4. Algorithm Parallelization and Optimization on Distributed Memory Architectures

Team : parlab03

Ioannis Palaios - el18184

I. Introduction

We were assigned to develop parallel and optimized versions of the K-Means algorithm and the Heat-Transfer in two dimensions (only the Jacobi Method) for a distributed memory architecture using MPI as the message-passing standard. For the K-means algorithm, we were assigned to benchmark it for a different number of MPI processes, while for the Jacobi method of the heat-transfer in two dimensions, we were assigned to benchmark it for the cases where we perform a convergence check and when we do not, again for different number of MPI processes and board sizes.

II. MPI parallelization of the K-means algorithm

As we seen in the previous assignments, the compute intensive part of the K-means algorithm happens on the calculation of the new clusters in each iteration of the do while(). In order to efficiently parallelize this part of the algorithm, we split the objects array onto the MPI processes. This means that each process is only assigned to calculate the new distances, cluster memberships and the convergence variable for the specific objects it was assigned. After each iteration, each process has to communicate with ***all*** others in order to renew the *newClusters[]* and *newClusterSize[]* arrays and then each process calculates the average sum of each cluster with the new memberships. Then, each process communicates with ***all*** the other processes in order to get the total sum of the convergence delta variable and then divide it with the number of objects.

In order to assign different objects to each different process in the start of the program, the root MPI process (with the rank equal to 0) is used to calculate the new objects (stored in the *objects[]* array in the *file_io.c* file) and then send different parts of this array to different processes using MPI. For this exact reason, we will use the *MPI_Scatter()* function which achieves exactly this. For this function, we have to calculate the displacements of each data sent (the offset from the start - *displs[]* array) and the number of data sent to each process ((sendcounts[]) array). The process these arrays are calculated is self-evident and is shown in Listing 1, along with others in the *file_io.c* file, solely defining the *dataset_generation()* function.

```
1  double * dataset_generation(int numObjs, int numCoords, long *rank_numObjs)
2  {
3      double * objects = NULL, * rank_objects = NULL;
4      long i, j, k;
5
6      double val_range = 10;
7
8      int rank, size;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(numObjs%size == 0)
        *rank_numObjs = numObjs/size;
    else
        *rank_numObjs = numObjs/size + 1;

    int sendcounts[size], displs[size];
    if (rank == 0) {
        objects = (typeof(objects)) malloc(numObjs * numCoords * sizeof(*objects));
        for(i = 0; i < size; i++) {
            int dim = (*rank_numObjs)*numCoords;
            displs[i] = dim*i;
            if(dim*(i+1) >= numObjs*numCoords)
                sendcounts[i] = (numObjs*numCoords) - dim*i;
            else
                sendcounts[i] = dim;
        }
    }

    MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);

    rank_objects = (typeof(rank_objects)) malloc(sendcounts[rank] * sizeof(*
    rank_objects));

    if (rank == 0) {
        for (i=0; i<numObjs; i++)
        {
            unsigned int seed = i;
            for (j=0; j<numCoords; j++)
            {
                objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) *
    val_range;
                if (_debug && i == 0)
                    printf("object[i=%ld][j=%ld]=%f\n",i,j,objects[i*numCoords + j])
    ;
            }
        }
    }

    int recvsize = (*rank_numObjs)*numCoords;
    MPI_Scatterv(objects, sendcounts, displs, MPI_DOUBLE, rank_objects, recvsize,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if(rank == 0)
        free(objects);

    return rank_objects;
}
```

Listing 1: The source code of file_io.c. In this file the dataset_generation() method is defined which essentially generates the new objects and scatters them to all the different processes.

