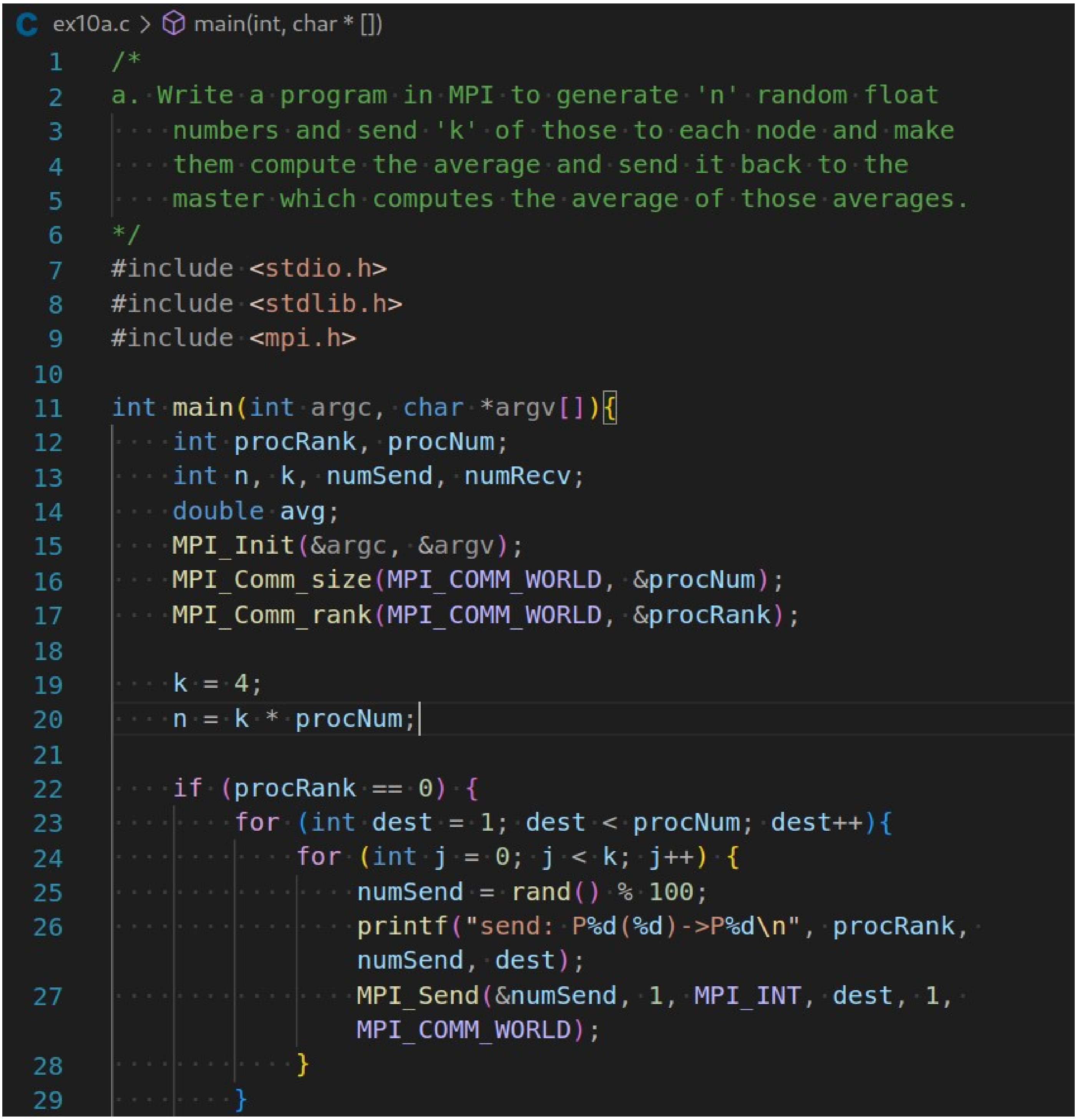
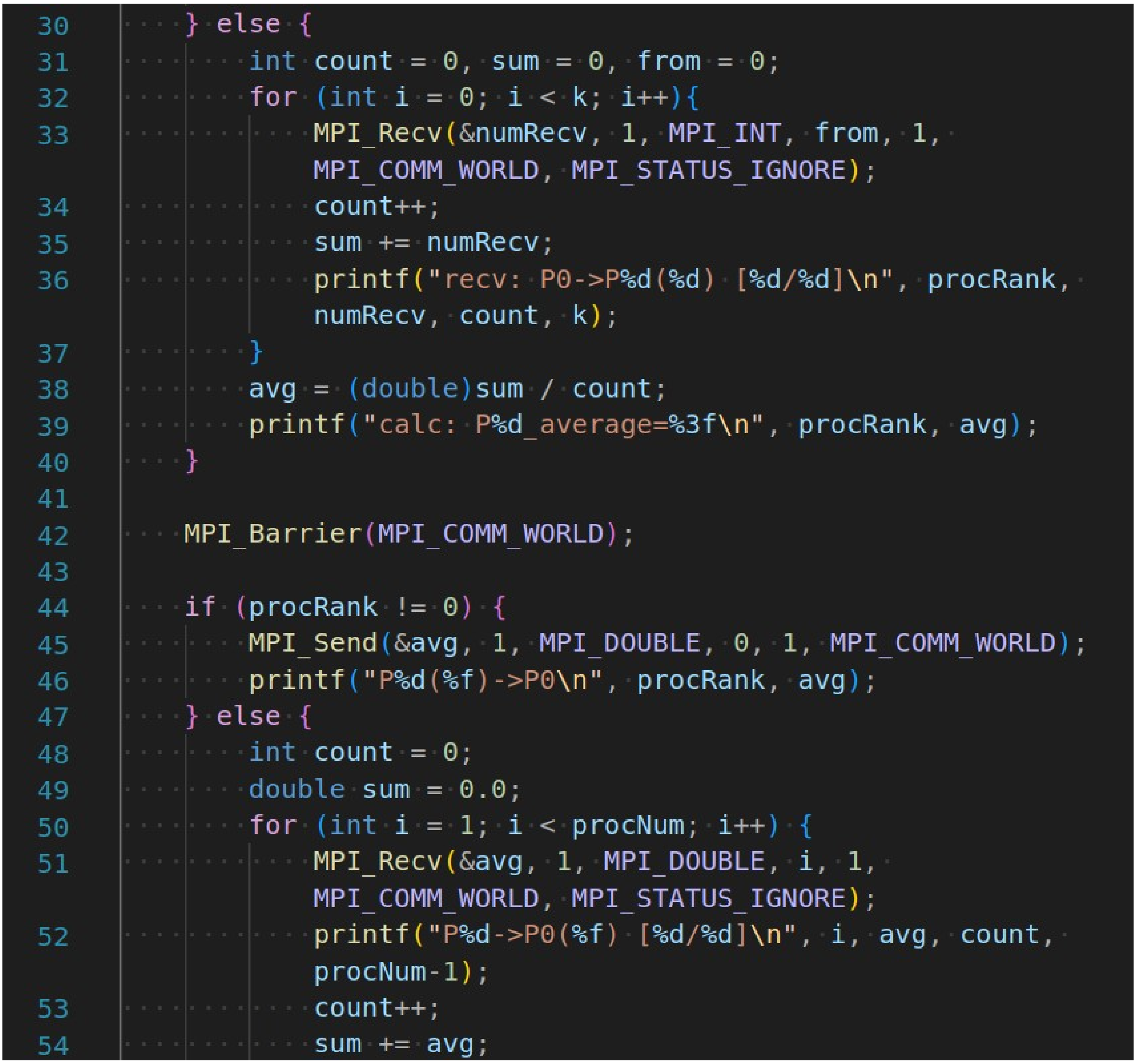
20BCE1550
Samridh Anand Paatni
CSE4001 Lab 10
MPI Collective Communication

a:

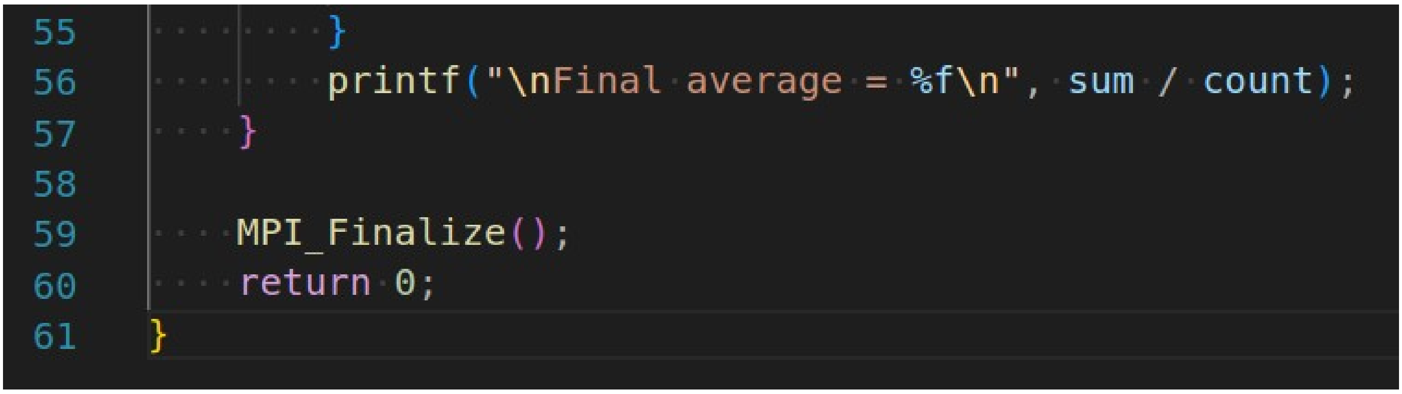
Code:



```c
/*
a. Write a program in MPI to generate 'n' random float
    numbers and send 'k' of those to each node and make
    them compute the average and send it back to the
    master which computes the average of those averages.
