```
#include <stdlib.h>
#include <argp.h>
#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "string.h"

//Aaron Holt
//HPSC
//MPI Ping Pong
//compile with mpicc hw3.1-holtat.c -o timeit.o
//run with mpiexec -np 2 ./timeit.o BufferSize

const char *argp_program_version =
    "argp-ex3 1.0";
const char *argp_program_bug_address =
    "<bug-gnu-utils@gnu.org>";

/* Program documentation. */
static char doc[] =
    "Argp example #3 -- a program with options and arguments using argp";

/* A description of the arguments we accept. */
static char args_doc[] = "BufferSize(bytes)";

/* The options we understand. */
static struct argp_option options[] = {
    {"verbose",  ’v’, 0,      0,  "Produce verbose output" },
    { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
    char *args[1];                /* buffer size */
    int verbose;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
    /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;

    switch (key)
        {
        case ’v’:
            arguments->verbose = 1;
            break;

        case ARGP_KEY_ARG:
            if (state->arg_num >= 1)
            /* Too many arguments. */
            argp_usage (state);

            arguments->args[state->arg_num] = arg;

            break;

        case ARGP_KEY_END:
            if (state->arg_num < 1)
            /* Not enough arguments. */
            argp_usage (state);
            break;
```

```
        default:
            return ARGP_ERR_UNKNOWN;
        }
    return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };

int
main (int argc, char **argv)
{
    struct arguments arguments;

    /* Parse our arguments; every option seen by parse_opt will
       be reflected in arguments. */
    argp_parse (&argp, argc, argv, 0, 0, &arguments);

    // printf ("Buffer Size (bytes) = %s\n"
    //         "VERBOSE = %s\n",
    //         arguments.args[0],
    //         arguments.verbose ? "yes" : "no");

    //buffer size from input char* to int
    int size;
    size = 1; //default
    if (sscanf (arguments.args[0], "%i", &size)!=1) {}


    //For now, hardcode tag (operation)
    int tag = 0; //tag = 0 => addition

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);


    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    //Bail if incorrect
    if (world_size > 2)
    {
        if (world_rank == 0)
        {
            printf("World size greater than 2, exiting\n");
        }
        exit(0);
    }
    if (world_size < 2)
    {
        if (world_rank == 0)
        {
            printf("World size less than 2, exiting\n");
        }
        exit(0);
    }
```

```c
    if (world_rank == 0 && arguments.verbose == 1)
    {
        printf("Buffer size (bytes) = %d\n", size);
    }

    //Timing variables
    double total_time;
    total_time = 0;
    double starttime, endtime;
    double alltime[50] = {0};

    //Dynamically allocate arrays
    char *buffer;     //buffer to send
    buffer = (char*) malloc(size*sizeof(char)+1);
    buffer[size] = ’\0’;

    int j = 0;
    if (world_rank == 0)
    {
        for (j=0; j<size; j++)
        {
            buffer[j] = (char)11.0;
        }
        if ( arguments.verbose == 1)
        {
            for (j=0; j<3; j++)
            {
                printf("Initial data in buffer[%d]: %d ", j, buffer[j]);
                printf("\n");
            }
        }
    }

    //Timing
    //10 warmup, 40 test
    int kk;
    for (kk=0; kk<60; kk++)
    {
        for (j=0; j<size; j++)
        {
            buffer[j] = (char)11.0;
        }

        MPI_Barrier(MPI_COMM_WORLD);

        if (kk>=10)
        {
            starttime = MPI_Wtime();
        }

        //Time bcast
        if (world_size > 1)
        {
            if (world_rank == 0)
            {
                MPI_Send(buffer, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
                MPI_Recv(buffer, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
            else if (world_rank == 1)
            {
                MPI_Recv(buffer, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
                MPI_Send(buffer, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
            }
        }
        if (kk>=10)
        {
            endtime = MPI_Wtime();
            total_time = total_time + endtime - starttime;
            alltime[kk-10] = endtime - starttime;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    // printf("Data %d, world rank %d\n", buffer[0], world_rank);

    if (world_rank == 0)
    {
        int i;
        for(i=0; i<50; i++)
        {
            if (i < 49)
            {
                printf("%2.9f,", alltime[i]);
            }
            else
            {
                printf("%2.9f", alltime[i]);
            }
        }
        printf("\n%2.9f\n", total_time/100);
    }
    MPI_Barrier(MPI_COMM_WORLD);


    //Free malloc’ed data
    free (buffer);

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

