ΠΜΣ “Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης”

Αλγοριθμικές Τεχνικές και Εφαρμογές

Καθηγητής: Χαράλαμπος Κωνσταντόπουλος

Κώστας Σπυρόπουλος – Πάνου Γιώργος

1η Εργασία

1ο Πρόβλημα

Το πρόβλημα της ταξινόμησης των στοιχείων μιας λίστας εμφανίζεται συχνά ως απαίτηση σε πολλές εφαρμογές. Για τον λόγο αυτό έχουν αναπτυχθεί πολλοί αλγόριθμοι που επιλύουν το πρόβλημα αυτό. Η πολυπλοκότητα των αλγορίθμων αυτών κινείται σε ένα ευρύ διάστημα, ανάλογα με το αν ικανοποιούνται συγκεκριμένες προϋποθέσεις για τα στοιχεία της λίστας κατά την έναρξη εκτέλεσης του αλγορίθμου. Τυπικά, η πολυπλοκότητα ενός καλού αλγόριθμου ταξινόμησης, χωρίς επιπλέον προϋποθέσεις για τα στοιχεία της λίστας, είναι O(n·logn), όπου n το πλήθος των στοιχείων της λίστας.

Η ταξινόμηση των στοιχείων σε ένα περιβάλλον κατανεμημένης μνήμης εισάγει επιπλέον δυσκολίες, καθώς απαιτείται ανταλλαγή δεδομένων κατά την εκτέλεση του εκάστοτε παράλληλου αλγόριθμου ταξινόμησης, αυξάνοντας το κόστος του αλγόριθμου. Κατά συνέπεια, πρέπει επιπλέον να μειωθεί και το πλήθος των ανταλλαγών δεδομένων.

Στα πλαίσια της εργασίας αυτής υλοποιήθηκε ο παράλληλος αλγόριθμος ταξινόμησης που περιγράφεται στη δημοσίευση:

«Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers. Minsoo Jeon and Dongseung Kim. In Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02), pp. 197-203, IEEE Computer Society, Washington, DC, USA, 2002» ,

χρησιμοποιώντας την βιβλιοθήκη MPI.

Για την υλοποίηση του αλγορίθμου οι διεργασίες περνούν από τρεις καταστάσεις ανάλογα με το στάδιο επεξεργασίας. Στο πιο χαμηλό επίπεδο οι διερργασίες ταξινομούν μόνο το κομμάτι που τους αποστέλεται, στο αμέσως επόμενο επίπεδο υπάρχει ένα ζεύγος διεργασιών που αποφασίζει ποιοι είναι οι splitters, χωρίζει τον αρχικό πίνακα σε υποπίνακες με τρόπο ώστε να είναι όλοι ξένοι μεταξύ τους ως προς την ταξινόμηση. Τέλος αποστέλει τα κομμάτια που αντιστοιχούν σε κάθε worker ταξινομεί το δικό του κομμάτι και κάνει το merging των υποπινάκων.

Ψευδοκώδικας Υλοποίησης Load-Balanced Parallel Merge Sort

1: Master splits initial array to n parts

2: All processes sort their own part

3: for (i = 2; i <= size; i = i \* 2)

4: The leading pair of each group creates splitters

5: Arch leader creates splitters as median

6: Other Leader performs a binary search for each splitter to define where it lies in his part

7: Leading pair sends splitted parts to all other processes

8: Arch Leader merges his part with other leader

Μέτρηση και αξιολόγηση μείωσης χρόνου ανάμεσα σε σειριακή και παράλληλη υλοποίηση:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 διεργασία | 2 διεργασίες | 4 διεργασίες | 8 διεργασίες |
| 10 million numbers | 3.1 sec | 2.8 sec | 1.9 sec | 1.6 sec |
| 50 million numbers | 16.7 sec | 10.6 sec | 9.7 sec | 8.9 sec |
| 100 million numbers | 34.1 sec | 23.4 sec | 21.1 sec | 19.1 sec |
| 120 million numbers | 45.5 sec | 31.0 sec | 27.7 sec | 24.7 sec |
| 300 million numbers | 128.3 sec | 82 sec | 71.2 sec | 66.8 sec |

2ο Πρόβλημα

Ένα σημαντικό πρόβλημα στην μάθηση χωρίς επίβλεψη είναι η εύρεση μιας δομής για την συσταδοποίηση ή ομαδοποίηση μιας συλλογής δεδομένων. Για τη λύση αυτού του προβλήματος, ένας από τους αλγόριθμους συσταδοποίησης που χρησιμοποιείται είναι ο DBCSAN.

Ο αλγόριθμος DBSCAN δέχεται σαν είσοδο, πέρα από το dataset που καλείται να ομαδοποιήσει, μόνο τον ελάχιστο αριθμό των δεδομένων που μπορούν να αποτελέσουν μια συστάδα και την ελάχιστη απόσταση που πρέπει να έχουν δύο στοιχεία ώστε να θεωρηθούν γείτονες (για την εύρεση της απόστασης χρησιμοποιείται η κλασσική Ευκλείδια απόσταση). Για την υλοποίηση του χρησιμοποιήθηκε γλώσσα προγραμματισμού C μέσω του IDE Visual Studio.

