MONASH University

Lab Session Week 9

Student ID	Student Name	Student Email
32844700	Teh Jia Xuan	

Task 1 – Code

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#define SHIFT_ROW 0
#define SHIFT_COL 1
#define DISP 1
int main(int argc, char *argv[]) {
    int ndims=2, size, my_rank, reorder, my_cart_rank, ierr;
    int nrows, ncols;
    int nbr_i_lo, nbr_i_hi;
    int nbr_j_lo, nbr_j_hi;
    MPI_Comm comm2D;
    int dims[ndims],coord[ndims];
    int wrap_around[ndims];
    /* start up initial MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    /* process command line arguments*/
    if (argc == 3) {
        nrows = atoi (argv[1]);
        ncols = atoi (argv[2]);
        dims[0] = nrows; /* number of rows */
        dims[1] = ncols; /* number of columns */
        if( (nrows*ncols) != size) {
            if( my_rank == 0) printf("ERROR: nrows*ncols)=%d *%d = %d != %d\n",
nrows, ncols, nrows*ncols,size);
            MPI_Finalize();
            return 0;
        }

    }

    else {
        nrows=ncols=(int)sqrt(size);
        dims[0]=dims[1]=0;
    }
```

```
    /*************************************************************
    */
    /* create cartesian topology for processes */
    /*************************************************************
    */
    MPI_Dims_create(size, ndims, dims); //use to divide the number of processor
across multiple dimensions
    if(my_rank == 0){
        printf("Root Rank: %d. Comm Size: %d: Grid Dimension =[%d x %d]
\n",my_rank,size,dims[0],dims[1]);
    }

    /* create cartesian mapping */
    wrap_around[0] = wrap_around[1] = 0; /* periodic shift is.false. no wrap around
*/
    reorder = 1; //allow to reorder the cartesian rank
    ierr =0;
    ierr= MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, wrap_around, reorder,
&comm2D); //arrange processors into a grid for communication
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);
    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm2D, my_rank, ndims, coord);
    /* use my cartesian coordinates to find my rank in cartesian
    group*/
    MPI_Cart_rank(comm2D, coord, &my_cart_rank);
    /* get my neighbors; axis is coordinate dimension of shift */
    /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
    P[my_row+1] */
    /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
    P[my_col+1] */

    MPI_Cart_shift(comm2D, SHIFT_ROW, DISP, &nbr_i_lo, &nbr_i_hi);
    MPI_Cart_shift(comm2D, SHIFT_COL, DISP, &nbr_j_lo, &nbr_j_hi);


    printf("Global rank: %d. Cart rank: %d. Coord: (%d, %d).Left: %d. Right: %d.
Top: %d. Bottom: %d\n", my_rank,my_cart_rank, coord[0], coord[1], nbr_j_lo,
nbr_j_hi, nbr_i_lo, nbr_i_hi);

