CMPE300

ANALYSIS OF ALGORITHMS

Seçkin Savaşçı
savasci@acm.org
2008400078

Barış Kurt

Programming Project:

Parallel K-means Algorithm Implementation in C using MPI

January 10th, 2011

Introduction

In this project I implemented a parallel C algorithm using MPI (Message Passing Interface) classes. The algorithm to have been implemented is a simple machine learning algorithm to divide given data points into groups according to a distance measure. In this project I used 2D Cartesian points and Euclidean distances. A detailed info about the algorithm can be found in project description file which is included to .zip file containing this report, and in Wikipedia.

Programming Interface – Input & Output

Both compilation & run commands for linux terminal environment are those;

The zip containing this report includes 2008400078.c, which can be compiled as;

mpicc -g 2008400078.c -o <name_for_the_executable_file>

And the resulting executable file can be run as;

mpiexec -n <num_of_processors>
./<name_for_the_executable_file> <input_file> <output_file>

Where the input file is a text file with the format:

NUM_CLUSTERS
NUM_DATA_POINTS
POINT_1_X,POINT_1_Y
POINT_2_X,POINT_2_Y
...
...
POINT_2_N,POINT_N_Y

And the output file is another text file with the format:

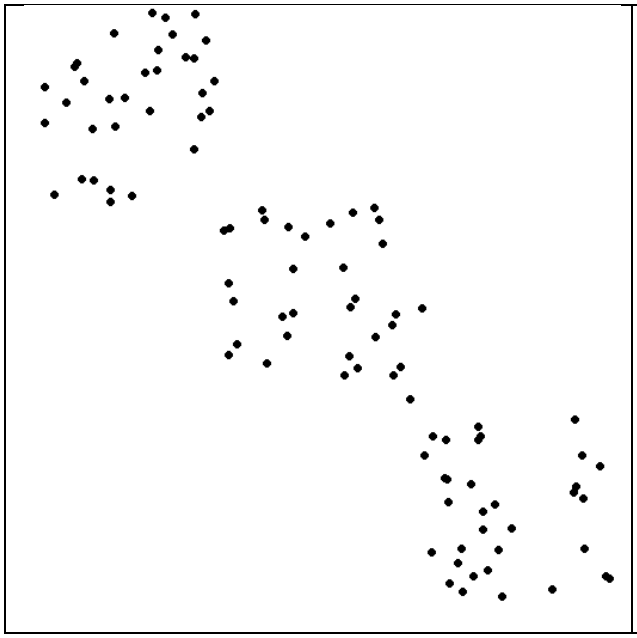
NUM_CLUSTERS
NUM_DATA_POINTS
CENTROID_1_X,CENTROID_1_Y
CENTROID_2_X,CENTROID_2_Y
…
...

...
CENTROID_k_X,CENTROID_k_Y
POINT_1_X,POINT_1_Y,CLUSTER_VALUE
POINT_2_X,POINT_2_Y,CLUSTER_VALUE
...
...
POINT_2_N,POINT_N_Y,CLUSTER_VALUE

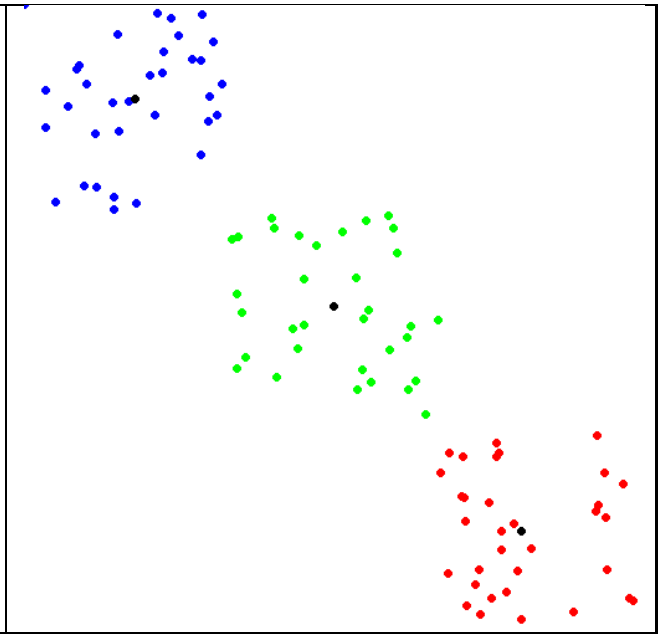
Program Execution – Examples

Program simply clusters the given points in given number of clusters in the input file. Output file can be visualized to make the output clear for the end-user. For the output file, Colored dots are original points, where the black ones are the mean of the clusters. These input & output files are included to the .zip file.