Υλοποιώντας τον αλγόριθμο DBSCAN σειριακά, παρατηρείται πως με την αύξηση των στοιχείων στο dataset που θέλουμε να ομαδοποιήσουμε, αυξάνεται εκθετικά και ο χρόνος που το πρόγραμμα εκτελείται. Για να μειωθεί ο χρόνος εκτέλεσης του κάναμε χρήση του προγραμματιστικού μοντέλου MPI, ώστε μεγάλο κομμάτι της υλοποίησής του να παραλληλοποιηθεί από πολλές διεργασίες. Να σημείωθεί πως ο αριθμός των διεργασιών είναι πάντα δύναμη του 2 συν 1 (1 p, 3 p, 5 p, 9 p κτλ).

Συγκεκριμένα, χωρίζουμε το dataset σε κομμάτια και η κάθε διεργασία υλοποιεί τον αλγόριθμο DBSCAN, ξεχωριστά και ανεξάρτητα από τις άλλες, στο δικό της dataset. Στη συνέχεια, η κάθε διεργασία υλοποιεί τα clusters για το dataset της και δημιουργούνται ζεύγη διεργασιών όπου η μια στέλνει τα αποτελέσματά της στη άλλη. Η διεργασία που δέχεται τα δεδομένα κάνει σύγκριση των δικών της clusters με αύτα που δέχτηκε από την προηγούμενη διεργασία και δημιουργεί τα καινούργια merged clusters. Η διαδικασία επαναλαμβάνεται μέχρι η τελευταία διεργασία να περιέχει τα clusters όλων των διεργασιών.

Ψευδοκώδικας Υλοποίησης DBSCAN:

1: all Worker nodes splits data set *D* into *D/*(*P −* 1) subsets

2: for all Worker nodes *vi, vi ∈ {v*1*, v*2*, . . . , vP−*1*}* do in parallel

3: Read data set, epsilon, minpts

4: Apply DBSCAN algorithm

5: define clusters

6: end for

7: rank *←* myid *{*node identification number*}*

*8: n ← 2, m ← 0*

9: while (m <= log(P - 1)) do

10: for all *vi, vi ∈ {v*1*, v*2*, . . . , vP−*1*}* do in parallel

11: if (*rank mod* 2 = n / 2)) then

12: Send clusters and points to *rank* + (n/2) *{*node is sender*}*

13: else if (rank mod n = 0)

14: Receive *clusters and points*, from *rank −* (n/2) *{*node is receiver*}*

15: Merge and reform clusters and points

16: end if

17: end for

18: *n ←* n\*2, m *← m + 1*

19: end while

20: Last worker node prints result clusters

Η διαδικασία της ένωσης των clusters μεταξύ δύο διεργασιών γίνεται ως εξής:

* Σύγκριση όλων των clusters της μιας διεργασίας με τα clusters της άλλης.
  + Εάν ένα cluster βρίσκεται εντός του άλλου, τα clusters συγχωνεύονται.
  + Εάν εφάπτονται σε δυο σημεία, τότε αν μεταξύ του κοινού χώρου βρίσκονται points τότε συγχωνεύονται
  + Εάν εφάπτονται σε ένα σημείο, αν σε αυτό το σημείο βρίσκεται point τότε συγχωνεύονται.
  + Εάν δεν έχουν καμία επαφή μεταξύ τους, μενουν ως έχει.
* Δημιουργία καινούργιου merged\_cluster struct όπου περιέχονται τα συγχωνευμένα clusters και τα clusters που δεν συγχωνεύτηκαν με κάποιο άλλο.
* Επανέλεγχος των points που είχαν οριστεί ως noise αν πλέον βρίσκονται εντός cluster.

Μέτρηση και αξιολόγηση μείωσης χρόνου ανάμεσα σε σειριακή και παράλληλη υλοποίηση:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 διεργασία | 3 διεργασίες | 5 διεργασίες | 9 διεργασίες |
| 10000 points | 18.856 sec | 4.939 sec | 2.022 sec | 0.807 sec |
| 5000 points | 4.745 sec | 1.024 sec | 0.765 sec | 0.39 sec |
| 1000 points | 0.311 sec | 0.134 sec | 0.032 sec | 0.029 sec |
| 100 points | 0.003 sec | 0.003 sec | 0.006 sec | 0.01 sec |

Παράρτημα - Κώδικας

Κώδικας MPI Load-Balanced Parallel Merge Sor tσε C:

//George Panou - Kostas Spyropoulos - Unipi 2019

//Defines and includes

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <limits.h>

#include "mpi.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "time.h"

#include "HelperFunctions.h"

#include <string.h>

typedef struct point\_s point\_t;

struct point\_s {

double x, y;

int cluster\_id;

};