*/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[]){
    int procRank, procNum;
    int n, k, numSend, numRecv;
    double avg;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    k = 4;
    n = k * procNum;

    if (procRank == 0) {
        for (int dest = 1; dest < procNum; dest++){
            for (int j = 0; j < k; j++) {
                numSend = rand() % 100;
                printf("send: P%d(%d)->P%d\n", procRank, numSend, dest);
                MPI_Send(&numSend, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            }
        }
```

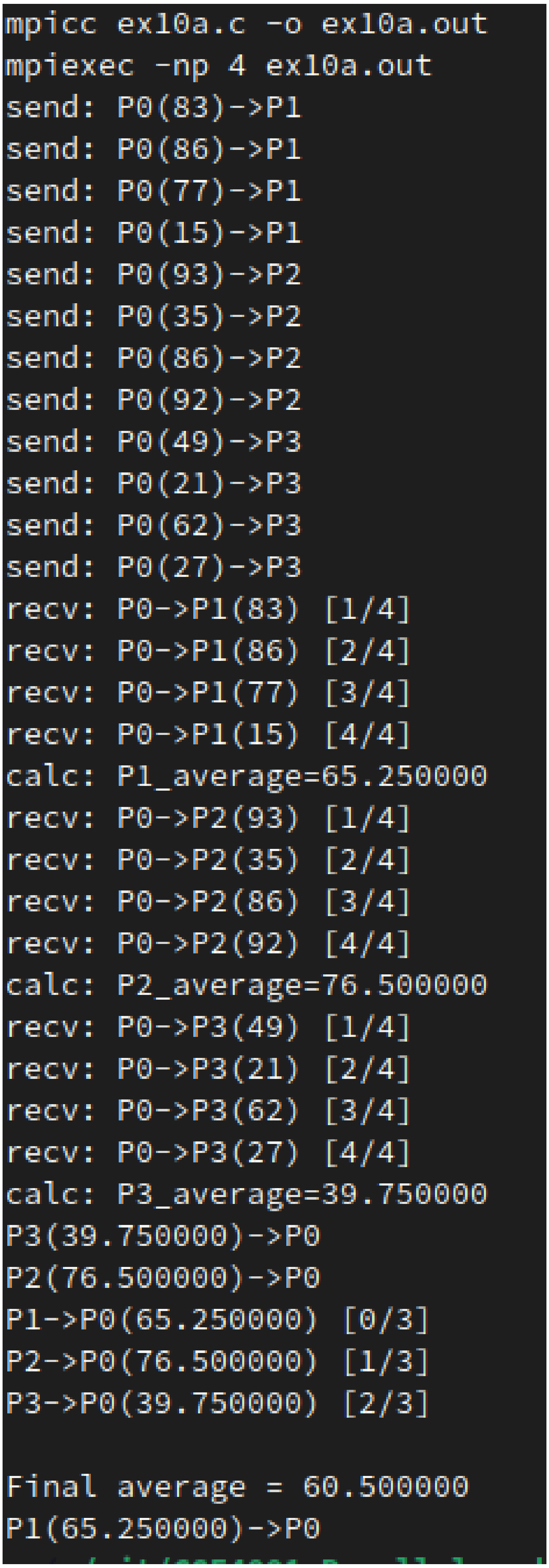


```c
    } else {
        int count = 0, sum = 0, from = 0;
        for (int i = 0; i < k; i++){
            MPI_Recv(&numRecv, 1, MPI_INT, from, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            count++;