Then the *dataset_generation()* function is called in the main function (*main.c*) and places the objects to each corresponding MPI process. Later in main.c, the *clusters[]* array is allocated and initialized in the root process and then broadcasted. Then, the *kmeans()* function is called, where the computations of the K-means algorithm happens and the clusters array is already calculated in each process so no other communication needs to happen for the clusters (the communication happens in the *kmeans()* function) but the root process has to gather the total

membership information from all the other processes, so we have to calculate the number of elements to receive from each other rank and the displacement of each rank's data (*recvcounts[]* and *displs[]* respectively). The important part of the source code of main.c is shown on Listing 2 below.

```
int main(int argc, char **argv) {
    // variable declarations etc.
        ...

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    numObjs = (dataset_size*1024*1024) / (numCoords*sizeof(double));

    objects = dataset_generation(numObjs, numCoords, &rank_numObjs);

    clusters = (double*)  malloc(numClusters * numCoords * sizeof(double));

    if (rank == 0) {
        for (i=0; i<numClusters; i++)
            for (j=0; j<numCoords; j++)
                clusters[i*numCoords + j] = objects[i*numCoords + j];

        // check initial cluster centers for repetition
        if (check_repeated_clusters(numClusters, numCoords, clusters) == 0) {
            printf("Error: some initial clusters are repeated. Please select
    distinct initial centers\n");
            MPI_Finalize();
            return 1;
        }

        /*
        printf("Initial cluster centers:\n");
        for (i=0; i<numClusters; i++) {
            printf("(0) clusters[%ld] =",i);
            for (j=0; j<numCoords; j++)
                printf(" %6.6f", clusters[i*numCoords + j]);
            printf("\n");
        } */

    }

    MPI_Bcast(clusters, numClusters*numCoords, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    membership = (int*) malloc(rank_numObjs * sizeof(int));
    tot_membership = (int*) malloc(numObjs * sizeof(int));

    double start_time = wtime();
    kmeans(objects, numCoords, rank_numObjs, numClusters, threshold, loop_threshold,
     membership, clusters);
    double time = wtime() - start_time;

    if (rank == 0) {
        printf("Final cluster centers:\n");
        for (i=0; i<2; i++) {
            printf("clusters[%ld] = ",i);
            for (j=0; j<numCoords; j++)
                printf("%6.6f ", clusters[i*numCoords + j]);
            printf("\n");
```

```
        }
        printf("\n%d MPI Processes | Time Spent : %7.4f sec\n", size, time);
    }

    int recvcounts[size], displs[size];
    if (rank == 0) {
        for(i = 0; i < size; i++) {
            long dim = rank_numObjs;
            displs[i] = dim*i;
            if(dim*(i+1) >= numObjs)
                recvcounts[i] = numObjs - dim*i;
            else
                recvcounts[i] = dim;
        }
    }

    MPI_Bcast(recvcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Gatherv(membership, rank_numObjs, MPI_INT, tot_membership, recvcounts,
    displs, MPI_INT, 0, MPI_COMM_WORLD);

    // free the dynamically allocated arrays
        ...
    return 0;
}
```

Listing 2: The source code of the main.c file.

Now, we have to perform the calculations of the K-means algorithm. As we mentioned above, each process is responsible for calculating the distances from the current clusters etc. only for the objects it was assigned. The only reasons it must communicate with the other process is the renew of the *newClusters[]* and *newClusterSize[]* arrays and the *delta* variable. As we have to calculate the sum of the each of the corresponding element of each rank's local *rank_newClusters[]* and *rank_newClusterSize[]* arrays and then store the result on the textitnewClusters[] and *newClusterSize[]* arrays respectively and then have them stored in each process (in order for each process to be able to calculate the same *clusters[]* array), we will the *MPI_Allreduce()* function which does exactly that. The do while() of the *kmeans.c* file is shown on Listing 3 below.

```
do {
     for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            rank_newClusters[i*numCoords + j] = 0.0;
            newClusters[i*numCoords + j] = 0.0;
        }
        rank_newClusterSize[i] = 0;
        newClusterSize[i] = 0;
    }
    delta = 0.0;
    rank_delta = 0.0;

    for (i=0; i<numObjs; i++) {
        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
    clusters);

        if (membership[i] != index)
            rank_delta += 1.0;

        membership[i] = index;

```

```
        rank_newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            rank_newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }

    MPI_Allreduce(rank_newClusters, newClusters, numClusters*numCoords, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM
    , MPI_COMM_WORLD);

    for (i=0; i<numClusters; i++) {
        if (newClusterSize[i] > 0) {
            for (j=0; j<numCoords; j++) {
                clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
    newClusterSize[i];
            }
        }
    }
    MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    delta /= numObjs;
    loop++;
} while (delta > threshold && loop < loop_threshold);
```

Listing 3: The source code of the do while() in the kmeans.c file which is responsible for the computations (and communications) of the K-means algorithm.