//Main

int main(int argc, char \*argv[]) {

double epsilon;

unsigned int minpts;

unsigned int num\_points, num\_points\_per\_proc, i = 0;

int rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

clock\_t t;

t = clock();

clock\_t total\_time;

clock\_t total\_parallel;

clock\_t init\_parallel;

float timeSpentTotal;

float timeSpentQS;

float timeParallelMergeSort;

int \* finalArray;

int finalArrayLenght;

int temp = 0;

int\* tempAr=(int \*)calloc(1, sizeof(int));

tempAr[0]=0;

int\* tmpmtr;

tmpmtr = (int\*)calloc(1,sizeof(int));

tmpmtr[0]=0;

//printf("Begin Time Calculation...\n");

//MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == 0){

total\_time = clock();

//Create a random dataset and save it to file

int length = 100000000;

int range = 200000;

int \* A = createDataSet(length, range);

//saveArrayToFile(A, length, "tmp.txt", "unsorted");

myArray\_t arr = loadArrayFromfile("tmp.txt");

t = clock();

int \* C = sort(arr.Array, arr.length,0);

t = (clock() - t);

timeSpentQS = ((float)t) / CLOCKS\_PER\_SEC;

printf("Process: %d : It took me %d clicks (%.3f seconds) to sort the array sequentially\n", rank, t, timeSpentQS);

//saveArrayToFile(C, arr.length, "sequencialSortOutput.txt", "sorted test");

MPI\_Barrier(MPI\_COMM\_WORLD);

init\_parallel=clock();

int partLength = arr.length / size;

int remainderCells = arr.length % size;

int totalLength = partLength + remainderCells;

int\* part = (int \*)calloc(totalLength, sizeof(int));

int\* myPart = (int \*)calloc(totalLength, sizeof(int));

int i;

int processRank = 1;

int j = 0;

for (i = 0; i < partLength; i++) {

myPart[i] = arr.Array[i];

}

for (i = partLength; i < arr.length; i++) {

part[j] = arr.Array[i];

j++;

//printf("cell:%d\n", arr.Array[i]);

if (i == arr.length - 1) {//final cell

MPI\_Send(&totalLength, 1, MPI\_INT, processRank, 0, MPI\_COMM\_WORLD);

MPI\_Send(part, totalLength, MPI\_INT, processRank, 0, MPI\_COMM\_WORLD);

}

if ((i+1) % partLength == 0 && processRank != size-1 ) {

MPI\_Send(&partLength, 1, MPI\_INT, processRank, 0, MPI\_COMM\_WORLD);

MPI\_Send(part, partLength, MPI\_INT, processRank, 0, MPI\_COMM\_WORLD);

j = 0;

processRank++;

}

}

//ARCH leader

int fallenLeader = 0;

int splitter = 0;

int overlapHead = 0;

int overlapTail = 0;

int overlapSplit = 0;

int leaderTail = 0;

int overlapHeadPos = 0;

int overlapTailPos = 0;

int \* mergedArray;

int mergedArrayLenght;

printf("\n\nProcess: %d >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n", rank);

MPI\_Request request;

int count=0;

int splittersCount=2;

sort(myPart, partLength, 1);

for (i = 2; i <= size; i = i \* 2) {//Merge Phazes loop

count++;

splittersCount=(int)pow(2,count)-1;

printf("Phaze: %d ---------- count %d ---------------- ----------------------- --------------------- ---------------- ----------------- -------\n", i,count);

if (rank % i == 0) {//

int otherLeader;

int fallen = rank + i / 2;

//group leader

int groupLeader = rank - i / 2;

groupLeader=0;

//find overlap:

//send last item of mypart to fallen leader

overlapTail = myPart[partLength - 1];

MPI\_Isend(&overlapTail, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

//recv first item from 2nd part

MPI\_Recv(&overlapHead, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

overlapHeadPos = binarySearch(myPart, overlapHead, 0, partLength - 1, 1, 0);

int overlapStatus = 1;

if(overlapHeadPos!=-2){

if (overlapHeadPos==-1)

overlapHeadPos=0;

int overlapStatus = 1;

MPI\_Isend(&overlapStatus, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

//first define the splitted arrays and send them, then receive from fallen part to locally sort then recv from workers and merge

int spart=1;

int fallenPartSize=0;

int overlapStart=0;

MPI\_Recv(&fallenPartSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

mergedArrayLenght=partLength+fallenPartSize;

int \* mergedPart = (int \*)calloc(mergedArrayLenght,sizeof(int));

printf("Fallenpart size: %d\n",fallenPartSize);

overlapStart=overlapHeadPos;

int splittedSize ;

int myOverlapSplitter;

int splitters[splittersCount+1];

int arraySize[splittersCount+1];

int\* array[splittersCount+1];

int receivers[splittersCount+1];

for(spart=0; spart<splittersCount; spart++){

splitter = overlapHeadPos + (partLength - 1 - overlapHeadPos) / (splittersCount+1) \* (spart+1);

int temp;

if (splitter < partLength){

temp=splitter+1;

while(myPart[splitter]==myPart[temp] && temp<partLength){//go to last occurence of equal values

splitter++;

temp++;