    fflush(stdout);
    MPI_Comm_free( &comm2D );
    MPI_Finalize();
    return 0;
}
```

MONASH University

Task 2 – Code (basic)

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include <stdbool.h>
#include <time.h>

#define SHIFT_ROW 0
#define SHIFT_COL 1
#define DISP 1
int randomNum(int lower, int upper);
bool isprime(int number);
int main(int argc, char *argv[]) {
    int ndims=2, size, my_rank, reorder, my_cart_rank, ierr;
    int nrows, ncols;
    int nbr_i_lo, nbr_i_hi;
    int nbr_j_lo, nbr_j_hi;
    MPI_Comm comm2D;
    int dims[ndims],coord[ndims];
    int wrap_around[ndims];
    int prime = 0;
    int rec_i_lo = 0, rec_i_hi = 0, rec_j_lo = 0, rec_j_hi = 0;
    char filename[20];
    FILE *log_file;
    struct timespec start, end, startComp, endComp;
    double time_taken;

    /* start up initial MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    clock_gettime(CLOCK_MONOTONIC, &start);

    //seed the random number generator with a unique value
    srand(time(NULL) + my_rank);

    /* process command line arguments*/
    if (argc == 3) {
        nrows = atoi (argv[1]);
        ncols = atoi (argv[2]);
        dims[0] = nrows; /* number of rows */
        dims[1] = ncols; /* number of columns */
        if( (nrows*ncols) != size) {
```

```
            if( my_rank == 0) printf("ERROR: nrows*ncols)=%d *%d = %d != %d\n",
nrows, ncols, nrows*ncols,size);
            MPI_Finalize();
            return 0;
        }

    }

    else {
        nrows=ncols=(int)sqrt(size);
        dims[0]=dims[1]=0;
    }
    /*************************************************************
    */
    /* create cartesian topology for processes */
    /*************************************************************
    */
    MPI_Dims_create(size, ndims, dims);
    if(my_rank == 0){
        printf("Root Rank: %d. Comm Size: %d: Grid Dimension =[%d x %d]
\n",my_rank,size,dims[0],dims[1]);
    }

    /* create cartesian mapping */
    wrap_around[0] = wrap_around[1] = 0; /* periodic shift is
    .false. */
    reorder = 1;
    ierr =0;
    ierr= MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, wrap_around, reorder,
&comm2D);
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);
    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm2D, my_rank, ndims, coord);
    /* use my cartesian coordinates to find my rank in cartesian
    group*/
    MPI_Cart_rank(comm2D, coord, &my_cart_rank);
    /* get my neighbors; axis is coordinate dimension of shift */
    /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
    P[my_row+1] */
    /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
    P[my_col+1] */

    MPI_Cart_shift(comm2D, SHIFT_ROW, DISP, &nbr_i_lo, &nbr_i_hi);
    MPI_Cart_shift(comm2D, SHIFT_COL, DISP, &nbr_j_lo, &nbr_j_hi);

    //iterate 500
    for(int i = 0; i < 500; i++){
        do {
            prime = randomNum(1, 1000); //get prime number
        } while (!isprime(prime));
```

```
        //send prime to its neighbour
        MPI_Send(&prime, 1, MPI_INT, nbr_i_lo, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_i_hi, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_j_lo, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_j_hi, 0, MPI_COMM_WORLD);

        //receive from its neighbour and store to rec
        MPI_Recv(&rec_i_lo, 1, MPI_INT, nbr_i_lo, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_i_hi, 1, MPI_INT, nbr_i_hi, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_j_lo, 1, MPI_INT, nbr_j_lo, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_j_hi, 1, MPI_INT, nbr_j_hi, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        sprintf(filename, "task2_rank_%d.txt", my_rank);

        log_file = fopen(filename, "a");
        if (log_file == NULL){
            printf("Error opening log file\n");
            MPI_Finalize();
            return 1;
        }
        //compare prime with neighbour and write down if is same
        if (prime == rec_i_lo) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_i_lo);
        }
        if (prime == rec_i_hi) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_i_hi);
        }
        if (prime == rec_j_lo) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_j_lo);
        }
        if (prime == rec_j_hi) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_j_hi);
        }

        fclose(log_file);

        printf("Prime: %d\n", prime);
        printf("rank: %d, Received primes: %d, %d, %d, %d\n", my_cart_rank,
rec_i_lo, rec_i_hi, rec_j_lo, rec_j_hi);

    }
    //wait for all processes
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Global rank: %d. Cart rank: %d. Coord: (%d, %d).Left: %d. Right: %d.
Top: %d. Bottom: %d\n", my_rank,my_cart_rank, coord[0], coord[1], nbr_j_lo,
nbr_j_hi, nbr_i_lo, nbr_i_hi);
    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time each processors: %f sec \n", time_taken);
    fflush(stdout);

    MPI_Comm_free( &comm2D );
    MPI_Finalize();
    return 0;
}

bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}

//generate random number
int randomNum(int lower, int upper){
    return (rand() % (upper - lower + 1)) + lower;
}
```

Task 2 – Code (Extended Task)

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include <stdbool.h>
#include <time.h>

#define SHIFT_ROW 0
#define SHIFT_COL 1
#define DISP 1
#define NUM_PRIMES 500

int randomPrime(int lower, int upper);
bool isprime(int number);

int main(int argc, char *argv[]) {
    int ndims=2, size, my_rank, reorder, my_cart_rank, ierr;
    int nrows, ncols;
    int nbr_i_lo, nbr_i_hi;
    int nbr_j_lo, nbr_j_hi;
    MPI_Comm comm2D;
    int dims[ndims],coord[ndims];
    int wrap_around[ndims];
    int prime = 0;
    int rec_i_lo = 0, rec_i_hi = 0, rec_j_lo = 0, rec_j_hi = 0;
    char filename[20];
    char *buffer, *buffer_i_lo, *buffer_i_hi, *buffer_j_lo, *buffer_j_hi;
    int buf_size, buf_size_int, position = 0;
    int primes[NUM_PRIMES];
    int received_primes[NUM_PRIMES], received_primes_i_hi[NUM_PRIMES],
received_primes_i_lo[NUM_PRIMES],
    received_primes_j_lo[NUM_PRIMES], received_primes_j_hi[NUM_PRIMES];
    FILE *log_file;
    struct timespec start, end, startComp, endComp;
    double time_taken;
    MPI_Request send_requests[4], recv_requests[4];

    /* start up initial MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    clock_gettime(CLOCK_MONOTONIC, &start);

    //determine size of buffer needed to pack
    MPI_Pack_size(NUM_PRIMES, MPI_INT, MPI_COMM_WORLD, &buf_size_int);
    buf_size = buf_size_int + MPI_BSEND_OVERHEAD;

    //create buffer for all the neighbours
```

```
    buffer = (char *) malloc((unsigned) buf_size);
    buffer_i_lo = (char *) malloc((unsigned) buf_size);
    buffer_i_hi = (char *) malloc((unsigned) buf_size);
    buffer_j_lo = (char *) malloc((unsigned) buf_size);
    buffer_j_hi = (char *) malloc((unsigned) buf_size);


    //seed the random number generator with a unique value
    srand(time(NULL) + my_rank);

    /* process command line arguments*/
    if (argc == 3) {
        nrows = atoi (argv[1]);
        ncols = atoi (argv[2]);
        dims[0] = nrows; /* number of rows */
        dims[1] = ncols; /* number of columns */
        if( (nrows*ncols) != size) {
            if( my_rank == 0) printf("ERROR: nrows*ncols)=%d *%d = %d != %d\n",
nrows, ncols, nrows*ncols,size);
            MPI_Finalize();
            return 0;
        }

