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

#define ERR_BADORDER    255
#define TAG_INIT        337
#define TAG_RESULT       42
#define DISP_MAXORDER    12

int getRowCount(int rowsTotal, int mpiRank, int mpiSize);
int matrixMultiply(double *A, double *B, double *C, int n, int n_local);

int main(int argc, char *argv[]) {
    int n = 0, n_ubound, n_local, n_sq, i;
    int mpiRank = 0, mpiSize = 1;
    double *A, *B, *C;
    double t;
    int sizeSent, sizeToBeSent;

    /* Hello, world */
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);

    /* Get n and broadcast it to all processes */
    if (!mpiRank) {
        if (argc > 1) {
            n = atoi(argv[1]);
        }

        if (!n) {
            printf("Order of matrix not supplied, terminating.\n");
        }
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (!n) {
        MPI_Finalize();
        return ERR_BADORDER;
    }

    n_local  = getRowCount(n, mpiRank, mpiSize);

    n_ubound = n * n_local;  /* slave array's upper bound (partial matrix) */
    n_sq     = n * n;    /* master array's upper bound (total matrix) */

    A = (double *) malloc(sizeof(double) * (mpiRank ? n_ubound : n_sq));
    B = (double *) malloc(sizeof(double) *               n_sq );
    C = (double *) malloc(sizeof(double) * (mpiRank ? n_ubound : n_sq));

    /* Initialize A and B using some functions */
    if (!mpiRank) {
        for (i=0; i<n_sq; i++) {
            A[i] = 1.0;
        B[i] = 1.0;
    }
}

/* Start timer */
t = MPI_Wtime();

/* Send A by splitting it in row-wise parts */
if (!mpiRank) {
    sizeSent = n_ubound;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + sizeSent, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT,
                 MPI_COMM_WORLD);
        sizeSent += sizeToBeSent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, n_ubound, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}

/* Send B completely to each process */
MPI_Bcast(B, n*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* Let each process initialize C to zero */
for (i=0; i<n_ubound; i++) {
    C[i] = 0.0;
}

/* Let each process perform its own multiplications */
matrixMultiply(A, B, C, n, n_local);

/* Receive partial results from each slave */
if (!mpiRank) {
    sizeSent = n_ubound;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Recv(C + sizeSent, sizeToBeSent, MPI_DOUBLE, i, TAG_RESULT,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        sizeSent += sizeToBeSent;
    }
}
else { /* Send partial results to master */
    MPI_Send(C, n_ubound, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);
}

/* Stop timer, includes communication time */
t = MPI_Wtime() - t;

/* Print out the final results matrix (root process only) */

/*
if (n > DISP_MAXORDER) {
    n = DISP_MAXORDER;
```

```
        n_sq = n * n;
    }
    */

    if (!mpiRank) {
        for (i=0; i<n_sq; i++) {
            printf("%5.1lf\t", C[i]);
            if (i%n == 0) printf("\n");
        }
    }
    printf("Total time for process #%d was %f seconds.\n", mpiRank, t);

    /* Goodbye, world */
    MPI_Finalize();
    return 0;
}

int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    /* Adjust slack of rows in case rowsTotal is not exactly divisible */
    return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank);
}

int matrixMultiply(double *a, double *b, double *c, int n, int n_local) {
    int i, j, k;
    for (i=0; i<n_local; i++) {
        for (j=0; j<n; j++) {
            for (k=0; k<n; k++) {
                c[i*n + j] += a[i*n + k] * b[k*n + j];
            }
        }
    }
    return 0;
}
```