}

}

//send splitters to fallen leader

int splitersValue=myPart[splitter];

splitters[spart]=splitersValue;

MPI\_Send(&splitters[spart], 1, MPI\_INT, fallen, 3, MPI\_COMM\_WORLD);

// MPI\_Recv(&tmpmtr, 1, MPI\_INT, fallen, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);//block to avoid invalid buffers

splittedSize = splitter - overlapStart;//-1 to keep splitter

if (splittedSize == 0){

if (spart+rank == rank){

myOverlapSplitter=splitter;

}

MPI\_Isend(&temp, 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

arraySize[spart]=temp;

MPI\_Isend(tempAr, temp, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

array[spart]=tempAr;

}else{

if(spart+rank>rank){

//send worker's overlap part to be sorted

arraySize[spart]=splittedSize;

MPI\_Isend(&arraySize[spart], 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(splittedSize, sizeof(int));

memcpy(array[spart], myPart+overlapStart+1, (splittedSize)\* sizeof(int));//+1 to keep splitter

MPI\_Isend(array[spart], splittedSize, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

}else if (spart+rank == rank){

myOverlapSplitter=splitter;

}

}

overlapStart=splitter;

}

printf("splitter:%d OverlapStart %d\n",splitter,splittedSize);

splittedSize = partLength - 1 - overlapStart;//-1 to keep splitter

//send worker's overlap part to be sorted

arraySize[spart]=splittedSize;

MPI\_Isend(&arraySize[spart], 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(splittedSize, sizeof(int));

memcpy(array[spart], myPart+overlapStart+1, (splittedSize)\* sizeof(int));//+1 to keep splitter

MPI\_Isend(array[spart], splittedSize, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

//receive fallen leaders' sent part to be merged and sorted locally

int workerOverlapCount = 0;

MPI\_Recv(&workerOverlapCount, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int overlapSize = workerOverlapCount + myOverlapSplitter + 1 - overlapHeadPos;//+1 to keep splitter

int \* overlap = (int \*)calloc(overlapSize, sizeof(int));

//if(workerOverlapCount>0)

MPI\_Recv(overlap, workerOverlapCount, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(overlap + workerOverlapCount, myPart + overlapHeadPos, ((myOverlapSplitter + 1 - overlapHeadPos) )\* sizeof(int));//+1 to keep splitter

sort(overlap, overlapSize, 1);

memcpy(mergedPart, myPart, (overlapHeadPos) \* sizeof(int));

memcpy(mergedPart + overlapHeadPos, overlap, (overlapSize) \* sizeof(int));

int workingPosition=overlapHeadPos + overlapSize;

for(spart=1; spart<=splittersCount; spart++){

//receive workers sorted overlap parrt

int sortedWorkerOverlapCount = 0;

MPI\_Recv(&sortedWorkerOverlapCount, 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* sortedWorkerOverlap = (int \*)calloc(sortedWorkerOverlapCount, sizeof(int));

MPI\_Recv(sortedWorkerOverlap, sortedWorkerOverlapCount, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(mergedPart+workingPosition, sortedWorkerOverlap, (sortedWorkerOverlapCount) \* sizeof(int));

workingPosition+=sortedWorkerOverlapCount;

}

//receive tail (trailing sorted elements)

int tailSize = 0;

MPI\_Recv(&tailSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* tail = (int \*)calloc(tailSize, sizeof(int));

//if(tailSize>0)

MPI\_Recv(tail, tailSize, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(mergedPart+workingPosition, tail, (tailSize) \* sizeof(int));

mergedArray=mergedPart;

free(myPart);

myPart=mergedPart;

partLength=mergedArrayLenght;

}else{//no overlap just recv other part

printf("no overlap just recv other part\n");

int tailSize;

int spart2=1;

int overlapStatus = -1;

MPI\_Isend(&overlapStatus, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

for(spart2=1; spart2<splittersCount; spart2++){

if(spart2!=fallen){

int temp=-1;

//inform workers about no overlap

MPI\_Isend(&temp, 1, MPI\_INT, spart2+rank, 0, MPI\_COMM\_WORLD,&request);

}

}

MPI\_Recv(&tailSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* tail = (int \*)calloc(tailSize, sizeof(int));

//if(tailSize>0)

MPI\_Recv(tail, tailSize, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

//concatenate all parts

int \* mergedPart = (int \*)calloc(partLength+tailSize, sizeof(int));

memcpy(mergedPart, myPart, (partLength) \* sizeof(int));

memcpy(mergedPart + partLength, tail, (tailSize) \* sizeof(int));

printf("lead4\n");

free(myPart);

myPart=mergedPart;

partLength=partLength+tailSize;

}

}

MPI\_Barrier(MPI\_COMM\_WORLD);

// saveArrayToFileWithCheck(myPart, partLength,"merged.txt", "merged array");

}

finalArray=myPart;

finalArrayLenght=partLength;

}

printf("----------------------------------------------------------------------------\n");

