3.6 Suppose *comm_sz=4* and suppose that x is a vector with $n = 14$ components.

(a) How would the components of x be distributed among the processes in a program that used a block distribution.

(b) How would the components of x be distributed in a process that used a cyclic distribution?

(c) How would the components of x be distributed among the processes in a program that used a block-cyclic distribution with block size $b = 2$.

(a) Block				
P_0	0	1	2	3
P_1	4	5	6	7
P_2	8	9	10	11
P_3	12	13		

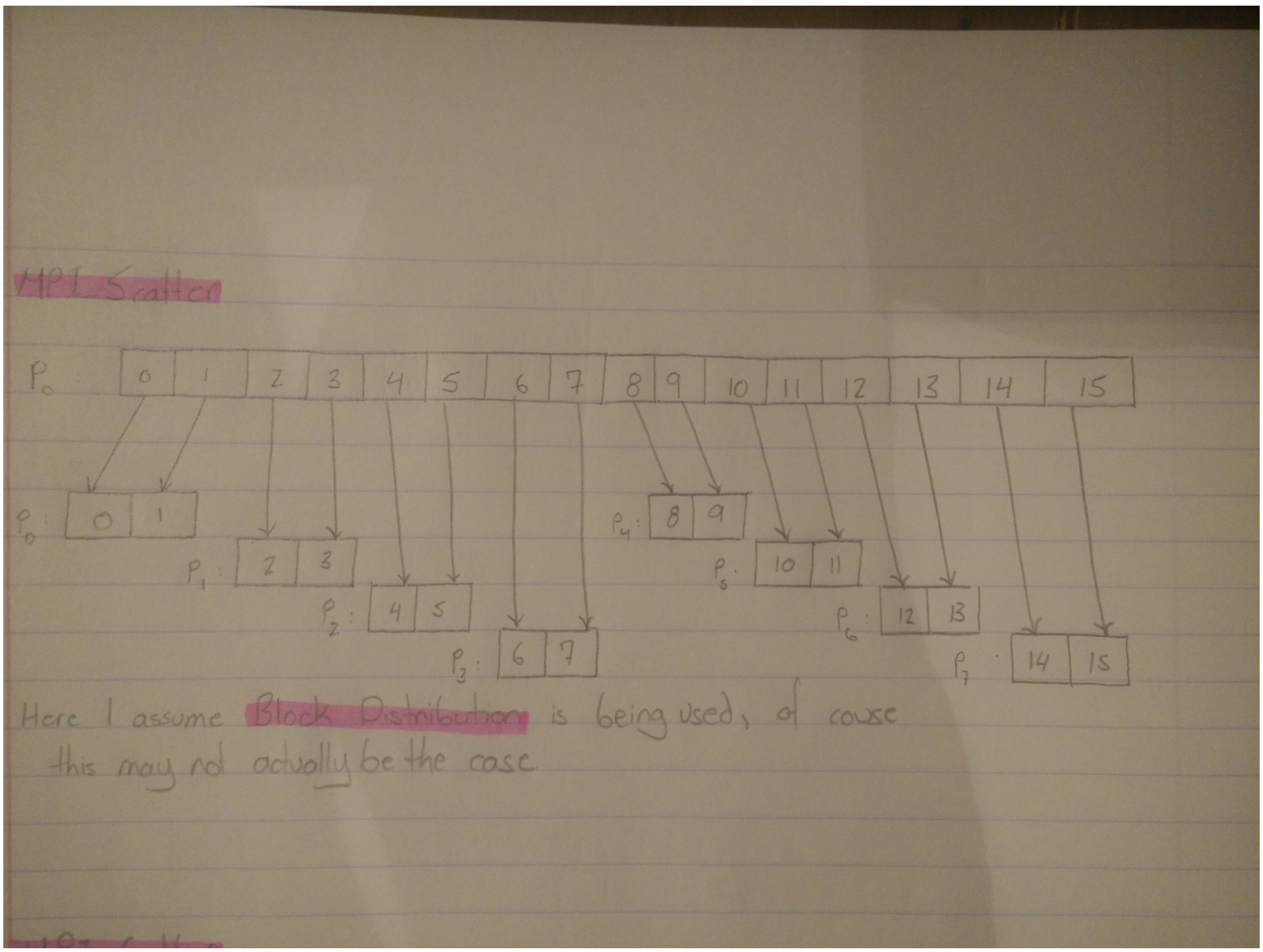
(b) Cyclic				
P_0	0	4	8	12
P_1	1	5	9	13
P_2	2	6	10	
P_3	3	7	11	