We benchmark our MPI implementation for the configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} and for different numbers of MPI processes, such as 1, 2, 4, 8, 16, 32 and 64. As we observe on the Figures 1 and 2 below, as we increase the number of MPI processes, we achieve significant performance benefits, almost in accordance with the number of Processes we create. The reason we achieve this performance scaling as we increase the number of processes is that the time spent for the communication between these processes is likely a lot less than the performance benefits of the parallelization of the compute intensive part of the algorithm. Another benefit in comparison with the shared memory architecture implementation (where we used OpenMP), is that each processes has a different memory address space and thus operates on its own data reducing memory conflicts, cache misses, cache contention problems, false sharing and other shared memory performance degrading phenomena. Another reason is that these MPI processes we create, are all on the same node which means that the communication between them, has lower latency and higher bandwidth than the case where these processes were located on another node. Of course we do not expect similar performance benefits (or scaling) when we create more processes than our node's virtual cores as context switching between them (by the Operating System) will likely introduce a large overhead. As we mentioned on the second assignment, the node we run these experiment has 32 physical cores and 64 virtual cores (by supporting hyper-threading). In the OpenMP implementation, we did not see sufficient scaling between increasing the number of threads from 32 to 64 as each of these threads (and specifically the threads occupying the same physical core) were using the same parts of the physical core and thus hyper-threading could not add performance benefits. This phenomenon does not happen with OpenMP as processes are likely having some benefits as resource isolation leading to less contention of shared resources and different memory access patterns compared to multi thread implementations.

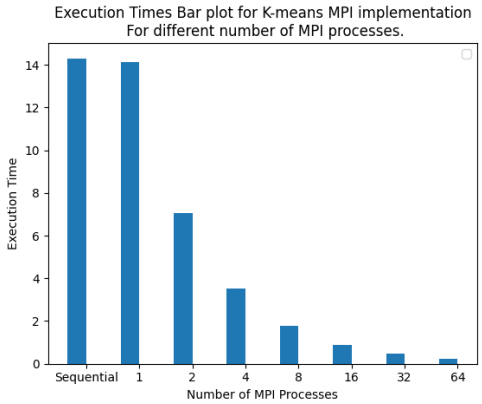

Figure 1: Bar Plot of the execution times of the K-Means Clustering Algorithm MPI implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} for different numbers of MPI processes.

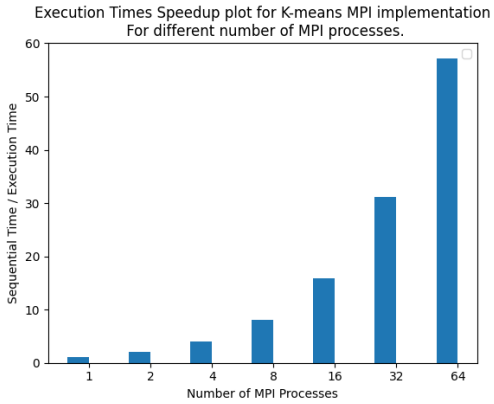

Figure 2: Speedup Plot of the execution times of the K-Means Clustering Algorithm MPI implementation for the Configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} for different numbers of MPI processes.

III. Heat Transfer - Jacobi Method

For this version, we had to solve the 2D Heat equation using the Jacobi Method. We want to parallelize the compute intensive part of the heat equation using multiple MPI processes. In order to do that, we have to split the data to the multiple processes with each process taking a 'block' and then performing the calculations of the temperature of each block separately. But, when we want to calculate the temperature change of this specific block, we have to take the temperature values of its neighbouring blocks (north, south, west and east if they exist) which are stored in the 'neighbouring' processes and as a result processes must communicate in each step of the calculating for loop.

