```
/* ==================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements the DNS (Dekel, Nassimi, Sahni) matrix multiplication
 * algorithm on a cluster using MPI.
 *
 * The DNS algorithm uses a 3D Mesh to partition the intermediate data
 * of the matrix multiplication problem.
 *
 * ================================================================== */

#include "mpi.h"
#include "common.h"
#include "cannon.h" // for struct problem
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <string.h>

#define iDIM 0
#define jDIM 1
#define kDIM 2

struct mesh_info {
        MPI_Comm mesh3d;
        MPI_Comm mesh_ik, ring_j;
        MPI_Comm mesh_jk, ring_i;
        MPI_Comm mesh_ij, ring_k;

        int myrank;
        int coords[3]; // my coordinates in the 3D mesh

        int numprocs; // the number of processors in our cluster

        struct matrix mtx; // the matrix we are multiplying (n x n)

        int q;  // the number of processors in each dimension of the 3D-Mesh
                // each processor will receive two blocks of (n/q)*(n/q) elements

        int n; // the size of the matrix dimension

        int k; // the power
};


void create_topology( struct mesh_info *info , int argc, char *argv[] ) {

        // MPI Initialization
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &info->numprocs);

        for( info->q = 1; (info->q+1)*(info->q+1)*(info->q+1) <= info->numprocs; info->q++ );

        // Create the Topology which we will be using.
        int *dims = malloc( sizeof(int) * 3 ); // 3 dimensions
        int *periods = malloc( sizeof(int) * 3); // wraparound
        dims[iDIM] = dims[jDIM] = dims[kDIM] = info->q;
        periods[iDIM] = periods[jDIM] = periods[kDIM] = 1;

        MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 0, &info->mesh3d);

        MPI_Comm_rank( info->mesh3d, &info->myrank);
        MPI_Cart_coords( info->mesh3d, info->myrank, 3, info->coords);
```

```c
        // the i-k plane
        dims[iDIM] = dims[kDIM] = 1;
        dims[jDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_ik );

        // the j ring
        dims[iDIM] = dims[kDIM] = 0;
        dims[jDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_j);

        // the j-k plane
        dims[jDIM] = dims[kDIM] = 1;
        dims[iDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_jk);

        //the i ring
        dims[jDIM] = dims[kDIM] = 0;
        dims[iDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_i);

        // the i-j plane
        dims[iDIM] = dims[jDIM] = 1;
        dims[kDIM] = 0;

        MPI_Cart_sub( info->mesh3d, dims, &info->mesh_ij);

        // the k rings
        dims[iDIM] = dims[jDIM] = 0;
        dims[kDIM] = 1;

        MPI_Cart_sub( info->mesh3d, dims, &info->ring_k);
}

/*
   Distribute the left or right hand matrix.
*/
void distribute_matrix(struct mesh_info *info, struct matrix *A, struct matrix *Asub, int ringdim, int
bycolumn) {

        int *Ablocks;
        int blksz = info->n / info->q;
        alloc_matrix(Asub, blksz);


        MPI_Comm mesh, ring;
        if (ringdim == jDIM) {
                mesh = info->mesh_ik;
                ring = info->ring_j;
        } else {
                mesh = info->mesh_jk;
                ring = info->ring_i;
        }

        if( info->myrank == 0 ) {
                Ablocks = malloc( sizeof(*A->data) * A->n * A->n );
                matrix_to_blocks(A, Ablocks, blksz, bycolumn);
        }

        if( info->coords[ringdim] == 0 ) {
                MPI_Scatter( Ablocks, blksz*blksz, MPI_INT, Asub->data, blksz*blksz, MPI_INT, 0, mesh );
        }

        if( info->myrank == 0 ) free (Ablocks);
        MPI_Bcast( Asub->data, blksz*blksz, MPI_INT, 0, ring);
}


void gather_C(struct mesh_info *info, struct matrix *Cred, struct matrix *C) {

        int *Cblocks;
        int blksz = info->n / info->q;

        if( info->myrank == 0 ) Cblocks = calloc( sizeof(*C->data), C->n * C->n );

        MPI_Gather( Cred->data, blksz*blksz, MPI_INT, Cblocks, blksz*blksz, MPI_INT, 0, info->mesh_ij);
        if( info->myrank == 0 ) {
                blocks_to_matrix(C, Cblocks, blksz, 0);
                free(Cblocks);
        }
}

/*
   This is the DNS algorithm implementation.
