Question:

Click Here (https://www.chegg.com/homework-help/questions-and-answers/assignment-4--input-d0-n-x-n-matrix-0-diaqonal-pooitive-values-places-output-d-x-mathrm-n--q111926854)

Step 1 of 2

Here's an MPI code in C that parallelizes the given sequential pseudo code using the MPI library. The parallelization is done by dividing the D and D0 matrices into submatrices of size n/sqrt(P) x n/sqrt(P) and distributing them across P processes. Each process only computes the elements of its submatrix and communicates with the other processes to exchange the necessary information to compute the final D matrix.

```
1
2     #include <mpi.h>
3     #include <stdio.h>
4     #include <stdlib.h>
5     #include <math.h>
6
7     #define N 100 // size of matrix
8     #define P 4 // number of processes
9     #define SQRT_P 2 // square root of P
10    #define SUB_SIZE N / SQRT_P // size of submatrix
11
12    int main(int argc, char** argv) {
13        int rank, size;
14        int i, j, k, proc_i, proc_j;
15        double D[SUB_SIZE][N] = {0}; // submatrix for D
16        double D0[SUB_SIZE][N] = {0}; // submatrix for D0
17        double sendbuf[SUB_SIZE * N] = {0}; // buffer for sending submatrices
```

```
18        double recvbuf[SUB_SIZE * N] = {0}; // buffer for receiving submatrices
19        MPI_Status status;
20
21        MPI_Init(&argc, &argv);
22        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
23        MPI_Comm_size(MPI_COMM_WORLD, &size);
24
25        if (size != P) {
26            fprintf(stderr, "Number of processes must be equal to P\n");
27            MPI_Finalize();
28            return 1;
29        }
30
31        // Initialize D0 matrix on rank 0 process
32        if (rank == 0) {
33            for (i = 0; i < N; i++) {
34                for (j = 0; j < N; j++) {
35                    if (i != j) {
36                        D0[i / SUB_SIZE][j] = 1.0 + (i + j) / (double)N;
37                    }
38                }
39            }
40        }
41
42        // Distribute D0 matrix to all processes
43        MPI_Scatter(&D0[0][0], SUB_SIZE * N, MPI_DOUBLE, &sendbuf[0], SUB_SIZE * N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
44
45        // Compute submatrix of D
46        for (k = 0; k < N; k++) {
47            for (i = 0; i < SUB_SIZE; i++) {
48                for (j = 0; j < N; j++) {
49                    D[i][j] = fmin(D[i][j], D0[i][k] + D0[k / SUB_SIZE][j]);
50                }
51            }
52
53            // Exchange submatrices
54            proc_i = rank / SQRT_P;
55            proc_j = k / SUB_SIZE;
```

```
56            MPI_Sendrecv(&sendbuf[0], SUB_SIZE * N, MPI_DOUBLE, (proc_i * SQRT_P + (proc_j +
1) % SQRT_P), 0, &recvbuf[0], SUB_SIZE * N, MPI_DOUBLE, (proc_i * SQRT_P + (proc_j - 1 + SQRT_P)
% SQRT_P), 0, MPI_COMM_WORLD, &status);
57
58            // Copy received submatrix to D0
59            for (i = 0; i < SUB_SIZE; i++) {
60                for (j = 0; j < N; j++) {
61                    D0[i][j] = recvbuf[i * N + j];
62                }
63            }
64        }
65
66        // Gather submatrices to form final D matrix on rank 0 process
67        MPI_Gather(&D[0][0], SUB_SIZE * N, MPI_DOUBLE, &D0[0][0], SUB_SIZE * N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
68
69        MPI_Finalize();
70        return 0;
71    }
72
```

Step 2 of 2

he above code is an MPI implementation of the Floyd-Warshall algorithm for finding the shortest path between all pairs of vertices in a weighted graph. The algorithm works by computing the shortest path between each pair of vertices that passes through a given vertex k, and iteratively updating the distances in a matrix D until all pairs of vertices have been considered.

In the MPI implementation, the matrix D is partitioned into submatrices of size n/root P x n/root P, where P is the number of processes and root P is the square root of P. Each process is responsible for computing a submatrix of D and a corresponding submatrix of the input matrix D0, and exchanging submatrices with its neighbors after each iteration of the algorithm.

Explanation:
Here's a breakdown of the code:

1. The MPI library is initialized, and the rank and size of the current process are obtained using MPI_Comm_rank and MPI_Comm_size.
2. The input matrix D0 is initialized on the rank 0 process. In this example, D0 is a square matrix of size N x N with 0 on the diagonal and positive values elsewhere.
3. The input matrix D0 is scattered to all processes using MPI_Scatter. Each process receives a submatrix of size n/root P x N.
4. The Floyd-Warshall algorithm is executed by iterating over the indices k, i, and j. The innermost loop computes the new distance between vertices i and j passing through vertex k, and updates the corresponding entry in the submatrix of D.
5. After each iteration of the algorithm, each process exchanges its submatrix of D0 with its neighbors using MPI_Sendrecv. The submatrix is sent to the neighbor in the positive direction and received from the neighbor in the negative direction.
6. The received submatrix is copied back into the submatrix of D0 on each process.
7. Once all iterations of the algorithm are complete, the submatrices of D are gathered on the rank 0 process using MPI_Gather to form the final matrix D.
8. The MPI library is finalized, and the program exits.

Final Answer

In conclusion, the above code is an MPI implementation of the Floyd-Warshall algorithm for finding the shortest path between all pairs of vertices in a weighted graph, which partitions the input matrix and computes submatrices in parallel across multiple processes.