First of all, the root process calls the *init2d()* function in order to initialize the double array containing the temperatures in each cell (only the boundary values have a temperature different than 0). Then, we have to send to each process their corresponding blocks. For this splitting we

use the *MPI_Scatterv()* function which send different data to each process. For many reasons including code readability etc, we assign each process to an MPI Cartesian Communicator which gives each process a 2 dimensional rank, thus assigning it to a specific block. For similar reasons, specific data types are defined for the *MPI_Scatterv()* call. The source code of this specific part is shown on Listing 4 below.

```
if (rank==0) {
    U=allocate2d(global_padded[0],global_padded[1]);
    init2d(U,global[0],global[1]);
}

u_previous=allocate2d(local[0]+2,local[1]+2);
u_current=allocate2d(local[0]+2,local[1]+2);

MPI_Datatype global_block;
MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
MPI_Type_commit(&global_block);

MPI_Datatype local_block;
MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
MPI_Type_commit(&local_block);

int * scatteroffset, * scattercounts;
if (rank==0) {
    scatteroffset=(int*)malloc(size*sizeof(int));
    scattercounts=(int*)malloc(size*sizeof(int));
    for (i=0;i<grid[0];i++)
        for (j=0;j<grid[1];j++) {
            scattercounts[i*grid[1]+j]=1;
            scatteroffset[i*grid[1]+j]=(local[0]*local[1]*grid[1]*i+local[1]*j);
        }
}
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_previous
    [1][1]), 1, local_block, 0, CART_COMM);

for(i = 0; i < local[0]+2; i++) {
    for(j = 0; j < local[1]+2; j++) {
        u_current[i][j] = u_previous[i][j];
    }
}
```

Listing 4: The source code of most of the matrix initilizations and the initial scatter.