   The multiplied matrix C will only be returned on the root (rank = 0) node.
*/
void dns_multiply(struct mesh_info *info, struct matrix *C, struct matrix *A, struct matrix *B, int
strassen)
{
        struct matrix Asub, Bsub, Csub, Cred;

        // distribute A along the i-k plane and then in the j direction
        distribute_matrix(info, A, &Asub, jDIM, 0);

        // distribute B along the k-j plane and then in the i direction
        distribute_matrix(info, B, &Bsub, iDIM, 1);

        alloc_matrix(&Csub, Asub.n);

        // do the serial matrix multiplication
        bzero(Csub.data, sizeof(*Csub.data)*Csub.n*Csub.n);
        if( strassen ) {
                strassen_matrix2_mult(&Csub, &Asub, &Bsub);
        }
        else {
                naive_matrix_mult_add(&Csub, &Asub, &Bsub);
        }

        // reduce along k dimension to the i-j plane
        if( info->coords[kDIM] == 0 ) alloc_matrix(&Cred, Csub.n);
        MPI_Reduce( Csub.data, Cred.data, Csub.n*Csub.n, MPI_INT, MPI_SUM, 0, info->ring_k);

        // gather on the i-j plane to the root node.
        if( info->coords[kDIM] == 0 ) gather_C(info, &Cred, C);

}


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

        struct input_params m_in_s, *m_in = &m_in_s;
        struct mesh_info info;

        create_topology(&info, argc, argv);

        int result = parse_args(argc, argv, &info.n, &info.k, m_in);
        if (result != 0) {
                MPI_Finalize();
                exit(result);
        }
```

```c
        double start_time = MPI_Wtime();

        // error checking
        if( info.q*info.q*info.q != info.numprocs ) {
                fprintf(stderr, "ERROR: The number of processors must be a perfect cube (not %d).\n",
info.numprocs);
                MPI_Finalize();
                return 1;
        }

        // required for calling the LU decomposition functions.
        struct problem pinfo;
        pinfo.n = info.n;
        pinfo.p = info.numprocs;
        pinfo.k = info.k;
        pinfo.rank = info.myrank;

        alloc_matrix(&pinfo.X, pinfo.n);
        fill_matrix(&pinfo.X, m_in);


        double load_time = MPI_Wtime();

        struct matrix B, C;
        if( info.myrank == 0 ) {
                alloc_matrix(&C, info.n);
                alloc_matrix(&B, info.n);
                memcpy(B.data, pinfo.X.data, sizeof(*pinfo.X.data)*pinfo.X.n*pinfo.X.n);
        }

        int i;
        for( i = 1; i < info.k; i++ ) {
                dns_multiply( &info, &C, &pinfo.X, &B, 0);
                if( info.myrank == 0 ) {
                        memcpy(B.data, C.data, sizeof(*pinfo.X.data)*pinfo.X.n*pinfo.X.n);
                }
        }


        pinfo.Xpow.data = C.data;
        pinfo.Xpow.n = C.n;

        double dns_time = MPI_Wtime();

        number_type determinant = lu1d_determinant(&pinfo);

        double lu_time = MPI_Wtime();

        if( info.myrank == 0 ) {
                if( m_in->print ) {
                        printf("X:\n");
                        print_matrix(&pinfo.X);
                        printf("X^%d:\n", pinfo.k);
                        print_matrix(&pinfo.Xpow);
                }
                printf("determinant: %f\n", determinant);

                double elapsed_time = MPI_Wtime() - start_time;
                printf("data loading time: %f\n", load_time - start_time);
                printf("matrix multiplication time: %f\n", dns_time - load_time);
                printf("LU time: %f\n", lu_time - dns_time);
                printf("total time: %f\n", elapsed_time);
        }

        MPI_Barrier(info.mesh3d);  // avoid processors that end early from killing the rest.

        MPI_Finalize();
        return 0;

}
```