            sum += numRecv;
            printf("recv: P0->P%d(%d) [%d/%d]\n", procRank, numRecv, count, k);
        }
        avg = (double)sum / count;
        printf("calc: P%d_average=%3f\n", procRank, avg);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (procRank != 0) {
        MPI_Send(&avg, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
        printf("P%d(%f)->P0\n", procRank, avg);
    } else {
        int count = 0;
        double sum = 0.0;
        for (int i = 1; i < procNum; i++) {
            MPI_Recv(&avg, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("P%d->P0(%f) [%d/%d]\n", i, avg, count, procNum-1);
            count++;
            sum += avg;
```



```c
        }
        printf("\nFinal average = %f\n", sum / count);
    }

    MPI_Finalize();
    return 0;
}
```

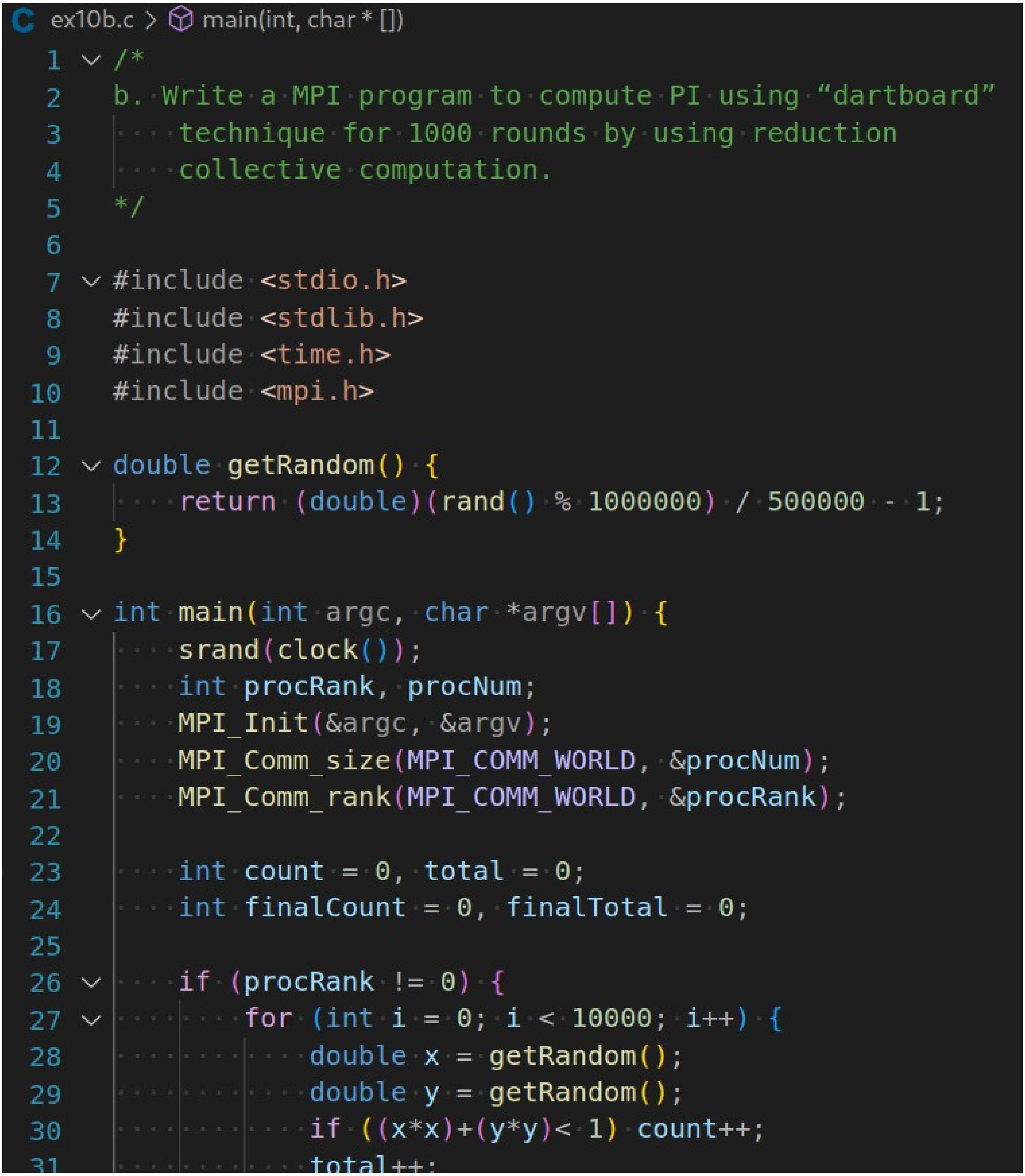