    }

    else {
        nrows=ncols=(int)sqrt(size);
        dims[0]=dims[1]=0;
    }
    /*************************************************************
    */
    /* create cartesian topology for processes */
    /*************************************************************
    */
    MPI_Dims_create(size, ndims, dims);
    if(my_rank == 0){
        printf("Root Rank: %d. Comm Size: %d: Grid Dimension =[%d x %d]
\n",my_rank,size,dims[0],dims[1]);
    }

    /* create cartesian mapping */
    wrap_around[0] = wrap_around[1] = 0; /* periodic shift is
    .false. */
    reorder = 1;
    ierr =0;
    ierr= MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, wrap_around, reorder,
&comm2D);
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);
    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm2D, my_rank, ndims, coord);
```

```
    /* use my cartesian coordinates to find my rank in cartesian
    group*/
    MPI_Cart_rank(comm2D, coord, &my_cart_rank);
    /* get my neighbors; axis is coordinate dimension of shift */
    /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
    P[my_row+1] */
    /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
    P[my_col+1] */

    MPI_Cart_shift(comm2D, SHIFT_ROW, DISP, &nbr_i_lo, &nbr_i_hi);
    MPI_Cart_shift(comm2D, SHIFT_COL, DISP, &nbr_j_lo, &nbr_j_hi);

    for(int i = 0; i < 500; i++){
        primes[i] = randomPrime(1,1000);
    }

    //pack primes in a buffer
    MPI_Pack(primes, NUM_PRIMES, MPI_INT, buffer, buf_size, &position,
MPI_COMM_WORLD);


    //send the buffers to neighbour using non blocking
    MPI_Isend(buffer, position, MPI_PACKED, nbr_i_lo, 0, MPI_COMM_WORLD,
&send_requests[0]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_i_hi, 0, MPI_COMM_WORLD,
&send_requests[1]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_j_lo, 0, MPI_COMM_WORLD,
&send_requests[2]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_j_hi, 0, MPI_COMM_WORLD,
&send_requests[3]);

    MPI_Irecv(buffer_i_lo, buf_size, MPI_PACKED, nbr_i_lo, 0, MPI_COMM_WORLD,
&recv_requests[0]);
    MPI_Irecv(buffer_i_hi, buf_size, MPI_PACKED, nbr_i_hi, 0, MPI_COMM_WORLD,
&recv_requests[1]);
    MPI_Irecv(buffer_j_lo, buf_size, MPI_PACKED, nbr_j_lo, 0, MPI_COMM_WORLD,
&recv_requests[2]);
    MPI_Irecv(buffer_j_hi, buf_size, MPI_PACKED, nbr_j_hi, 0, MPI_COMM_WORLD,
&recv_requests[3]);

    //wait for all buffer received
    MPI_Waitall(4, send_requests, MPI_STATUSES_IGNORE);
    MPI_Waitall(4, recv_requests, MPI_STATUSES_IGNORE);

    //unpack and store to an array
    position = 0;
    MPI_Unpack(buffer_i_lo, buf_size, &position, received_primes_i_lo, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
```

```
    MPI_Unpack(buffer_i_hi, buf_size, &position, received_primes_i_hi, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_j_lo, buf_size, &position, received_primes_j_lo, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_j_hi, buf_size, &position, received_primes_j_hi, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    sprintf(filename, "task2_ex_rank_%d.txt", my_rank);

    //open a file
    log_file = fopen(filename, "a");
    if (log_file == NULL){
        printf("Error opening log file\n");
        MPI_Finalize();
        return 1;
    }
    //compare all the prime numbers if same then write
    for (int i = 0; i < NUM_PRIMES; i++){

        if (primes[i] == received_primes_i_lo[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_i_lo);
        }
        if (primes[i] == received_primes_i_hi[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_i_hi);
        }
        if (primes[i] == received_primes_j_lo[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_j_lo);
        }
        if (primes[i] == received_primes_j_hi[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_j_hi);
        }

        printf("Prime: %d\n", primes[i]);
        printf("rank: %d, Received primes: %d, %d, %d, %d\n", my_cart_rank,
received_primes_i_lo[i], received_primes_i_hi[i], received_primes_j_lo[i],
received_primes_j_hi[i]);
        fflush(stdout);

    }

        fclose(log_file);
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Global rank: %d. Cart rank: %d. Coord: (%d, %d).Left: %d. Right: %d.
Top: %d. Bottom: %d\n", my_rank,my_cart_rank, coord[0], coord[1], nbr_j_lo,
nbr_j_hi, nbr_i_lo, nbr_i_hi);

    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time each processors: %f sec \n", time_taken);

    MPI_Comm_free( &comm2D );
    MPI_Finalize();
    return 0;
}

bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}

//generate random prime number
int randomPrime(int lower, int upper){
    int prime = 0;
    do {
        prime = (rand() % (upper - lower + 1)) + lower;
        } while (!isprime(prime));
    return prime;
```