////printArray(myPart, totalLength, "worker");

if (rank != 0)

{

//int length = 10000;

//int range = 100000;

//int \* A = createDataSet(length, range);

int partLength = 10;

MPI\_Barrier(MPI\_COMM\_WORLD);

//if (rank == 3) {

MPI\_Recv(&partLength, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* myPart = (int \*)calloc(partLength, sizeof(int));

MPI\_Recv(myPart, partLength, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

//}

////printArray(myPart, partLength, "7");

printf("\n\nProcess: %d >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n", rank);

printf("\n===part length:%d\n\n", partLength);

t = clock();

////printArray(myPart, partLength, "arr");

int fallenLeader = 0;

int splitter = 0;

int overlapHead = 0;

int overlapTail = 0;

int overlapSplit = 0;

int leaderTail = 0;

int overlapHeadPos = 0;

int overlapTailPos = 0;

int leaderOverlapCount = 0;

int \* mergedArray;

int mergedArrayLenght;

int i;

int count=0;

MPI\_Request request;

int splittersCount=2;

MPI\_Status status;

int splittedArraySize1;

int splittedArraySize2;

for (i = 2; i <= size; i = i \* 2) {//Merge Phazes loop

count++;

splittersCount=(int)pow(2,count)-1;

sort(myPart, partLength, 1);

printf("I am %d and rank mod i = %d and \n",rank,rank % i);

printf("Phaze: %d ---------- splittersCount %d\n", i,splittersCount);

////printArray(myPart, partLength, "mypart");

if (rank % i == 0) {//

int otherLeader;

int fallen = rank + i / 2;

//group leader

int groupLeader = rank - i / 2;

groupLeader=0;

//find overlap:

//send last item of mypart to fallen leader

overlapTail = myPart[partLength - 1];

MPI\_Isend(&overlapTail, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

//recv first item from 2nd part

MPI\_Recv(&overlapHead, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

overlapHeadPos = binarySearch(myPart, overlapHead, 0, partLength - 1, 1, 0);

int overlapStatus = 1;

if(overlapHeadPos!=-2){

if (overlapHeadPos==-1)

overlapHeadPos=0;

int overlapStatus = 1;

MPI\_Isend(&overlapStatus, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

//first define the splitted arrays and send them, then receive from fallen part to locally sort then recv from workers and merge

int spart=1;

int fallenPartSize=0;

int overlapStart=0;

MPI\_Recv(&fallenPartSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

mergedArrayLenght=partLength+fallenPartSize;

int \* mergedPart = (int \*)calloc(mergedArrayLenght,sizeof(int));

printf("Fallenpart size: %d\n",fallenPartSize);

overlapStart=overlapHeadPos;

int splittedSize ;

int myOverlapSplitter;

int splitters[splittersCount+1];

int arraySize[splittersCount+1];

int\* array[splittersCount+1];

int receivers[splittersCount+1];

for(spart=0; spart<splittersCount; spart++){

splitter = overlapHeadPos + (partLength - 1 - overlapHeadPos) / (splittersCount+1) \* (spart+1);

int temp;

if (splitter < partLength){

temp=splitter+1;

while(myPart[splitter]==myPart[temp] && temp<partLength){//go to last occurence of equal values

splitter++;

temp++;

}

}

//send splitters to fallen leader

int splitersValue=myPart[splitter];

splitters[spart]=splitersValue;

MPI\_Send(&splitters[spart], 1, MPI\_INT, fallen, 3, MPI\_COMM\_WORLD);

// MPI\_Recv(&tmpmtr, 1, MPI\_INT, fallen, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);//block to avoid invalid buffers

splittedSize = splitter - overlapStart;//-1 to keep splitter

if (splittedSize == 0){

if (spart+rank == rank){

myOverlapSplitter=splitter;

}

MPI\_Isend(&temp, 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

arraySize[spart]=temp;

MPI\_Isend(tempAr, temp, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

array[spart]=tempAr;

}else{

if(spart+rank>rank){

//send worker's overlap part to be sorted

arraySize[spart]=splittedSize;

MPI\_Isend(&arraySize[spart], 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(splittedSize, sizeof(int));

memcpy(array[spart], myPart+overlapStart+1, (splittedSize)\* sizeof(int));//+1 to keep splitter

MPI\_Isend(array[spart], splittedSize, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

}else if (spart+rank == rank){

myOverlapSplitter=splitter;

}

}

overlapStart=splitter;

}

printf("splitter:%d OverlapStart %d\n",splitter,splittedSize);

splittedSize = partLength - 1 - overlapStart;//-1 to keep splitter

//send worker's overlap part to be sorted

arraySize[spart]=splittedSize;

MPI\_Isend(&arraySize[spart], 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(splittedSize, sizeof(int));

memcpy(array[spart], myPart+overlapStart+1, (splittedSize)\* sizeof(int));//+1 to keep splitter

MPI\_Isend(array[spart], splittedSize, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD,&request);//+1 to keep splitter

//receive fallen leaders' sent part to be merged and sorted locally

int workerOverlapCount = 0;