Output:



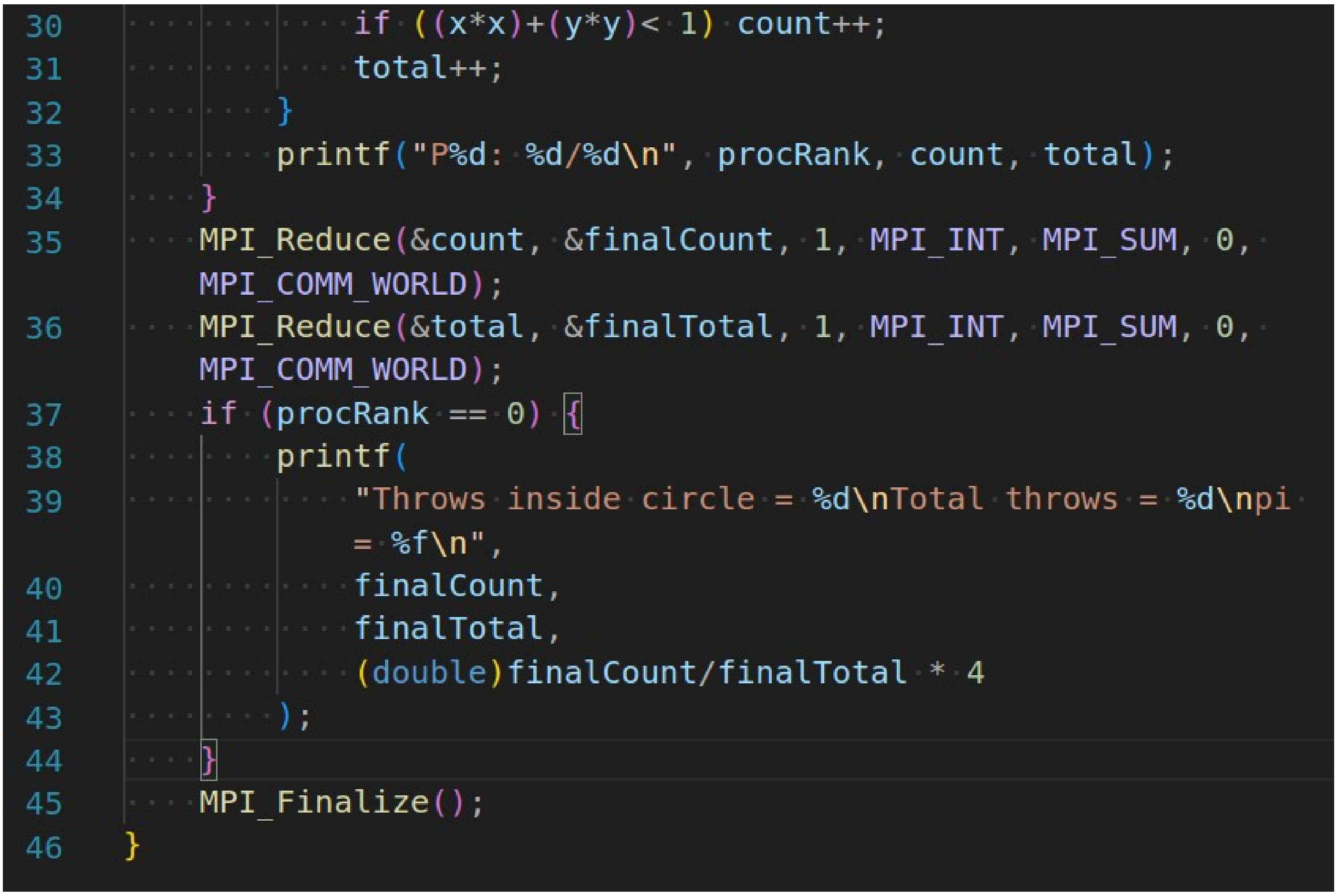
```
mpicc ex10a.c -o ex10a.out
mpiexec -np 4 ex10a.out
send: P0(83)->P1
send: P0(86)->P1
send: P0(77)->P1
send: P0(15)->P1
send: P0(93)->P2
send: P0(35)->P2
send: P0(86)->P2
send: P0(92)->P2
send: P0(49)->P3
send: P0(21)->P3
send: P0(62)->P3
send: P0(27)->P3
recv: P0->P1(83) [1/4]
recv: P0->P1(86) [2/4]
recv: P0->P1(77) [3/4]
recv: P0->P1(15) [4/4]
calc: P1_average=65.250000
recv: P0->P2(93) [1/4]
recv: P0->P2(35) [2/4]
recv: P0->P2(86) [3/4]
recv: P0->P2(92) [4/4]
calc: P2_average=76.500000
recv: P0->P3(49) [1/4]
recv: P0->P3(21) [2/4]
recv: P0->P3(62) [3/4]
recv: P0->P3(27) [4/4]
calc: P3_average=39.750000
P3(39.750000)->P0
P2(76.500000)->P0
P1->P0(65.250000) [0/3]
P2->P0(76.500000) [1/3]
P3->P0(39.750000) [2/3]

Final average = 60.500000
P1(65.250000)->P0
```

b:

Code:

