++) {
      if (is_prime(local[i])) {
         local_primes[local_count++] = local[i];
```

```
    }
}

int recv_counts[size];    // Number of primes from each process
MPI_Gather(&local_count, 1, MPI_INT, recv_counts, 1, MPI_INT, 0, MPI_COMM_WORLD);

int displs[size], total = 0;
if (rank == 0) {
    displs[0] = 0;
    for (int i = 1; i < size; i++) {
        displs[i] = displs[i - 1] + recv_counts[i - 1];
    }
    total = displs[size - 1] + recv_counts[size - 1];
}

int all_primes[N]; // assuming total primes ≤ N

MPI_Gatherv(local_primes, local_count, MPI_INT,
            all_primes, recv_counts, displs, MPI_INT,
            0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Prime numbers in the array:\n");
    for (int i = 0; i < total; i++) {
        printf("%d ", all_primes[i]);
    }
    printf("\n");
}

MPI_Finalize();
return 0;
```

```
}
```