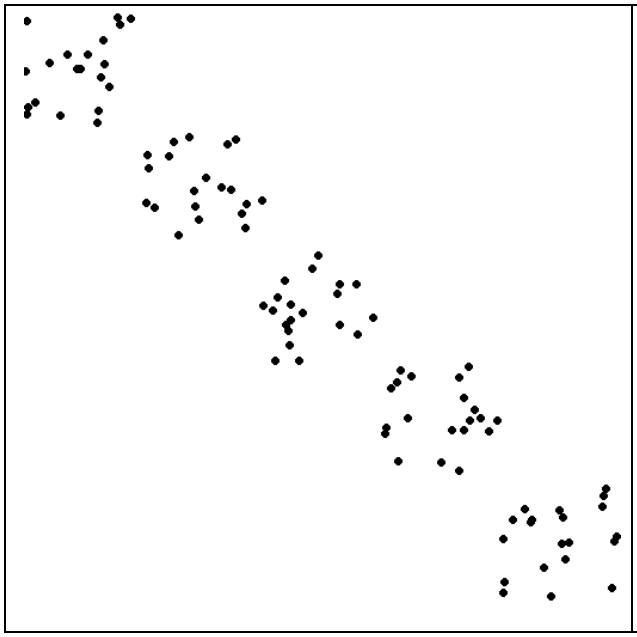
Input1.txt



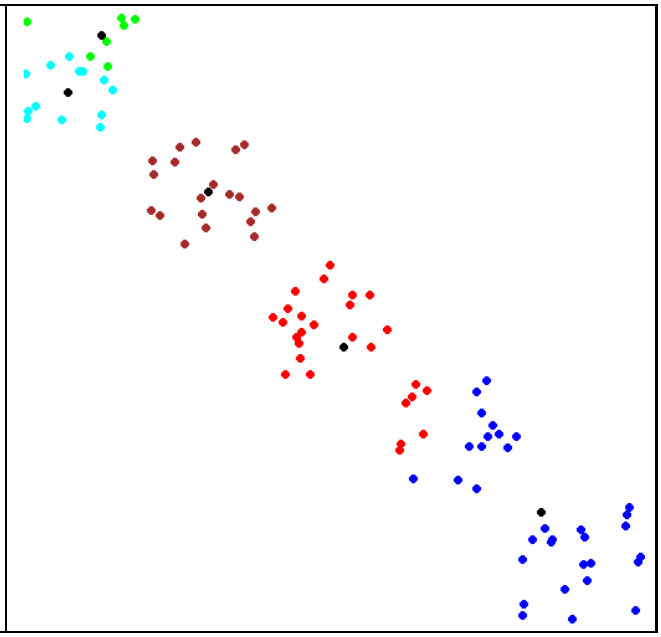
Output1.txt



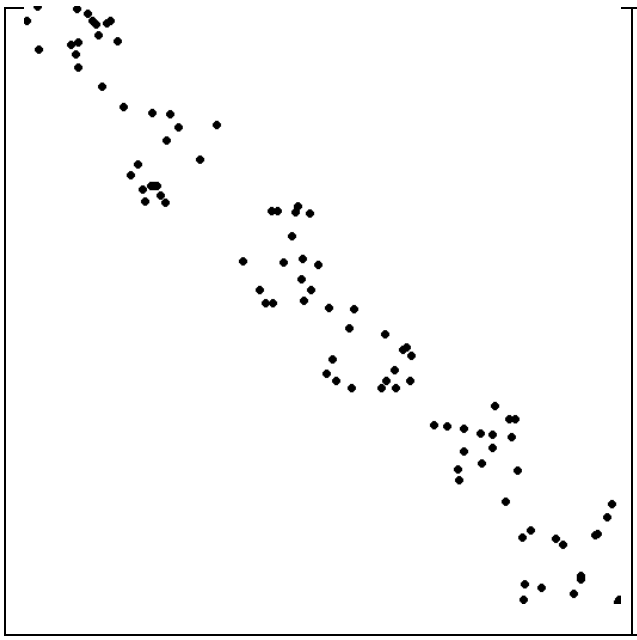
Input2.txt



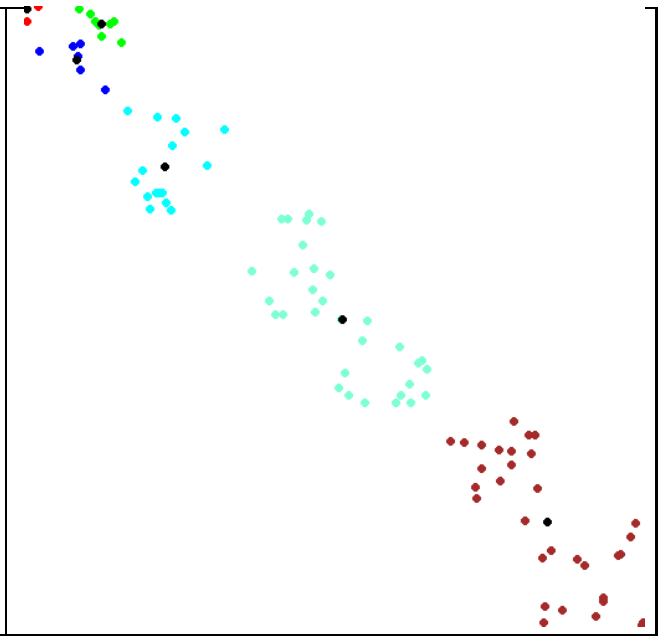
Output2.txt



Input3.txt



Output3.txt



Before Executing the program you must ensure that processor number is a divider of the number of points in the input file. There are no any other restrictions to consider when running this project code. But if you have any questions, feel free to ask me!

Program Structure

This project is developed with considering the power plays of both C language and MPI. From the side of C, program code is divided into functions as much as possible for modularity, dynamic memory allocation is used to minimize the unused memory, a structure is created for storing a 2d point,commented heavily for further reuse. For higher precision, double data type is used in point structure.
From the MPI side, MPI functions are stated as clear as possible, A new derived data type is created for sending the point structures. Furthermore MPI_Barrier is used for ensuring the synchronization.

A simple lifecycle of a program can be listed as:

- Program starts
- Program reads the header info from input file
- Allocates needed data for input of the points
- Reads the points & store them
- Randomly initialize centroids
- Resets the cluster assignments for each point
- <loop>Finds the closest centroids to each file
- Recalculates the centroids as taking the mean of the closest points
- Checks the convergence status, as comparing the new and old cluster assignment of each point
- If converged program goes to next step; else jumps to <loop>
- Program writes the output file with described format in this report