Then, when the blocks are sent to their corresponding processes, each process has to find the coordinates of it's neighbours while also taking into account the possibility that the process is in the boundary of the global block and does not have one or two neighbours. Then, each block has to find the ranks of the neighbouring processes (from their coordinates) in order to communicate with them. The source code of this specific part is shown on Listing 5 below.

```
int row_idx = rank_grid[0], col_idx = rank_grid[1];
int *num_of_rows = (int*) malloc(sizeof(int));
int *num_of_cols = (int*) malloc(sizeof(int));
*num_of_rows = local[0];
*num_of_cols = local[1];
int *row_size = (int*) malloc(sizeof(int));
int *col_size = (int*) malloc(sizeof(int));
*row_size = local[0]+2;
*col_size = local[1]+2;
```

```

// send element arrays
double *north_arr = (double*) malloc(*num_of_cols * sizeof(double));
double *south_arr = (double*) malloc(*num_of_cols * sizeof(double));
double *east_arr = (double*) malloc(*num_of_rows * sizeof(double));
double *west_arr = (double*) malloc(*num_of_rows * sizeof(double));

// receive element arrays
double *north_arr_recv = (double*) malloc((*num_of_cols) * sizeof(double));
double *south_arr_recv = (double*) malloc((*num_of_cols) * sizeof(double));
double *east_arr_recv = (double*) malloc((*num_of_rows) * sizeof(double));
double *west_arr_recv = (double*) malloc((*num_of_rows) * sizeof(double));

int north, south, east, west;
north = row_idx - 1;
south = row_idx + 1;
east = col_idx + 1;
west = col_idx - 1;

 int i_min,i_max,j_min,j_max;

/*Fill your code here*/

// if is west boundary
if(west < 0) {
    j_min = 2;
} else {
    j_min = 1;
}

// if is east boundary
if(east >= grid[1]) {
    j_max = *num_of_cols-1;
} else {
    j_max = *num_of_cols;
}

// if is north boundary
if(north < 0) {
    i_min = 2;
} else {
    i_min = 1;
}

// if is south boundary
if(south >= grid[0]) {
    i_max = *num_of_rows-1;
} else {
    i_max = *num_of_rows;
}

 int *cart_coords_north = (int*) malloc(2 * sizeof(int));
int *cart_coords_south = (int*) malloc(2 * sizeof(int));
int *cart_coords_west = (int*) malloc(2 * sizeof(int));
int *cart_coords_east = (int*) malloc(2 * sizeof(int));

cart_coords_north[0] = row_idx-1;
cart_coords_north[1] = col_idx;
cart_coords_south[0] = row_idx+1;
cart_coords_south[1] = col_idx;
cart_coords_east[0] = row_idx;
```

```
cart_coords_east[1] = col_idx+1;
cart_coords_west[0] = row_idx;
cart_coords_west[1] = col_idx-1;

//*********************************//

MPI_Status *north_stats = (MPI_Status*) malloc(2 *sizeof(MPI_Status));
MPI_Status *south_stats = (MPI_Status*) malloc(2 *sizeof(MPI_Status));
MPI_Request *north_reqs = (MPI_Request*) malloc(2 *sizeof(MPI_Request));
MPI_Request *south_reqs = (MPI_Request*) malloc(2 *sizeof(MPI_Request));
MPI_Status *east_stats = (MPI_Status*) malloc(2 *sizeof(MPI_Status));
MPI_Status *west_stats = (MPI_Status*) malloc(2 *sizeof(MPI_Status));
MPI_Request *east_reqs = (MPI_Request*) malloc(2 *sizeof(MPI_Request));
MPI_Request *west_reqs = (MPI_Request*) malloc(2 *sizeof(MPI_Request));

int *tags = (int*) malloc(4 * sizeof(int));
for(i = 0; i < 4; i++)
    tags[i] = 0;
int *c = (int*) malloc(1 * sizeof(int));

int *cart_rank_north = (int*) malloc(1 * sizeof(int));
int *cart_rank_south = (int*) malloc(1 * sizeof(int));
int *cart_rank_east = (int*) malloc(1 * sizeof(int));
int *cart_rank_west = (int*) malloc(1 * sizeof(int));

*cart_rank_north = -1;
*cart_rank_south = -1;
*cart_rank_east = -1;
*cart_rank_west = -1;

int coord_var = grid[0];
if(cart_coords_north[0]>=0) {
    *cart_rank_north = coord_var*cart_coords_north[0] + cart_coords_north[1];
}
if(cart_coords_south[0]<grid[0]) {
    *cart_rank_south = coord_var*cart_coords_south[0] + cart_coords_south[1];
}
if(cart_coords_west[1]>=0) {
    *cart_rank_west = coord_var*cart_coords_west[0] + cart_coords_west[1];
}
if(cart_coords_east[1]<grid[1]) {
    *cart_rank_east = coord_var*cart_coords_east[0] + cart_coords_east[1];
}
```

Listing 5: The source code of the assignment of coordinates and ranks to the neighbouring processes etc.