MPI\_Recv(&workerOverlapCount, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int overlapSize = workerOverlapCount + myOverlapSplitter + 1 - overlapHeadPos;//+1 to keep splitter

int \* overlap = (int \*)calloc(overlapSize, sizeof(int));

//if(workerOverlapCount>0)

MPI\_Recv(overlap, workerOverlapCount, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(overlap + workerOverlapCount, myPart + overlapHeadPos, ((myOverlapSplitter + 1 - overlapHeadPos) )\* sizeof(int));//+1 to keep splitter

sort(overlap, overlapSize, 1);

memcpy(mergedPart, myPart, (overlapHeadPos) \* sizeof(int));

memcpy(mergedPart + overlapHeadPos, overlap, (overlapSize) \* sizeof(int));

int workingPosition=overlapHeadPos + overlapSize;

for(spart=1; spart<=splittersCount; spart++){

//receive workers sorted overlap parrt

int sortedWorkerOverlapCount = 0;

MPI\_Recv(&sortedWorkerOverlapCount, 1, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* sortedWorkerOverlap = (int \*)calloc(sortedWorkerOverlapCount, sizeof(int));

MPI\_Recv(sortedWorkerOverlap, sortedWorkerOverlapCount, MPI\_INT, spart+rank, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(mergedPart+workingPosition, sortedWorkerOverlap, (sortedWorkerOverlapCount) \* sizeof(int));

workingPosition+=sortedWorkerOverlapCount;

}

//receive tail (trailing sorted elements)

int tailSize = 0;

MPI\_Recv(&tailSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* tail = (int \*)calloc(tailSize, sizeof(int));

//if(tailSize>0)

MPI\_Recv(tail, tailSize, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(mergedPart+workingPosition, tail, (tailSize) \* sizeof(int));

mergedArray=mergedPart;

//printArr(myPart);

myPart=mergedPart;

partLength=mergedArrayLenght;

}else{//no overlap just recv other part

printf("no overlap just recv other part\n");

int tailSize;

int spart2=1;

int overlapStatus = -1;

MPI\_Isend(&overlapStatus, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD,&request);

for(spart2=1; spart2<splittersCount; spart2++){

if(spart2!=fallen){

int temp=-1;

//inform workers about no overlap

MPI\_Isend(&temp, 1, MPI\_INT, spart2+rank, 0, MPI\_COMM\_WORLD,&request);

}

}

MPI\_Recv(&tailSize, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int\* tail = (int \*)calloc(tailSize, sizeof(int));

//if(tailSize>0)

MPI\_Recv(tail, tailSize, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

//concatenate all parts

int \* mergedPart = (int \*)calloc(partLength+tailSize, sizeof(int));

memcpy(mergedPart, myPart, (partLength) \* sizeof(int));

memcpy(mergedPart + partLength, tail, (tailSize) \* sizeof(int));

printf("lead4\n");

//printArr(myPart);

myPart=mergedPart;

partLength=partLength+tailSize;

}

}else {//fallen leader

int groupLeader;

int fallen;

if(rank>i){

groupLeader = rank - rank % i;

fallen=groupLeader+(i/2);

}else{

groupLeader=0;

fallen=groupLeader+(i/2);

}

if (fallenLeader == 0) {//fallen leader only (at 1st iter all workers are fallen leaders)

int groupLeader = rank - i / 2;

//find overlap:

//send first item of mypart

overlapHead = myPart[0];

MPI\_Isend(&overlapHead, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

//recv last item from 1st part

MPI\_Recv(&leaderTail, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int overlapStatus = 0;

MPI\_Recv(&overlapStatus, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

overlapTailPos = binarySearchReturnLess(myPart, leaderTail, 0, partLength - 1, 1);

if(overlapTailPos!=-1 && overlapStatus>0){

if (overlapTailPos == -2)

overlapTailPos = partLength - 1;

//recv splitter

int spart;

int overlapSplitSize;

int mySplitter = overlapTailPos;

int overlapStart = 0;

int tempSplitter = 0;

int overlapSplit =0;

int prevSplitter = 0;

MPI\_Isend(&partLength, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

overlapSplit = -1;

int flag=0;

int splitters[splittersCount+1];

int arraySize[splittersCount+1];

int\* array[splittersCount+1];

int receivers[splittersCount+1];

int splitter;

for(spart=0; spart<splittersCount; spart++){

overlapStart=overlapSplit+1;

MPI\_Recv(&splitter, 1, MPI\_INT, groupLeader, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

//MPI\_Send(&tmpmtr, 1, MPI\_INT, groupLeader, 2, MPI\_COMM\_WORLD);//unblock leader

//binary search to define where the splitter lies

overlapSplit = binarySearchReturnLess(myPart, splitter, 0, overlapTailPos, 1);

if(overlapSplit==0){

flag=1;

}

if(overlapSplit==-1){

overlapSplit=0;

}else if(overlapSplit==-2){

overlapSplit=overlapTailPos;