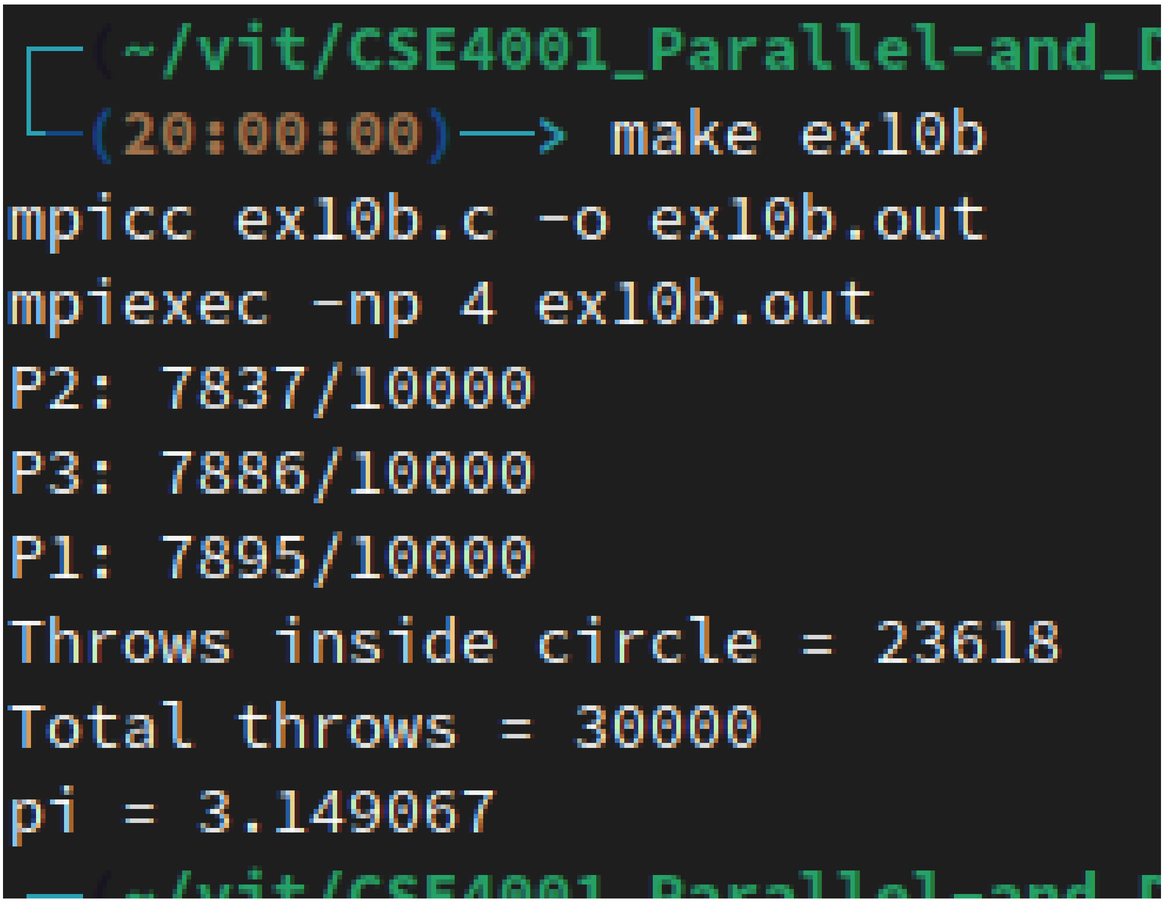


```
/*
b. Write a MPI program to compute PI using “dartboard”
    technique for 1000 rounds by using reduction
    collective computation.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

double getRandom() {
    return (double)(rand() % 1000000) / 500000 - 1;
}

int main(int argc, char *argv[]) {
    srand(clock());
    int procRank, procNum;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    int count = 0, total = 0;
    int finalCount = 0, finalTotal = 0;

    if (procRank != 0) {
        for (int i = 0; i < 10000; i++) {
            double x = getRandom();
            double y = getRandom();
            if ((x*x)+(y*y)< 1) count++;
            total++;
```



```c
            if ((x*x)+(y*y)< 1) count++;
            total++;
        }
        printf("P%d: %d/%d\n", procRank, count, total);
