```
/* ======================================================================
 *
 * CS 566 - Assignment 03
 * Camillo Lugaresi, Cosmin Stroe
 *
 * This code implements Cannon's algorithm for matrix multiplication
 * on a cluster using MPI.
 *
 * Cannon's algorithm uses a 2D Mesh to partition the output data
 * of the matrix multiplication problem.
 *
 * ====================================================================== */

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include "common.h"
#include "cannon.h"

// setup our variables
void setup_info(       problem *info, int numprocs)
{
        info->p = numprocs;
        int sqp = 1;
              (sqp*sqp < numprocs) sqp++;
           (sqp*sqp != numprocs) {
                fprintf(stderr, "ERROR: processor count %d is not a square ", numprocs);
                MPI_Finalize();
                exit(1);
        }
        info->sqp = sqp;

        info->blksz = info->n / sqp;
           (info->n % sqp) {
                fprintf(stderr, "ERROR: problem size %d is not multiple of square root of processor "
                        "count %d\n", info->n, numprocs);
                MPI_Finalize();
                exit(1);
        }
        info->blkcells = info->blksz*info->blksz;
}

// setup the mesh topology
void setup_mesh(       problem *info)
{
        /* setup mesh */
        int dims[2] = {info->sqp, info->sqp};
        int periods[2] = {1, 1};        /* wraparound */
        MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &info->mesh);
        MPI_Comm_rank(info->mesh, &(info->rank));

        /* find my coordinates */
        MPI_Cart_coords(info->mesh, info->rank, 2, info->coords);

        /* make row and column communicators */
        int keepdims[2];
        keepdims[HDIM] = 1;
        keepdims[VDIM] = 0;
        MPI_Cart_sub(info->mesh, keepdims, &info->hcomm);
        keepdims[HDIM] = 0;
        keepdims[VDIM] = 1;
        MPI_Cart_sub(info->mesh, keepdims, &info->vcomm);
```

```
}

// allocate the matrix according to the input parameters
void setup_matrix(       problem *info,        input_params *m_in)
{
           (info->rank == 0) {
                alloc_matrix(&info->X, info->n);
                fill_matrix(&info->X, m_in);

                info->Xblocks = malloc(      (*info->Xblocks)*info->n*info->n);
                /* rearrange data into blocks for scatter */
                matrix_to_blocks(&info->X, info->Xblocks, info->blksz, 0);
        }

        alloc_matrix(&info->Xb, info->blksz);
        MPI_Scatter(info->Xblocks, info->blkcells, MPI_INT, info->Xb.data, info->blkcells, MPI_INT, 0,
info->mesh);

        alloc_matrix(&info->A, info->blksz);
        alloc_matrix(&info->B, info->blksz);
        alloc_matrix(&info->C, info->blksz);
        info->temp = malloc(      (*info->temp)*info->blkcells);
}


// the shift operation performed (twice) at every iteration
void ring_shift(int *data, int *temp, size_t count, MPI_Comm ring, int delta)
{
        int src, dst;
        MPI_Status status;

        MPI_Cart_shift(ring, 0, delta, &src, &dst);
        MPI_Sendrecv(data, count, MPI_INT, dst, 0,
                                temp, count, MPI_INT, src, 0,
                                ring, &status);
        memcpy(data, temp,       (int)*count);
}

// the cannon matrix multiplication algorithm.
void cannon(       problem *info)
{
        int i, j;
               matrix *A = &info->A, *B = &info->B, *C = &info->C;

        /* skew */
        ring_shift(A->data, info->temp, info->blkcells, info->hcomm, -info->coords[VDIM]);
        ring_shift(B->data, info->temp, info->blkcells, info->vcomm, -info->coords[HDIM]);

        /* shift and compute */
        bzero(C->data,       (*C->data)*info->blkcells);
            (i = 0; i < info->sqp; i++) {
                naive_matrix_mult_add(C, A, B);
                ring_shift(A->data, info->temp, info->blkcells, info->hcomm, -1);
                ring_shift(B->data, info->temp, info->blkcells, info->vcomm, -1);
        }
}

int main( int argc, char *argv[] )
{
        int numprocs, namelen, i;
        char processor_name[MPI_MAX_PROCESSOR_NAME];

        /* setup */
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Get_processor_name(processor_name,&namelen);
```

```
        double start_time = MPI_Wtime();

               problem info_s, *info = &info_s;
               input_params m_in_s, *m_in = &m_in_s;

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

        setup_info(info, numprocs);
        setup_mesh(info);
        setup_matrix(info, m_in);

        double load_time = MPI_Wtime();

        /* run Cannon's algorithm */
        memcpy(info->C.data, info->Xb.data,       (*info->Xb.data)*info->blkcells);
            (i = 1; i < info->k; i++) {
                memcpy(info->A.data, info->Xb.data,       (*info->Xb.data)*info->blkcells);
                memcpy(info->B.data, info->C.data,       (*info->C.data)*info->blkcells);
                cannon(info);
        }

        double cannon_time = MPI_Wtime();

        /* gather the product */
        MPI_Gather(info->C.data, info->blkcells, MPI_INT, info->Xblocks, info->blkcells, MPI_INT, 0, info-
>mesh);
           (info->rank == 0) {
                alloc_matrix(&info->Xpow, info->n);
                blocks_to_matrix(&info->Xpow, info->Xblocks, info->blksz, 0);
        }

        double gather_time = MPI_Wtime();

        number_type determinant = m_in->lu2d ? lu2d_determinant(info) : lu1d_determinant(info);

        double det_time = MPI_Wtime();

        /* print results */
           (info->rank == 0) {
                   (m_in->print) {
                        printf("X:\n");
                        print_matrix(&info->X);
                        printf("X^%d:\n", info->k);
                        print_matrix(&info->Xpow);
                }

                /* print result */
                extern double convert_time;
                extern double lu_time;
                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", cannon_time - load_time);
                printf("gather time: %f\n", gather_time - cannon_time);
                printf("convert time: %f\n", convert_time - gather_time);
                printf("LU time: %f\n", lu_time - convert_time);
                printf("determinant time: %f\n", det_time - lu_time);
                printf("total time: %f\n", elapsed_time);

                /* profiling */
                printf("\n");

        }
```

```
        MPI_Finalize();
               0;
}
```