```
#include <stdlib.h>
#include <argp.h>
#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "string.h"

//Aaron Holt
//HPSC
//MPI Dense Matrix Transpose
//compile with mpicc hw3.2-holtat.c -o dense_transpose.o
//run with mpiexec -np 2 ./dense_transpose.o SquareMatrixSize

const char *argp_program_version =
    "argp-ex3 1.0";
const char *argp_program_bug_address =
    "<bug-gnu-utils@gnu.org>";

/* Program documentation. */
static char doc[] =
    "Argp example #3 -- a program with options and arguments using argp";

/* A description of the arguments we accept. */
static char args_doc[] = "MatrixSize";

/* The options we understand. */
static struct argp_option options[] = {
    {"verbose",  ’v’, 0,      0,  "Produce verbose output" },
    { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
    char *args[1];                /* m x m */
    int verbose;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
    /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;

    switch (key)
        {
        case ’v’:
            arguments->verbose = 1;
            break;

        case ARGP_KEY_ARG:
            if (state->arg_num >= 1)
            /* Too many arguments. */
            argp_usage (state);
            arguments->args[state->arg_num] = arg;
            break;

        case ARGP_KEY_END:
            if (state->arg_num < 1)
            /* Not enough arguments. */
            argp_usage (state);
            break;

        default:
            return ARGP_ERR_UNKNOWN;
        }
    return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };


void matrix_transpose(int m, double matrix[m][m], int from, int to, int world_rank)
{
    int i,j;

    //Create new column derived datatype
    MPI_Datatype column;
    //count, blocklength, stride, oldtype, *newtype
    MPI_Type_hvector(m, 1, m*sizeof(double), MPI_DOUBLE, &column);
    MPI_Type_commit(&column);

    //Send columns, 1 at a time
    if (world_rank == from)
    {
        for(i=0; i<m; i++)
        {
            //*data,count,type,to,tag,comm
            MPI_Send(&matrix[0][i], 1, column, to, 0, MPI_COMM_WORLD);
        }
    }
    //Receive as rows, 1 at a time
    else if (world_rank == to)
    {

        for(i=0; i<m; i++)
        {
            //*data,count,type,from,tag,comm,mpi_status
            MPI_Recv(&matrix[i][0], m, MPI_DOUBLE, from, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
    }

    return;
}



int main (int argc, char **argv)
{
    struct arguments arguments;

    /* Parse our arguments; every option seen by parse_opt will
       be reflected in arguments. */
    argp_parse (&argp, argc, argv, 0, 0, &arguments);

    //matrix size, mxm
    int m;
    m = 5;
    if (sscanf (arguments.args[0], "%i", &m)!=1) {}

    //verbose?
    int verbose;
    verbose = arguments.verbose;

    // printf("m x n = %d x %d\n", m, m);

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);


    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    int i, j, from, to;
    double matrix[m][m];

    //initialize matrices
    if (world_rank == 0)
    {
        printf("Initial matrix, world_rank = %d\n", world_rank);
        for(i=0;i<m;i++)
        {
            for(j=0;j<m;j++)
            {
                matrix[i][j] = j;
                printf("%d ", j);
            }
            printf("\n");
        }
        printf("\n");
    }
    else
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<m;j++)
            {
                matrix[i][j] = -1;
            }
        }
    }


    //Call matrix transpose function
    from = 0;
    to = 1;
    matrix_transpose(m, matrix, from, to, world_rank);

    //Print final matrix
    if (world_rank == to)
    {
       printf("Final matrix, world_rank = %d\n", world_rank);
        for (i=0; i<m; i++)
        {
            for(j=0; j<m; j++)
            {
                printf("%d ", (int)matrix[i][j]);
            }
            printf("\n");
        }
    }


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