```
}
```

Task 2 – Observations, results and explanation

Basic code

```
Overall time each processors: 1.867147 sec
Overall time each processors: 1.867180 sec
Overall time each processors: 1.867275 sec
Overall time each processors: 1.867285 sec
Overall time each processors: 1.867214 sec
Overall time each processors: 1.867234 sec
Overall time each processors: 1.867222 sec

real    0m2.211s
user    0m5.036s
sys     0m0.921s
```

Arrangement

```
Global rank: 3. Cart rank: 3. Coord: (1, 1).Left: 2. Right: -2. Top: 1. Bottom: 5
Global rank: 5. Cart rank: 5. Coord: (2, 1).Left: 4. Right: -2. Top: 3. Bottom: 7
Global rank: 2. Cart rank: 2. Coord: (1, 0).Left: -2. Right: 3. Top: 0. Bottom: 4
Global rank: 7. Cart rank: 7. Coord: (3, 1).Left: 6. Right: -2. Top: 5. Bottom: -2
Global rank: 0. Cart rank: 0. Coord: (0, 0).Left: -2. Right: 1. Top: -2. Bottom: 2
Global rank: 1. Cart rank: 1. Coord: (0, 1).Left: 0. Right: -2. Top: -2. Bottom: 3
Global rank: 6. Cart rank: 6. Coord: (3, 0).Left: -2. Right: 7. Top: 4. Bottom: -2
Global rank: 4. Cart rank: 4. Coord: (2, 0).Left: -2. Right: 5. Top: 2. Bottom: 6
```

Extended code

```
Overall time each processors: 0.019382 sec
Overall time each processors: 0.019499 sec
Overall time each processors: 0.019414 sec
Overall time each processors: 0.019546 sec
Overall time each processors: 0.019361 sec
Overall time each processors: 0.019499 sec
Overall time each processors: 0.019494 sec
Overall time each processors: 0.019536 sec