(c) Block Cyclic: b=2				
P_0	0	1	8	9
P_1	2	3	10	11
P_2	4	5	12	13
P_3	6	7		

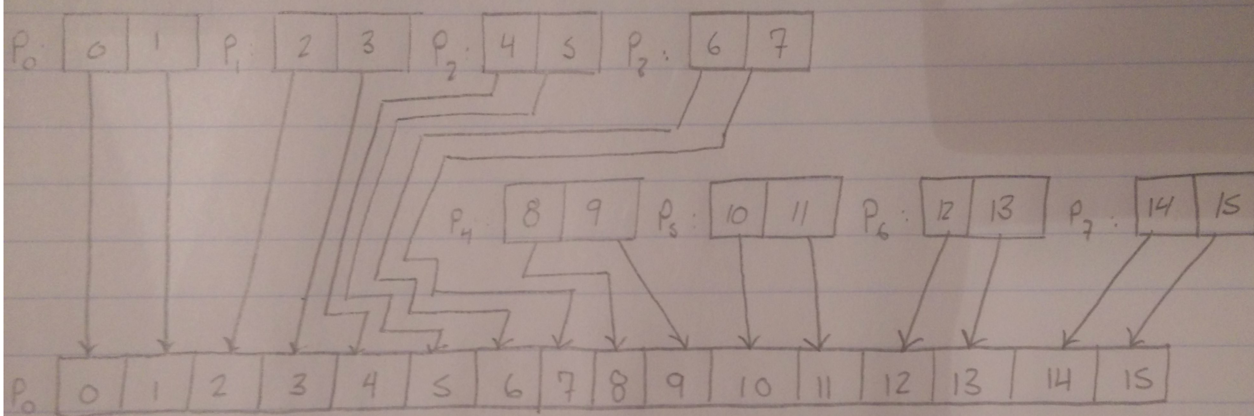
3.8 Suppose *comm_sz=8* and $n = 16$.

(a) Draw a diagram that shows how MPI_Scatter can be implemented using tree-structured communication on with comm_sz processes when process 0 needs to distribute an array containing n elements.

(b) Draw a diagram that shows how MPI_Gather can be implemented using tree-structured communication when an n-element array that has been distributed among comm_sz processes needs to be gathered into process 0.



MPI_Gather



Again, this assumes the Block Distribution used in part (a) above. Note that it is virtually identical, with sent arrows inverted.

3.9 Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by comm_sz.

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

#define ROOT 0

// prints a vector inline to the standard output
void print_vec(float * vec, size_t count);

// only returns actual result to 'root' process, all others receive 0
float vec_dot_prod(float * a, float * b, int vcount, int root) {
        int comm_sz;
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

        float * reca, * recb; // received vectors by all processes

        /* --- SYNC DATA BETWEEN PROCESSES. --- */
        MPI_Bcast(&vcount, 1, MPI_INT, root, MPI_COMM_WORLD);
        int count = vcount / comm_sz;
        reca = malloc(sizeof(float) * count);
        recb = malloc(sizeof(float) * count);

        MPI_Scatter(a, count, MPI_FLOAT, reca, count, MPI_FLOAT,
                        root, MPI_COMM_WORLD);
        MPI_Scatter(b, count, MPI_FLOAT, recb, count, MPI_FLOAT,
                        root, MPI_COMM_WORLD);

        /* --- COMPUTE PARTIAL SUM OF DOT PRODUCT. --- */
```

```
        float dot = 0;
        for (int i=0; i<count; i++) {
                dot += reca[i] * recb[i];
        }