- Program ends

As you can see, there is no deallocation of the memory. At first, it can be seen as the programmer's lack of use, but all allocated memory is used during the execution and there isn't actually a real need for deallocation because program ends when the memory becomes deallocatable for the most memory blocks.

This code is written in C but with using MPI; so in the processor side of the work we must explain enough. There established a master-slave relationship between the processors. A processor acts as the master and does the main part of the job. Only work done by slave ones are finding the closest centroid. If we look at the message passing side, we can list the life cycle as:

- Master partitions & sends the points to slaves
- Master sends the random centroids to slaves
- <loop>Slaves find the closest centroids for each point
- Slaves send the array of assignments to clusters for points to master
- Master checks for convergence, comparing old & new assignments to clusters
- If converged sends exit signal to slaves & writes the output & ends, otherwise sends continue signal
- If exit signal is received, slaves end execution; otherwise waits
- Master calculates new centroids & send them to slaves
- Execution jumps to <loop>

Improvements & Extensions – Difficulties Encountered

The program fails to make partitions with unequal sizes. For example, 100 points cannot be partitioned to three processors with this source code (33+33+34). So main improvement can be done in this point. Also information messages are printed to STDOUT, but in printing synchronizations problems are exist. And printing order is not fully established. This could be another good improvement to current development.