real    0m0.369s
user    0m0.164s
sys     0m0.267s
```

Arrangement

```
Global rank: 6. Cart rank: 6. Coord: (3, 0).Left: -2. Right: 7. Top: 4. Bottom: -2
Global rank: 5. Cart rank: 5. Coord: (2, 1).Left: 4. Right: -2. Top: 3. Bottom: 7
Global rank: 0. Cart rank: 0. Coord: (0, 0).Left: -2. Right: 1. Top: -2. Bottom: 2
Global rank: 2. Cart rank: 2. Coord: (1, 0).Left: -2. Right: 3. Top: 0. Bottom: 4
Global rank: 3. Cart rank: 3. Coord: (1, 1).Left: 2. Right: -2. Top: 1. Bottom: 5
Global rank: 1. Cart rank: 1. Coord: (0, 1).Left: 0. Right: -2. Top: -2. Bottom: 3
Global rank: 4. Cart rank: 4. Coord: (2, 0).Left: -2. Right: 5. Top: 2. Bottom: 6
Global rank: 7. Cart rank: 7. Coord: (3, 1).Left: 6. Right: -2. Top: 5. Bottom: -2
```

The program speed up from 2.211s to 0.369s. The speed up is 2.211/0.369 = 5.992 times
Here are the reasons why my extended code is faster:

1. We can observe that extended code is faster than basic code as I have optimised the code using MPI_Isend and MPI_Irecv. By using non blocking communication it allows processes to continue computation while messages are being sent or received. Hence, reducing idle times.

2. Reduced communication overhead, in my code, Instead of sending a single prime number and waiting for a response in every iteration. I have generated all 500 prime numbers and store it in an array. After that, I sent the array across its adjacent processors at once. So I just need to send once instead of sending 500 times. This minimises the communication overhead caused by repeatedly sending and receiving messages. This makes the code more efficient.

3. Improved message size efficiency, sending a single large message is more efficient than sending 500 smaller messages due to it has fixed communication cost attached with each message.

Task 4 – Code (basic)

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include <stdbool.h>



#define SHIFT_X 0
#define SHIFT_Y 1
#define SHIFT_Z 2
#define DISP 1
int randomPrime(int lower, int upper);
bool isprime(int number);
int main(int argc, char *argv[]) {
    int ndims=3, size, my_rank, reorder, my_cart_rank, ierr;
    int nrows, ncols, nz;
    int nbr_x_lo, nbr_x_hi;
```

```
    int nbr_y_lo, nbr_y_hi;
    int nbr_z_lo, nbr_z_hi;
    MPI_Comm comm3D;
    int dims[ndims],coord[ndims];
    int wrap_around[ndims];
    char filename[20];
    FILE *log_file;
    int prime = 0,rec_x_lo = 0, rec_x_hi = 0, rec_y_lo = 0, rec_y_hi = 0, rec_z_lo
= 0, rec_z_hi = 0;
    struct timespec start, end, startComp, endComp;
    double time_taken;


    /* start up initial MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    clock_gettime(CLOCK_MONOTONIC, &start);
    srand(time(NULL) + my_rank);

    //input error checking
    if (argc == 4) {
        nrows = atoi (argv[1]);
        ncols = atoi (argv[2]);
        nz = atoi(argv[3]);
        dims[0] = nrows; /* number of rows */
        dims[1] = ncols; /* number of columns */
        dims[2] = nz;
        if( (nrows*ncols*nz) != size) {
            if( my_rank == 0) printf("ERROR: nrows*ncols*nz)=%d *%d *%d
= %d != %d\n", nrows, ncols, nz, nrows*ncols*nz,size);
            MPI_Finalize();
            return 0;
        }

    }

    else {
        nrows=ncols=(int)cbrt(size);
        dims[0]=dims[1]=dims[2]=0;
    }

    /*************************************************************
    */
    /* create cartesian topology for processes */
    /*************************************************************
    */
    MPI_Dims_create(size, ndims, dims);
```

```
    if(my_rank == 0){
        printf("Root Rank: %d. Comm Size: %d: Grid Dimension =[%d x %d]
\n",my_rank,size,dims[0],dims[1]);
    }

    /* create cartesian mapping */
    wrap_around[0] = wrap_around[1] = wrap_around[2] = 0; /* periodic shift is
    .false. */
    reorder = 1;
    ierr =0;
    ierr= MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, wrap_around, reorder,
&comm3D);
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);
    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm3D, my_rank, ndims, coord);
    /* use my cartesian coordinates to find my rank in cartesian
    group*/
    MPI_Cart_rank(comm3D, coord, &my_cart_rank);
    /* get my neighbors; axis is coordinate dimension of shift */
    /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
    P[my_row+1] */
    /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
    P[my_col+1] */

    //shift the cart to let the processor know the neighbours
    MPI_Cart_shift(comm3D, SHIFT_X, DISP, &nbr_x_lo, &nbr_x_hi);
    MPI_Cart_shift(comm3D, SHIFT_Y, DISP, &nbr_y_lo, &nbr_y_hi);
    MPI_Cart_shift(comm3D, SHIFT_Z, DISP, &nbr_z_lo, &nbr_z_hi);

    //iterate 500 times
    for(int i = 0; i < 500; i++){

        prime = randomPrime(1, 1000); //get a random prime number

        //mpi send to its neighbours
        MPI_Send(&prime, 1, MPI_INT, nbr_x_lo, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_x_hi, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_y_lo, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_y_hi, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_z_lo, 0, MPI_COMM_WORLD);
        MPI_Send(&prime, 1, MPI_INT, nbr_z_hi, 0, MPI_COMM_WORLD);

        //receive from its neighbour
        MPI_Recv(&rec_x_lo, 1, MPI_INT, nbr_x_lo, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_x_hi, 1, MPI_INT, nbr_x_hi, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_y_lo, 1, MPI_INT, nbr_y_lo, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
        MPI_Recv(&rec_y_hi, 1, MPI_INT, nbr_y_hi, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_z_hi, 1, MPI_INT, nbr_z_lo, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        MPI_Recv(&rec_z_hi, 1, MPI_INT, nbr_z_hi, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        //determine the file name
        sprintf(filename, "task4_rank_%d.txt", my_rank);
        //open the filename
        log_file = fopen(filename, "a");
        if (log_file == NULL){
            printf("Error opening log file\n");
            MPI_Finalize();
            return 1;
        }
        //if it matches the prime then write in file
        if (prime == rec_x_lo) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_x_lo);
        }
        if (prime == rec_x_hi) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_x_hi);
        }
        if (prime == rec_y_lo) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_y_lo);
        }
        if (prime == rec_y_hi) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_y_hi);
        }
        if (prime == rec_z_lo) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_z_lo);
        }
        if (prime == rec_z_hi) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, prime, nbr_z_hi);
        }

        fclose(log_file);

        printf("Prime: %d\n", prime);

        printf("rank: %d, Received primes: %d, %d, %d, %d, %d, %d\n",
        my_cart_rank, rec_x_lo, rec_x_hi, rec_y_lo, rec_y_hi, rec_z_lo, rec_z_hi);
    }
    //wait for all processes
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Global rank: %d, Cartesian rank: %d, Coordinates: (%d, %d, %d)\n",
           my_rank, my_cart_rank, coord[0], coord[1], coord[2]);

    printf("Neighbors: Left: %d, Right: %d, Top: %d, Bottom: %d, Front: %d,
Rear: %d\n",
           nbr_x_lo, nbr_x_hi, nbr_y_lo, nbr_y_hi, nbr_z_lo, nbr_z_hi);

    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time each processors: %f sec \n", time_taken);

    fflush(stdout);
    MPI_Comm_free( &comm3D );
    MPI_Finalize();
    return 0;
}

bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}

//generate random prime number
int randomPrime(int lower, int upper){
    int prime = 0;
    do {
```

```
        prime = (rand() % (upper - lower + 1)) + lower;
        } while (!isprime(prime));
    return prime;
}
```

Time

```
Overall time each processors: 2.555284 sec
Overall time each processors: 2.555398 sec
Overall time each processors: 2.555344 sec
Overall time each processors: 2.555233 sec
Overall time each processors: 2.555396 sec
Overall time each processors: 2.555432 sec
Overall time each processors: 2.555221 sec
Overall time each processors: 2.555179 sec
Overall time each processors: 2.555241 sec
Overall time each processors: 2.555238 sec
Overall time each processors: 2.555233 sec
Overall time each processors: 2.555324 sec

real    0m2.981s
user    0m2.173s
sys     0m10.720s
```