}

overlapSplitSize = overlapSplit-overlapStart +1;

if(overlapSplitSize==0 || overlapSplit<0 ){

if(overlapSplit==overlapStart){

flag=1;

}

MPI\_Isend(&temp, 1, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

MPI\_Isend(tempAr, temp, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

if(rank!=spart+groupLeader){//dont send to self

}else{

mySplitter=overlapSplit;

prevSplitter=tempSplitter;

}

tempSplitter=overlapSplit;

}else{

if (overlapSplit == 0) {

// overlapSplitSize = 0;

}

if(rank!=spart+groupLeader){//dont send to self

//send overlap part to be sorted to wrker and leader

arraySize[spart]=overlapSplitSize;

MPI\_Isend(&arraySize[spart], 1, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(overlapSplitSize, sizeof(int));

memcpy(array[spart], myPart+overlapStart, (overlapSplitSize)\* sizeof(int));

MPI\_Isend(array[spart], overlapSplitSize, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

}else{

mySplitter=overlapSplit;

prevSplitter=tempSplitter;

}

tempSplitter=overlapSplit;

}

}

if(splittersCount==1){

prevSplitter = binarySearchReturnLess(myPart, splitter, overlapStart, overlapTailPos, 1);

if(prevSplitter==-1){

prevSplitter=0;

}else if(prevSplitter==-2){

prevSplitter=overlapTailPos;

}

}

if(rank!=spart+groupLeader){//dont send to self

//send last overlap part

overlapSplitSize = overlapTailPos-overlapSplit;

arraySize[spart]=overlapSplitSize;

MPI\_Isend(&overlapSplitSize, 1, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

array[spart]=(int \*)calloc(overlapSplitSize, sizeof(int));

memcpy(array[spart], myPart+overlapSplit+1, (overlapSplitSize)\* sizeof(int));

MPI\_Isend( array[spart],overlapSplitSize, MPI\_INT, spart+groupLeader, 0, MPI\_COMM\_WORLD,&request);

}

//receive other subpart from leader

MPI\_Recv(&leaderOverlapCount, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int overlapSize = leaderOverlapCount + mySplitter - prevSplitter;

int\* overlap = (int \*)calloc(overlapSize, sizeof(int));

//if(leaderOverlapCount>0)

MPI\_Recv(overlap, leaderOverlapCount, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

memcpy(overlap + leaderOverlapCount, myPart + prevSplitter+1, (mySplitter - prevSplitter) \* sizeof(int));

sort(overlap, overlapSize, 1);

//send sorted overlap part

MPI\_Isend(&overlapSize, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

MPI\_Isend(overlap, overlapSize, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

int tailSize = partLength -1 - overlapTailPos;

MPI\_Isend(&tailSize, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

MPI\_Isend(myPart + overlapTailPos+1, tailSize, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

//free(overlap);

}else{//no overlap just send whole part

MPI\_Isend(&partLength, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

//if(partLength>0)

MPI\_Isend(myPart, partLength, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

}

fallenLeader--;

} else {//worker

int groupLeader;

int fallen;

if(rank>i){

groupLeader = rank - rank % i;

fallen=groupLeader+(i/2);

}else{

groupLeader=0;

fallen=groupLeader+(i/2);

}

// receive message from leader

MPI\_Recv(&splittedArraySize1, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&splittedArraySize2, 1, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, &status);

if(splittedArraySize1!=-1){//else do nothing as there is no overlap

int splittedArraySize=splittedArraySize1+splittedArraySize2;

int\* splittedArray = (int \*)calloc(splittedArraySize, sizeof(int));

MPI\_Recv(splittedArray, splittedArraySize1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(splittedArray+splittedArraySize1, splittedArraySize2, MPI\_INT, fallen, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

sort(splittedArray,splittedArraySize,1);

// send reply

MPI\_Isend(&splittedArraySize, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

MPI\_Isend(splittedArray, splittedArraySize, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

}else{

MPI\_Isend(&temp, 1, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

MPI\_Isend(tempAr, temp, MPI\_INT, groupLeader, 0, MPI\_COMM\_WORLD,&request);

}

}

}

MPI\_Barrier(MPI\_COMM\_WORLD);

//free(myPart);

}

}

printf("----------------------------------------------------------------------------\n");

total\_parallel = clock();

MPI\_Barrier(MPI\_COMM\_WORLD);

//time calculations

if (rank == 0) {

total\_parallel = (clock() - init\_parallel);

timeParallelMergeSort = ((float)total\_parallel) / CLOCKS\_PER\_SEC;

//printArray(finalArray,finalArrayLenght,"finalarr");

//saveArrayToFileWithCheck(finalArray, finalArrayLenght,"merged.txt", "merged array");

printf("It took %d clicks (%.3f seconds) to sort the array sequentially\n Using %u processors and parallel mergesort: %d clicks - (%.3f seconds).\n", t, timeSpentQS, size, total\_parallel, timeParallelMergeSort);

total\_time = (clock() - total\_time);

timeSpentTotal = ((float)total\_time) / CLOCKS\_PER\_SEC;