At early stage of the development, I used structures to message passing to processors, but this heavy traffic of data makes the program execution unmanageable as a debugger. So I reduced my message traffic as using predefined data types. Also, first I tried to use broadcasting system of MPI, namely MPI_BCast, but its unblocking behavior also made the program code untraceable in long run. In final source code, only MPI_Send & MPI_Recv are used, but if MPI_BCast can be implemented to this system, performance gain for sending the common data as a tree traversal can be achieved.

Conclusion

Failed or not, it was a huge experience gained as developing for parallel systems. Also blocking-nonblocking messaging, broadcasting etc. are good topics to dive in, that I understood while making this project. Parallelization is a hard but joyful process, in my opinion.

Appendices

Source Code

```
/**
 * @author Seckin Savasci
 * @contact savasci@acm.org
 * @info 2008400078 BOUN,Cmpe300,Programming project
 * @desc Parallel K-means Algorithm Implemantation in C using MPI
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "mpi.h"
#define MASTER 0
/**
 * Point structure for point data
 */
typedef struct
     {
          double _x;
          double _y;
     } Point;
/**
 * reader function of the input file's first(for number of clusters)
 * & second(for number of points) line
 * @param input input file handler
 * @param num_clusters pointer to return number of clusters
 * @param num_points pointer to return number of points
 */
void readHeaders(FILE *input,int* num_clusters,int* num_points)
{
     fscanf(input,"%d\n",num_clusters);
     printf("%d\n",*num_clusters);

     fscanf(input,"%d\n",num_points);
     printf("%d\n",*num_points);
}
/**
 * reader function of the points in the input file
 * This function must be called after  readHeaders(...) function
 * @param input input file handler
 * @param points pointer to return the array of points
 * @param num_points number of points to read
 */
void readPoints(FILE* input,Point *points,int num_points)
{
     int dex;
     for(dex=0;dex<num_points;dex++)
```

```
	{
		fscanf(input,"%lf,%lf",&points[dex]._x,&points[dex]._y);
	}
}
/**
 * initializer function that randomly initialize the centroids
 * @param centroids pointer to return array of centroids
 * @param num_cluster number of clusters(so number of centroids, too)
 */
void initialize(Point* centroids,int num_clusters)
{
	int dex;
	srand(time(NULL));
	for(dex=0;dex<num_clusters;dex++)
	{
		centroids[dex]._x=((double)(rand()%1000))/1000;
		centroids[dex]._y=((double)(2*rand()%1000))/1000;
	}
}
/**
 * initializer function that initializes the all cluster array values
to -1
 * @param data pointer to return array of cluster data
 * @param num_points number of points to initialize
 */
int resetData(int *data,int num_points)
{
	int dex;
	for(dex=0;dex<num_points;dex++)
	{
		data[dex]=-1;
	}
}
/**
 * calculate distance between two points
 * @param point1 first point
 * @param point2 second point
 * @return distance in double precision
 */
double calculateDistance(Point point1,Point point2)
{
	return (pow((point1._x-point2._x)*100,2)+pow((point1._y-
point2._y)*100,2));
}
/**
 * Wierd name but essential function; decides witch centroid is closer
to the given point
 * @param point point given
 * @param centroids pointer to centroids array
 * @param num_centroids number of centroids to check
 * @return closest centroid's index in centroids array(2nd param)
 */
```

```
int whoIsYourDaddy(Point point,Point* centroids,int num_centroids)
{
	int daddy=0;
	double distance=0;
	double minDistance=calculateDistance(point,centroids[0]);
	int dex;

	for(dex=1;dex<num_centroids;dex++)
	{
		distance=calculateDistance(point,centroids[dex]);
		if(minDistance>=distance)
		{
			daddy=dex;
			minDistance=distance;
		}
	}
	return daddy;
}
/**
 * Cumulative function that must be called after the closest centroid
for each point is found
 * Calculates new centroids as describen in kmeans algorithm