Arrangement

```
Global rank: 9, Cartesian rank: 9, Coordinates: (2, 0, 1)
Neighbors: Left: 5, Right: -2, Top: -2, Bottom: 11, Front: 8, Rear: -2
Global rank: 3, Cartesian rank: 3, Coordinates: (0, 1, 1)
Neighbors: Left: -2, Right: 7, Top: 1, Bottom: -2, Front: 2, Rear: -2
Global rank: 6, Cartesian rank: 6, Coordinates: (1, 1, 0)
Neighbors: Left: 2, Right: 10, Top: 4, Bottom: -2, Front: -2, Rear: 7
Global rank: 2, Cartesian rank: 2, Coordinates: (0, 1, 0)
Neighbors: Left: -2, Right: 6, Top: 0, Bottom: -2, Front: -2, Rear: 3
Global rank: 1, Cartesian rank: 1, Coordinates: (0, 0, 1)
Neighbors: Left: -2, Right: 5, Top: -2, Bottom: 3, Front: 0, Rear: -2
Global rank: 11, Cartesian rank: 11, Coordinates: (2, 1, 1)
Neighbors: Left: 7, Right: -2, Top: 9, Bottom: -2, Front: 10, Rear: -2
Global rank: 5, Cartesian rank: 5, Coordinates: (1, 0, 1)
Neighbors: Left: 1, Right: 9, Top: -2, Bottom: 7, Front: 4, Rear: -2
Global rank: 8, Cartesian rank: 8, Coordinates: (2, 0, 0)
Neighbors: Left: 4, Right: -2, Top: -2, Bottom: 10, Front: -2, Rear: 9
Global rank: 10, Cartesian rank: 10, Coordinates: (2, 1, 0)
Neighbors: Left: 6, Right: -2, Top: 8, Bottom: -2, Front: -2, Rear: 11
Global rank: 0, Cartesian rank: 0, Coordinates: (0, 0, 0)
Neighbors: Left: -2, Right: 4, Top: -2, Bottom: 2, Front: -2, Rear: 1
Global rank: 7, Cartesian rank: 7, Coordinates: (1, 1, 1)
Neighbors: Left: 3, Right: 11, Top: 5, Bottom: -2, Front: 6, Rear: -2
Global rank: 4, Cartesian rank: 4, Coordinates: (1, 0, 0)
Neighbors: Left: 0, Right: 8, Top: -2, Bottom: 6, Front: -2, Rear: 5
```

MONASH University

Task 4 – Code (Extended Task)

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>
#include <stdbool.h>
#include <time.h>

#define SHIFT_X 0
#define SHIFT_Y 1
#define SHIFT_Z 2
#define DISP 1
#define NUM_PRIMES 500

int randomPrime(int lower, int upper);
bool isprime(int number);
int main(int argc, char *argv[]) {
    int ndims=3, size, my_rank, reorder, my_cart_rank, ierr;
    int nrows, ncols, nz;
    int nbr_x_lo, nbr_x_hi;
    int nbr_y_lo, nbr_y_hi;
    int nbr_z_lo, nbr_z_hi;
    MPI_Comm comm3D;
    int dims[ndims],coord[ndims];
    int wrap_around[ndims];
    int prime = 0,rec_x_lo = 0, rec_x_hi = 0, rec_y_lo = 0, rec_y_hi = 0, rec_z_lo
= 0, rec_z_hi = 0;
    char filename[20];
    char *buffer, *buffer_x_lo, *buffer_x_hi, *buffer_y_lo, *buffer_y_hi,
*buffer_z_lo, *buffer_z_hi;
    int buf_size, buf_size_int, position = 0;
    int primes[NUM_PRIMES];
    int received_primes[NUM_PRIMES], received_primes_x_hi[NUM_PRIMES],
received_primes_x_lo[NUM_PRIMES],
    received_primes_y_lo[NUM_PRIMES], received_primes_y_hi[NUM_PRIMES],
received_primes_z_lo[NUM_PRIMES],
    received_primes_z_hi[NUM_PRIMES];

    FILE *log_file;
    struct timespec start, end, startComp, endComp;
    double time_taken;
    MPI_Request send_requests[6], recv_requests[6];

    /* start up initial MPI environment */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    clock_gettime(CLOCK_MONOTONIC, &start);
```

```
    //determine size of buffer needed to pack
    MPI_Pack_size(NUM_PRIMES, MPI_INT, MPI_COMM_WORLD, &buf_size_int);
    buf_size = buf_size_int + MPI_BSEND_OVERHEAD;
    //create buffer for all neighbours
    buffer = (char *) malloc((unsigned) buf_size);
    buffer_x_lo = (char *) malloc((unsigned) buf_size);
    buffer_x_hi = (char *) malloc((unsigned) buf_size);
    buffer_y_lo = (char *) malloc((unsigned) buf_size);
    buffer_y_hi = (char *) malloc((unsigned) buf_size);
    buffer_z_lo = (char *) malloc((unsigned) buf_size);
    buffer_z_hi = (char *) malloc((unsigned) buf_size);

    //seed the random number generator with a unique value
    srand(time(NULL) + my_rank);

    //input error checking
     if (argc == 4) {
        nrows = atoi (argv[1]);
        ncols = atoi (argv[2]);
        nz = atoi(argv[3]);
        dims[0] = nrows; /* number of rows */
        dims[1] = ncols; /* number of columns */
        dims[2] = nz;
        if( (nrows*ncols*nz) != size) {
            if( my_rank == 0) printf("ERROR: nrows*ncols*nz)=%d *%d *%d
= %d != %d\n", nrows, ncols, nz, nrows*ncols*nz,size);
            MPI_Finalize();
            return 0;
        }