        /* --- COMPUTE FINAL DOT PRODUCT BY REDUCING DOT. --- */
        float dot_sum = 0;
        MPI_Reduce(&dot, &dot_sum, 1, MPI_FLOAT, MPI_SUM,
                        root, MPI_COMM_WORLD);
        free(reca);
        free(recb);
        return dot_sum;
}

// only returns data to the 'root' process, all other processes receive NULL
float * scalar_mul(float s, float * vec, int vcount, int root) {
        int comm_sz, my_rank;
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* --- SYNC DATA BETWEEN PROCESSES. --- */
        MPI_Bcast(&vcount, 1, MPI_INT, root, MPI_COMM_WORLD);
        int count = vcount / comm_sz;
        float * rec = malloc(sizeof(float) * count);
        MPI_Scatter(vec, count, MPI_FLOAT, rec, count, MPI_FLOAT,
                        root, MPI_COMM_WORLD);
        MPI_Bcast(&s, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

        /* --- COMPUTE THE MULTIPLICATION, THEN GATHER THE RESULTS --- */
        for (int i=0; i<count; i++) {
                rec[i] *= s;
        }

        float * ret = (my_rank == root) ? malloc(sizeof(float) * vcount) : NULL;
        MPI_Gather(rec, count, MPI_FLOAT, ret, count, MPI_FLOAT,
                        root, MPI_COMM_WORLD);
```

```
        free(rec);
        return ret;
}

int main(int argc, char ** argv) {
        int comm_sz, my_rank;

        // These will be read by vcount
        float s;
        float * veca, * vecb; // vectors read by process 0
        int vcount;

        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* --- READ THE VECTORS AND SCALAR VALUE. --- */
        if (my_rank == ROOT) {
                printf("Enter the size of each vector (must be a multiple of %d): ",
                                comm_sz);
                fflush(stdout);
                scanf("%d", &vcount);

                veca = malloc(sizeof(float) * vcount);
                vecb = malloc(sizeof(float) * vcount);

                for (int i=0; i<vcount; i++) {
                        printf("A[%d]: ", i);
                        fflush(stdout);
                        scanf("%f", &veca[i]);
                }

                printf("\n");
                for (int i=0; i<vcount; i++) {
                        printf("B[%d]: ", i);
```

```
                        fflush(stdout);
                        scanf("%f", &vecb[i]);
                }

                printf("\nEnter a scalar value: ");
                fflush(stdout);
                scanf("%f", &s);
        }

        /* --- PERFORM COMPUTATIONS. --- */
        float dot = vec_dot_prod(veca, vecb, vcount, ROOT);
        float * s_times_veca = scalar_mul(s, veca, vcount, ROOT);
        float * s_times_vecb = scalar_mul(s, vecb, vcount, ROOT);

        /* --- OUTPUT THE RESULTS. --- */
        if (my_rank == ROOT) {
                // First, dot product results
                printf("\n------------------------------------------"
                                "---------------------\n");
                print_vec(veca, vcount);
                printf(" dot ");
                print_vec(vecb, vcount);
                printf(" = %f\n", dot);

                printf("\n------------------------------------------"
                                "---------------------\n");
                printf("%f x ", s);
                print_vec(veca, vcount);
                printf(" = ");
                print_vec(s_times_veca, vcount);
                printf("\n");

                printf("\n------------------------------------------"
                                "---------------------\n");
                printf("%f x ", s);
                print_vec(vecb, vcount);
```

```
                printf(" = ");
                print_vec(s_times_vecb, vcount);
                printf("\n------------------------------------------"
                                "---------------------\n");

                fflush(stdout);
        }

        if (my_rank == ROOT) {
                free(s_times_veca);
                free(s_times_vecb);
        }

        MPI_Finalize();
        return 0;
}

void print_vec(float * vec, size_t count) {
        printf("[");
        for (int i=0; i<count; i++) {
                printf("%.1f", vec[i]);
                if (i != count-1) { printf(", "); }
        }
        printf("]");
}
```