After, we have the neighbouring information, we move onto the compute intensive part which is the for loop (of default size equal to 256 iterations). We use non blocking Sends and Receives (*MPI_Isend()* and *MPI_Irecv()* respectively) and then we use the *MPI_Waitall()* method in order to make sure each process has received its corresponding boundary parts. Then, we perform the convergence check in each process and if all processes have converged then the for loop finishes. We use *MPI_Allreduce()* and the MPI operation *MPI_MIN*, reducing the converged value with the minimum operation in order to make sure that if one process block has not converged (*converged* variable equal to 0), all the other processes/blocks have not converged either. The source code of the for loop is shown on Listing 7 below.

```
gettimeofday(&tts, NULL);
    #ifdef TEST_CONV
```

```
    for (t=0;t<T && !global_converged;t++) {
    #endif
    #ifndef TEST_CONV
    #undef T
    #define T 256
    for (t=0;t<T;t++) {
    #endif
        // initialize boundary arrays
        for(i = 1; i < (*num_of_cols)+1; i++) {
            north_arr[i-1] = u_previous[1][i];
            south_arr[i-1] = u_previous[*num_of_rows][i];
        }

        for(i = 1; i < (*num_of_rows)+1; i++) {
            west_arr[i-1] = u_previous[i][1];
            east_arr[i-1] = u_previous[i][*num_of_cols];
        }

        // sending phase
        if(*cart_rank_north > -1) {
            MPI_Isend(north_arr, *num_of_cols, MPI_DOUBLE, *cart_rank_north, tags
    [0], CART_COMM, &(north_reqs[0]));
        }
        if(*cart_rank_south > -1) {
            MPI_Isend(south_arr, *num_of_cols, MPI_DOUBLE, *cart_rank_south, tags
    [1], CART_COMM, &(south_reqs[0]));
        }
        if(*cart_rank_west > -1) {
            MPI_Isend(west_arr, *num_of_rows, MPI_DOUBLE, *cart_rank_west, tags[2],
    CART_COMM, &(west_reqs[0]));
        }
        if(*cart_rank_east > -1) {
            MPI_Isend(east_arr, *num_of_rows, MPI_DOUBLE, *cart_rank_east, tags[3],
    CART_COMM, &(east_reqs[0]));
        }

         // receiving phase
        if(*cart_rank_north > -1) {
            MPI_Irecv(north_arr_recv, *num_of_cols, MPI_DOUBLE, *cart_rank_north,
    tags[0], CART_COMM, &north_reqs[1]);
        }
        if(*cart_rank_south > -1) {
            MPI_Irecv(south_arr_recv, *num_of_cols, MPI_DOUBLE, *cart_rank_south,
    tags[1], CART_COMM, &south_reqs[1]);
        }
        if(*cart_rank_west > -1) {
            MPI_Irecv(west_arr_recv, *num_of_rows, MPI_DOUBLE, *cart_rank_west, tags
    [2], CART_COMM, &west_reqs[1]);
        }
        if(*cart_rank_east > -1) {
            MPI_Irecv(east_arr_recv, *num_of_rows, MPI_DOUBLE, *cart_rank_east, tags
    [3], CART_COMM, &east_reqs[1]);
        }

        // process synchronization
        if(*cart_rank_north > -1) {
            MPI_Waitall(2, north_reqs, north_stats);
        }
        if(*cart_rank_south > -1) {
            MPI_Waitall(2, south_reqs, south_stats);
        }
```

```
        if(*cart_rank_west > -1) {
            MPI_Waitall(2, west_reqs, west_stats);
        }
        if(*cart_rank_east > -1) {
            MPI_Waitall(2, east_reqs, east_stats);
        }

        /*Compute and Communicate*/

        /*Add appropriate timers for computation*/

        gettimeofday(&tcs, NULL);
        for(i = i_min; i < i_max; i++) {
            for(j = j_min; j < j_max; j++) {
                u_current[i][j] = (u_previous[i-1][j] + u_previous[i][j-1]+
    u_previous[i+1][j]+u_previous[i][j+1])/4;
            }
        }
        gettimeofday(&tcf, NULL);
        tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

         #ifdef TEST_CONV
        if (t%C==0) {
            gettimeofday(&tconvs, NULL);
            converged=1;
            for(i = i_min; i < i_max; i++) {
                for(j = j_min; j < j_max; j++) {
                    if(abs(u_current[i][j]-u_previous[i][j])>0.01) {
                        converged=0;
                        i = i_max;
                        j = j_max;
                        break;
                    }
                }
            }
            gettimeofday(&tconvf, NULL);
            tconv += (tconvf.tv_sec-tconvs.tv_sec)+(tconvf.tv_usec-tconvs.tv_usec)
    *0.000001;
        }
        #endif
        for(i = i_min; i < i_max; i++) {
            for(j = j_min; j < j_max; j++) {
                u_previous[i][j] = u_current[i][j];
           }
        }
        //************************************//
        MPI_Allreduce(&converged, &global_converged, 1, MPI_DOUBLE, MPI_MIN,
    CART_COMM);
    }
```

Listing 6: The source code of the compute intensive for loop.