    }

    else {
        nrows=ncols=(int)cbrt(size);
        dims[0]=dims[1]=dims[2]=0;
    }


    /*************************************************************
    */
    /* create cartesian topology for processes */
    /*************************************************************
    */
    MPI_Dims_create(size, ndims, dims);
    if(my_rank == 0){
        printf("Root Rank: %d. Comm Size: %d: Grid Dimension =[%d x %d]
\n",my_rank,size,dims[0],dims[1]);
    }
```

```
    /* create cartesian mapping */
    wrap_around[0] = wrap_around[1] = wrap_around[2] =0; /* periodic shift is
    .false. */
    reorder = 1;
    ierr =0;
    ierr= MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, wrap_around, reorder,
&comm3D);
    if(ierr != 0) printf("ERROR[%d] creating CART\n",ierr);
    /* find my coordinates in the cartesian communicator group */
    MPI_Cart_coords(comm3D, my_rank, ndims, coord);
    /* use my cartesian coordinates to find my rank in cartesian
    group*/
    MPI_Cart_rank(comm3D, coord, &my_cart_rank);
    /* get my neighbors; axis is coordinate dimension of shift */
    /* axis=0 ==> shift along the rows: P[my_row-1]: P[me] :
    P[my_row+1] */
    /* axis=1 ==> shift along the columns P[my_col-1]: P[me] :
    P[my_col+1] */

    //shift the neighbours
    MPI_Cart_shift(comm3D, SHIFT_X, DISP, &nbr_x_lo, &nbr_x_hi);
    MPI_Cart_shift(comm3D, SHIFT_Y, DISP, &nbr_y_lo, &nbr_y_hi);
    MPI_Cart_shift(comm3D, SHIFT_Z, DISP, &nbr_z_lo, &nbr_z_hi);

    for(int i = 0; i < 500; i++){
        primes[i] = randomPrime(1,1000);
    }
    //pack the prime into the buffer
    MPI_Pack(primes, NUM_PRIMES, MPI_INT, buffer, buf_size, &position,
MPI_COMM_WORLD);

    //send the prime buffer to neighbours
    MPI_Isend(buffer, position, MPI_PACKED, nbr_x_lo, 0, MPI_COMM_WORLD,
&send_requests[0]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_x_hi, 0, MPI_COMM_WORLD,
&send_requests[1]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_y_lo, 0, MPI_COMM_WORLD,
&send_requests[2]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_y_hi, 0, MPI_COMM_WORLD,
&send_requests[3]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_z_lo, 0, MPI_COMM_WORLD,
&send_requests[4]);
    MPI_Isend(buffer, position, MPI_PACKED, nbr_z_hi, 0, MPI_COMM_WORLD,
&send_requests[5]);

    //receive from neighbours with the buffer
    MPI_Irecv(buffer_x_lo, buf_size, MPI_PACKED, nbr_x_lo, 0, MPI_COMM_WORLD,
&recv_requests[0]);
    MPI_Irecv(buffer_x_hi, buf_size, MPI_PACKED, nbr_x_hi, 0, MPI_COMM_WORLD,
&recv_requests[1]);
```

```
    MPI_Irecv(buffer_y_lo, buf_size, MPI_PACKED, nbr_y_lo, 0, MPI_COMM_WORLD,
&recv_requests[2]);
    MPI_Irecv(buffer_y_hi, buf_size, MPI_PACKED, nbr_y_hi, 0, MPI_COMM_WORLD,
&recv_requests[3]);
    MPI_Irecv(buffer_y_lo, buf_size, MPI_PACKED, nbr_z_lo, 0, MPI_COMM_WORLD,
&recv_requests[4]);
    MPI_Irecv(buffer_y_hi, buf_size, MPI_PACKED, nbr_z_hi, 0, MPI_COMM_WORLD,
&recv_requests[5]);

    MPI_Waitall(6, send_requests, MPI_STATUSES_IGNORE);
    MPI_Waitall(6, recv_requests, MPI_STATUSES_IGNORE);

    //unpack the buffer and store in respective buffer and reset the  position
    position = 0;
    MPI_Unpack(buffer_x_lo, buf_size, &position, received_primes_x_lo, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_x_hi, buf_size, &position, received_primes_x_hi, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_y_lo, buf_size, &position, received_primes_y_lo, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_y_hi, buf_size, &position, received_primes_y_hi, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_z_lo, buf_size, &position, received_primes_z_lo, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    position = 0;
    MPI_Unpack(buffer_z_hi, buf_size, &position, received_primes_z_hi, NUM_PRIMES,
MPI_INT, MPI_COMM_WORLD);

    sprintf(filename, "task4_ex_rank_%d.txt", my_rank);

    log_file = fopen(filename, "a");
    if (log_file == NULL){
        printf("Error opening log file\n");
        MPI_Finalize();
        return 1;
    }
    //check the prime number
    for (int i = 0; i < NUM_PRIMES; i++){