    }
    MPI_Reduce(&count, &finalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&total, &finalTotal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (procRank == 0) {
        printf(
            "Throws inside circle = %d\nTotal throws = %d\npi = %f\n",
            finalCount,
            finalTotal,
            (double)finalCount/finalTotal * 4
        );
    }
    MPI_Finalize();
}
```

Output:



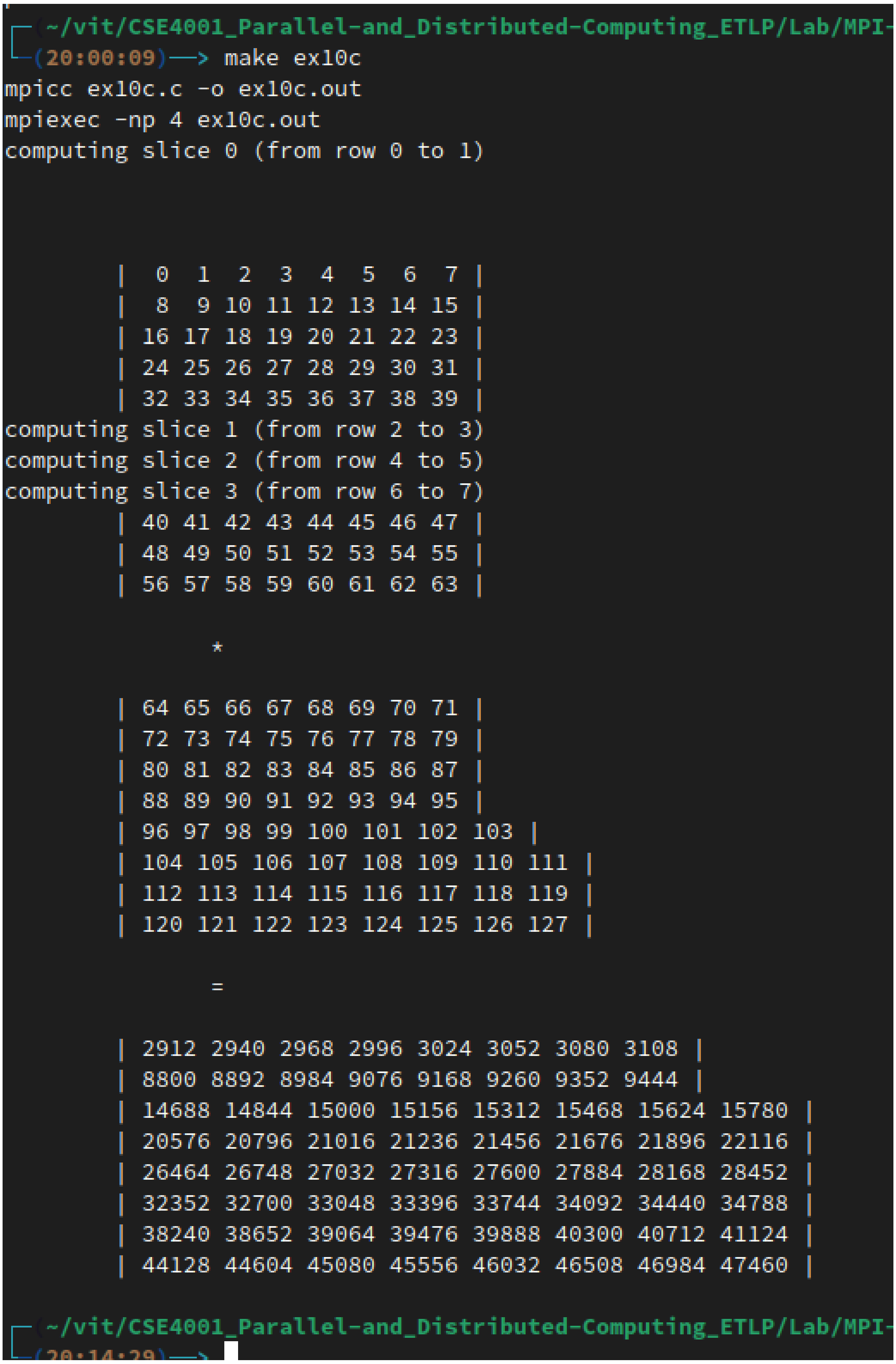
```
┌─(~/vit/CSE4001_Parallel-and_D
└─(20:00:00)──> make ex10b
mpicc ex10b.c -o ex10b.out
mpiexec -np 4 ex10b.out
P2: 7837/10000
P3: 7886/10000
P1: 7895/10000
Throws inside circle = 23618
Total throws = 30000
pi = 3.149067
┌─(~/vit/CSE4001_Parallel-and_D
```

c:

Code:

```
/*
c. Write a MPI program to perform matrix multiplication
(1000x1000) using scatter and gather routines.