 * @param points array of points
 * @param data array of cluster assignments
 * @param centroids return array of centroids
 * @param num_clusters number of clusters(so number of centroids)
 * @param num_points number of points
 */
void calculateNewCentroids(Point* points,int* data,Point*
centroids,int num_clusters,int num_points)
{
	Point* newCentroids=malloc(sizeof(Point)*num_clusters);
	int* population=malloc(sizeof(int)*num_clusters);
	int dex;

	for(dex=0;dex<num_clusters;dex++)
	{
		population[dex]=0;
		newCentroids[dex]._x=0;
		newCentroids[dex]._y=0;
	}
	for(dex=0;dex<num_points;dex++)
	{
		population[data[dex]]++;
		newCentroids[data[dex]]._x+=points[dex]._x;
		newCentroids[data[dex]]._y+=points[dex]._y;
	}
	for(dex=0;dex<num_clusters;dex++)
	{
		if(population[dex]!=0.0)
		{
			newCentroids[dex]._x/=population[dex];
```

```
                newCentroids[dex]._y/=population[dex];
            }
        }
        for(dex=0;dex<num_clusters;dex++)
        {
            centroids[dex]._x=newCentroids[dex]._x;
            centroids[dex]._y=newCentroids[dex]._y;
        }
}
/**
 * Convergence checker (see project description for further info)
 * @param former_clusters pointer to array of older cluster
assignments
 * @param latter_clusters pointer to array of newer cluster
assignments
 * @param num_points number of points
 * @return -1 if not converged, 0 if converged.
 */
int checkConvergence(int *former_clusters,int *latter_clusters,int
num_points)
{
        int dex;
        for(dex=0;dex<num_points;dex++)
            if(former_clusters[dex]!=latter_clusters[dex])
                return -1;
        return 0;
}
/**
 * main function
 * divided to two brances for master & slave processors respectively
 * @param argc commandline argument count
 * @param argv array of commandline arguments
 * @return 0 if success
 */
int main(int argc, char* argv[])
{
        int rank;
        int size;
        int num_clusters;
        int num_points;
        int dex;
        int job_size;
        int job_done=0;

        Point* centroids;
        Point* points;
        Point* received_points;
        int  * slave_clusters;
        int  * former_clusters;
        int  * latter_clusters;

        MPI_Init(&argc, &argv);
```

```
      MPI_Status status;

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

      //creation of derived MPI structure
      MPI_Datatype MPI_POINT;
      MPI_Datatype type=MPI_DOUBLE;
      int blocklen=2;
      MPI_Aint disp=0;
      MPI_Type_create_struct(1,&blocklen,&disp,&type,&MPI_POINT);
      MPI_Type_commit(&MPI_POINT);

/******** MASTER PROCESSOR WORKS
HERE*************************************************/

      if(rank==MASTER)
      {
            //inputting from file
            FILE *input;
            input=fopen(argv[1],"r");
            readHeaders(input,&num_clusters,&num_points);
            points=(Point*)malloc(sizeof(Point)*num_points);
            readPoints(input,points,num_points);
            fclose(input);

            //other needed memory locations
            former_clusters=(int*)malloc(sizeof(int)*num_points);
            latter_clusters=(int*)malloc(sizeof(int)*num_points);
            job_size=num_points/(size-1);
            centroids=malloc(sizeof(Point)*num_clusters);

            //reseting and initializing to default behaviour
            initialize(centroids,num_clusters);
            resetData(former_clusters,num_points);
            resetData(latter_clusters,num_points);

            //Sending the essential data to slave processors
            for(dex=1;dex<size;dex++)
            {
                  printf("Sending to [%d]\n",dex);
                  MPI_Send(&job_size              ,1           , MPI_INT
,dex,0,MPI_COMM_WORLD);
                  MPI_Send(&num_clusters          ,1           , MPI_INT
,dex,0,MPI_COMM_WORLD);
                  MPI_Send(centroids              ,num_clusters,
MPI_POINT      ,dex,0,MPI_COMM_WORLD);
                  MPI_Send(points+(dex-1)*job_size,job_size    ,
MPI_POINT      ,dex,0,MPI_COMM_WORLD);