        if (primes[i] == received_primes_x_lo[i]) {
```

```
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_x_lo);
        }
        if (primes[i] == received_primes_x_hi[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_x_hi);
        }
        if (primes[i] == received_primes_y_lo[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_y_lo);
        }
        if (primes[i] == received_primes_y_hi[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_y_hi);
        }
        if (primes[i] == received_primes_z_lo[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_z_lo);
        }
        if (primes[i] == received_primes_z_hi[i]) {
            fprintf(log_file, "Rank %d: Prime %d is equal to adjacent prime from
rank %d.\n", my_rank, primes[i], nbr_z_hi);
        }

        printf("Prime: %d\n", primes[i]);
        printf("rank: %d, Received primes: %d, %d, %d, %d, %d, %d\n", my_cart_rank,
received_primes_x_lo[i], received_primes_x_hi[i],
                received_primes_y_lo[i], received_primes_y_hi[i],
received_primes_y_lo[i], received_primes_y_hi[i]);


    }

        fclose(log_file);


    MPI_Barrier(MPI_COMM_WORLD);
    printf("Global rank: %d. Cart rank: %d. Coord: (%d, %d).Left: %d. Right: %d.
Top: %d. Bottom: %d. Front:%d. Rear:%d\n", my_rank,my_cart_rank,
            coord[0], coord[1], nbr_x_lo, nbr_x_hi, nbr_y_lo, nbr_y_hi, nbr_z_lo,
nbr_z_hi);

    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time each processors: %f sec \n", time_taken);

    MPI_Comm_free( &comm3D );
    MPI_Finalize();
    return 0;
```

```
}

bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2 || number == 3){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    if (number % 3 == 0){
        return false;
    }

    //loop until sqrt of n as p*q = n so loop until one of them is sufficient
    for (int i = 5; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;
}

//generate prime number
int randomPrime(int lower, int upper){
    int prime = 0;
    do {
        prime = (rand() % (upper - lower + 1)) + lower;
        } while (!isprime(prime));
    return prime;
}
```

```
Overall time each processors: 0.021854 sec
Overall time each processors: 0.021728 sec
Overall time each processors: 0.021797 sec
Overall time each processors: 0.021572 sec
Overall time each processors: 0.021800 sec
Overall time each processors: 0.021830 sec
Overall time each processors: 0.021769 sec
Overall time each processors: 0.021906 sec
Overall time each processors: 0.021797 sec
Overall time each processors: 0.021775 sec
Overall time each processors: 0.021906 sec
Overall time each processors: 0.021611 sec

real    0m0.447s
user    0m0.281s
sys     0m0.542s
```

Arrangement

```
Global rank: 6. Cart rank: 6. Coord: (1, 1).Left: 2. Right: 10. Top: 4. Bottom: -2. Front:-2. Rear:7
Global rank: 0. Cart rank: 0. Coord: (0, 0).Left: -2. Right: 4. Top: -2. Bottom: 2. Front:-2. Rear:1
Global rank: 10. Cart rank: 10. Coord: (2, 1).Left: 6. Right: -2. Top: 8. Bottom: -2. Front:-2. Rear:11
Global rank: 8. Cart rank: 8. Coord: (2, 0).Left: 4. Right: -2. Top: -2. Bottom: 10. Front:-2. Rear:9
Global rank: 11. Cart rank: 11. Coord: (2, 1).Left: 7. Right: -2. Top: 9. Bottom: -2. Front:10. Rear:-2
Global rank: 9. Cart rank: 9. Coord: (2, 0).Left: 5. Right: -2. Top: -2. Bottom: 11. Front:8. Rear:-2
Global rank: 5. Cart rank: 5. Coord: (1, 0).Left: 1. Right: 9. Top: -2. Bottom: 7. Front:4. Rear:-2
Global rank: 2. Cart rank: 2. Coord: (0, 1).Left: -2. Right: 6. Top: 0. Bottom: -2. Front:-2. Rear:3
Global rank: 4. Cart rank: 4. Coord: (1, 0).Left: 0. Right: 8. Top: -2. Bottom: 6. Front:-2. Rear:5
Global rank: 1. Cart rank: 1. Coord: (0, 0).Left: -2. Right: 5. Top: -2. Bottom: 3. Front:0. Rear:-2
Global rank: 3. Cart rank: 3. Coord: (0, 1).Left: -2. Right: 7. Top: 1. Bottom: -2. Front:2. Rear:-2
Global rank: 7. Cart rank: 7. Coord: (1, 1).Left: 3. Right: 11. Top: 5. Bottom: -2. Front:6. Rear:-2
```

Task 4 – Observations, results and explanation

We can observe that the refine version is running faster than the basic one. It improves from 2.981s to 0.447s. The speed up is 2.981/0.447 = 6.668 times. The reasons that it is running faster are:

1. Packing multiple primes in a buffer, by packing 500 prime numbers into a single buffer and sending that buffer in one communication, it reduces the total number of messages exchanged. Each communication in MPI involves some latency and overhead, so reducing the number of sends and receives leads to significant performance improvements.

2. Non blocking communication with MPI_Isend and MPI_Irecv, non blocking communication allows processes to overlap computation and communication. Instead of waiting for each communication to complete before proceeding, the program continues its computation while messages are being sent and received. Hence, it spent less idle time waiting for communication to finish.