Finally, in order to gather the local matrices that are stored in each process back to the global array, we use *MPI_Gatherv()* with similar offsets and counts as the original scatter performed in the beginning of the program.

```
// identical to the scatter above
int * gatheroffset, * gathercounts;
if (rank==0) {

    gatheroffset=(int*)malloc(size*sizeof(int));
```

```
    gathercounts=(int*)malloc(size*sizeof(int));

    for (i=0;i<grid[0];i++)
        for (j=0;j<grid[1];j++) {
            gathercounts[i*grid[1]+j]=1;
            gatheroffset[i*grid[1]+j]=(local[0]*local[1]*grid[1]*i+local[1]*j);
        }
    gatheroffset[(grid[0]-1)*grid[1]+grid[1]-1] -= 1;
}

MPI_Bcast(gatheroffset, size, MPI_INT, 0, CART_COMM);
MPI_Bcast(gathercounts, size, MPI_INT, 0, CART_COMM);

MPI_Gatherv(&(u_current[1][1]), 1, local_block, &(U[0][0]), gathercounts,
    gatheroffset, global_block, 0, CART_COMM);
```

Listing 7: The source code of most of the matrix initilizations and the initial scatter.

We were assigned to benchmark the Jacobi method for 1024x1024 board size and 64 MPI processes. We also leave the convergence check variable called C at it's default value of 100 although it might be beneficial to reduce it. The execution time of this benchmark is 0.206033, the computation time is 0.025395, the communication time is 0.180638 and time spent on the converge check is 0.000319. It converges after the first 100 iterations and this means that each process gets into the convergence check exactly two times. We observe that the time spent for communication is almost 7x bigger than the time spent for communication. This is reasonable as most processes have to communicate with 4 other processes, in order to receive and send data (degrading performance due to the waiting) , while the computational part is performing a double nested for loop with 16384 iterations.

For the benchmarks with no convergence, we were assigned to benchmark this version for a constant number of 256 iterations for board sizes of 2048x2048, 4096x4096, 6144x6144 and 1, 2, 4, 8, 16, 32 and 64 MPI processes. We observe that our program increases it's performance when we increase the number of MPI Processes. This happens due to that while the workload that each process gets is smaller than when we had less processes, the communication overhead also gets proportionately less as the size of the data that get transferred are also less. The execution time bar plot is shown on Figure 2.

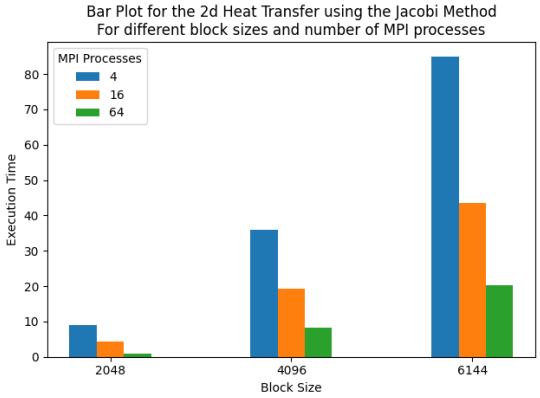

Figure 3: Bar Plot of the execution times of the 2d Heat Transfer with the Jacobi Method using 4, 16 and 64 MPI processes for the block sizes : 2048x2048, 4096x4096, 6144x6144.

We also observe that, as we increase the number of processes the fraction of communication

time divided by the total time gets bigger. Namely, for 4 processes the fraction is around 50%, for 16 processes around 55% and for 64 processes it is around 60%. This happens due to that applying the Jacobi method to smaller blocks decreases the computation time, while the communication time does not decrease as drastically.

IV. Conclusion

In this assignment, we observed how different algorithm can be optimized and parallelized in a distributed memory environment, using MPI. First, we parallelized the K-means algorithm with MPI exploring how to efficiently distribute the data to multiple processes and communicate in each iteration in order to reduce an array and make it available to all the processes. We observed significant performance benefits in comparison to the shared memory architecture implementation for reason we also observed and explained. Lastly, we implemented and optimized a parallel version of the 2d Heat Equation using the Jacobi method. After a number of weird program behavior and *a lot of segmentation faults*, we figured out how to efficiently distribute and gather data in a fault tolerant manner while also understanding how inter process communication can be used in a geometric fashion (2d grid) in order to solve a partial differential equation in a parallel and distributed memory fashion. Lastly, we gathered our results and plotted them, visualizing our results and comparing them.