            }
            printf("Sent!\n");
```

```
            MPI_Barrier(MPI_COMM_WORLD);

            //Main job of master processor is done here
            while(1)
            {
                  MPI_Barrier(MPI_COMM_WORLD);

                  printf("Master Receiving\n");
                  for(dex=1;dex<size;dex++)
                        MPI_Recv(latter_clusters+(job_size*(dex-
1)),job_size,MPI_INT,dex,0,MPI_COMM_WORLD,&status);

                  printf("Master Received\n");


      calculateNewCentroids(points,latter_clusters,centroids,num_cluste
rs,num_points);
                  printf("New Centroids are done!\n");

      if(checkConvergence(latter_clusters,former_clusters,num_points)==
0)
                  {
                        printf("Converged!\n");
                        job_done=1;
                  }
                  else
                  {
                        printf("Not converged!\n");
                        for(dex=0;dex<num_points;dex++)
                              former_clusters[dex]=latter_clusters[dex];
                  }

                  //Informing slaves that no more job to be done
                  for(dex=1;dex<size;dex++)
                        MPI_Send(&job_done,1,
MPI_INT,dex,0,MPI_COMM_WORLD);

                  MPI_Barrier(MPI_COMM_WORLD);
                  if(job_done==1)
                        break;

                  //Sending the recently created centroids
                  for(dex=1;dex<size;dex++)
                        MPI_Send(centroids,num_clusters, MPI_POINT,dex,0,
MPI_COMM_WORLD);

                  MPI_Barrier(MPI_COMM_WORLD);
            }

            //Outputting to the output file
            FILE* output=fopen(argv[2],"w");
```

```
            fprintf(output,"%d\n",num_clusters);
            fprintf(output,"%d\n",num_points);
            for(dex=0;dex<num_clusters;dex++)

      fprintf(output,"%lf,%lf\n",centroids[dex]._x,centroids[dex]._y);
            for(dex=0;dex<num_points;dex++)

      fprintf(output,"%lf,%lf,%d\n",points[dex]._x,points[dex]._y,latte
r_clusters[dex]+1);
            fclose(output);
      }
/*************END OF MASTER PROCESSOR'S BRANCH -- SLAVE PROCESSORS'
JOB IS TO FOLLOW ************************/
      else
      {
            //Receiving the essential data
            printf("Receiving\n");
            MPI_Recv(&job_size    ,1           ,MPI_INT
,MASTER,0,MPI_COMM_WORLD,&status);
            MPI_Recv(&num_clusters,1           ,MPI_INT
,MASTER,0,MPI_COMM_WORLD,&status);
            centroids=malloc(sizeof(Point)*num_clusters);
            MPI_Recv(centroids
,num_clusters,MPI_POINT,MASTER,0,MPI_COMM_WORLD,&status);
            printf("part_size =%d\n",job_size);
            received_points=(Point*)malloc(sizeof(Point)*job_size);
            slave_clusters=(int*)malloc(sizeof(int)*job_size);
            MPI_Recv(received_points,job_size,MPI_POINT
,MASTER,0,MPI_COMM_WORLD,&status);
            printf("Received [%d]\n",rank);

            MPI_Barrier(MPI_COMM_WORLD);

            while(1)
            {
                  printf("Calculation of new clusters [%d]\n",rank);
                  for(dex=0;dex<job_size;dex++)
                  {

      slave_clusters[dex]=whoIsYourDaddy(received_points[dex],centroids
,num_clusters);
                  }

                  printf("sending to master [%d]\n",rank);
                  MPI_Send(slave_clusters,job_size, MPI_INT,MASTER, 0,
MPI_COMM_WORLD);
                  MPI_Barrier(MPI_COMM_WORLD);
                  MPI_Barrier(MPI_COMM_WORLD);
                  MPI_Recv(&job_done,1,
MPI_INT,MASTER,0,MPI_COMM_WORLD,&status);

                  if(job_done==1) //No more work to be done
```

```
                    break;

                //Receiving recently created centroids from master
                MPI_Recv(centroids,num_clusters,MPI_POINT,MASTER,0,
MPI_COMM_WORLD,&status);

                MPI_Barrier(MPI_COMM_WORLD);
            }
        }
        //End of all
        MPI_Finalize();
        return 0;
}
/* EOF */
```