*/

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

#define SIZE 8

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill_matrix(int m[SIZE][SIZE])
{
static int n=0;
int i, j;
for (i=0; i<SIZE; i++)
for (j=0; j<SIZE; j++)
m[i][j] = n++;
}

void print_matrix(int m[SIZE][SIZE])
{
int i, j = 0;
for (i=0; i<SIZE; i++) {
printf("\n\t| ");
for (j=0; j<SIZE; j++)
printf("%2d ", m[i][j]);
printf("|");
}
}


int main(int argc, char *argv[])
{
int myrank, P, from, to, i, j, k;
int tag = 666; /* any value will do */
MPI_Status status;
MPI_Init (&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* who am i */
MPI_Comm_size(MPI_COMM_WORLD, &P); /* number of processors */

/* Just to use the simple variants of MPI_Gather and MPI_Scatter we */
/* impose that SIZE is divisible by P. By using the vector versions, */
/* (MPI_Gatherv and MPI_Scatterv) it is easy to drop this restriction. */
if (SIZE%P!=0) {
if (myrank==0) printf("Matrix size not divisible by number of processors\n");
MPI_Finalize();
exit(-1);
}

from = myrank * SIZE/P;
to = (myrank+1) * SIZE/P;

/* Process 0 fills the input matrices and broadcasts them to the rest */
/* (actually, only the relevant stripe of A is sent to each process) */

if (myrank==0) {
fill_matrix(A);
fill_matrix(B);
}

MPI_Bcast (B, SIZE*SIZE, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (A, SIZE*SIZE/P, MPI_INT, A[from], SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);

printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);
for (i=from; i<to; i++)
for (j=0; j<SIZE; j++) {
C[i][j]=0;
for (k=0; k<SIZE; k++)
C[i][j] += A[i][k]*B[k][j];
}

MPI_Gather (C[from], SIZE*SIZE/P, MPI_INT, C, SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);

if (myrank==0) {
printf("\n\n");
print_matrix(A);
printf("\n\n\t * \n");
print_matrix(B);
printf("\n\n\t = \n");
print_matrix(C);
printf("\n\n");
}

MPI_Finalize();
return 0;
}
```

Output:



```
┌─(~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/MPI-
└─(20:00:09)──> make ex10c
mpicc ex10c.c -o ex10c.out
mpiexec -np 4 ex10c.out
computing slice 0 (from row 0 to 1)



        |  0  1  2  3  4  5  6  7 |
        |  8  9 10 11 12 13 14 15 |
        | 16 17 18 19 20 21 22 23 |
        | 24 25 26 27 28 29 30 31 |
        | 32 33 34 35 36 37 38 39 |
computing slice 1 (from row 2 to 3)
computing slice 2 (from row 4 to 5)
computing slice 3 (from row 6 to 7)
        | 40 41 42 43 44 45 46 47 |
        | 48 49 50 51 52 53 54 55 |
        | 56 57 58 59 60 61 62 63 |

               *

        | 64 65 66 67 68 69 70 71 |
        | 72 73 74 75 76 77 78 79 |
        | 80 81 82 83 84 85 86 87 |
        | 88 89 90 91 92 93 94 95 |
        | 96 97 98 99 100 101 102 103 |
        | 104 105 106 107 108 109 110 111 |
        | 112 113 114 115 116 117 118 119 |
        | 120 121 122 123 124 125 126 127 |

               =

        | 2912 2940 2968 2996 3024 3052 3080 3108 |
        | 8800 8892 8984 9076 9168 9260 9352 9444 |
        | 14688 14844 15000 15156 15312 15468 15624 15780 |
        | 20576 20796 21016 21236 21456 21676 21896 22116 |
        | 26464 26748 27032 27316 27600 27884 28168 28452 |
        | 32352 32700 33048 33396 33744 34092 34440 34788 |
        | 38240 38652 39064 39476 39888 40300 40712 41124 |
        | 44128 44604 45080 45556 46032 46508 46984 47460 |

┌─(~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/MPI-
└─(20:14:29)──> █
```