}

MPI\_Finalize();

return 0;

}

Κώδικας MPI DBSCAN σε C:

/\*2-axis DBSCAN Code with MPI\*/

//Defines and includes

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <limits.h>

#include "mpi.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "time.h"

#include <stdbool.h>

#define UNCLASSIFIED -1

#define NOISE -2

#define CORE\_POINT 1

#define NOT\_CORE\_POINT 0

#define SUCCESS 0

#define FAILURE -3

//Structs of points, nodes and epsilon\_neighbours

typedef struct point\_s point\_t;

struct point\_s {

double x, y;

int cluster\_id;

};

typedef struct cluster cluster\_t;

struct cluster {

double x, y;

int gravity;

int cluster\_id;

double radius;

int points\_count;

bool exists;

};

typedef struct node\_s node\_t;

struct node\_s {

unsigned int index;

node\_t \*next;

};

typedef struct epsilon\_neighbours\_s epsilon\_neighbours\_t;

struct epsilon\_neighbours\_s {

unsigned int num\_members;

node\_t \*head, \*tail;

};

//Functions Declaration

node\_t \*create\_node(unsigned int index);

int append\_at\_end(unsigned int index, epsilon\_neighbours\_t \*en);

epsilon\_neighbours\_t \*get\_epsilon\_neighbours(unsigned int index, point\_t \*points, unsigned int num\_points, double epsilon, double(\*dist)(point\_t \*a, point\_t \*b));

void print\_epsilon\_neighbours(point\_t \*points, epsilon\_neighbours\_t \*en);

void destroy\_epsilon\_neighbours(epsilon\_neighbours\_t \*en);

unsigned int dbscan(point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b));

int expand(unsigned int index, unsigned int cluster\_id, point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b));

int spread(unsigned int index, epsilon\_neighbours\_t \*seeds, unsigned int cluster\_id, point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b));

double euclidean\_dist(point\_t \*a, point\_t \*b);

void print\_points(point\_t \*points, unsigned int num\_points);

cluster\_t \*clusters\_init(point\_t \*points, unsigned int num\_points\_per\_proc, int clusters\_count, int rank);

cluster\_t \*compare\_clusters(cluster\_t \*clusters, cluster\_t \*send\_cluster, point\_t \*points, point\_t \*recv\_points, int cluster\_count, int cluster\_count\_send, int minpts, double eps);

cluster\_t \*find\_noise\_relation(cluster\_t \*clusters, point\_t \*points, int num\_points\_per\_proc, int c);

//Node Creation

node\_t \*create\_node(unsigned int index)

{

node\_t \*n = (node\_t \*)calloc(1, sizeof(node\_t));

if (n == NULL)

perror("Failed to allocate node.");

else {

n->index = index;

n->next = NULL;

}

return n;

}

// Append At End

int append\_at\_end(unsigned int index, epsilon\_neighbours\_t \*en)

{

node\_t \*n = create\_node(index);

if (n == NULL) {

free(en);

return FAILURE;

}

if (en->head == NULL) {

en->head = n;

en->tail = n;

}

else {

en->tail->next = n;

en->tail = n;

}

++(en->num\_members);

return SUCCESS;

}

//Get Neighbours

epsilon\_neighbours\_t \*get\_epsilon\_neighbours(unsigned int index, point\_t \*points, unsigned int num\_points, double epsilon, double(\*dist)(point\_t \*a, point\_t \*b))

{

epsilon\_neighbours\_t \*en = (epsilon\_neighbours\_t \*)calloc(1, sizeof(epsilon\_neighbours\_t));

if (en == NULL) {

perror("Failed to allocate epsilon neighbours.");

return en;

}

for (unsigned int i = 0; i < num\_points; ++i) {

if (i == index)

continue;

if (dist(&points[index], &points[i]) > epsilon)

continue;

else {

if (append\_at\_end(i, en) == FAILURE) {

destroy\_epsilon\_neighbours(en);

en = NULL;

break;

}

}

}

return en;

}

// Print Neighbours

void print\_epsilon\_neighbours(point\_t \*points, epsilon\_neighbours\_t \*en)

{

if (en) {

node\_t \*h = en->head;

while (h) {

printf("(%lfm, %lf)\n", points[h->index].x, points[h->index].y);

h = h->next;

}

}

}

// Destroy Neighbours

void destroy\_epsilon\_neighbours(epsilon\_neighbours\_t \*en)

{

if (en) {

node\_t \*t, \*h = en->head;

while (h) {

t = h->next;

free(h);

h = t;

}

free(en);

}

}

// Main DBSCAN Algorithm

unsigned int dbscan(point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b))

{

unsigned int i, cluster\_id = 0;

for (i = 0; i < num\_points; ++i) {

if (points[i].cluster\_id == UNCLASSIFIED) {

if (expand(i, cluster\_id, points, num\_points, epsilon, minpts, dist) == CORE\_POINT)

++cluster\_id;

}

}

return cluster\_id;

}

int expand(unsigned int index, unsigned int cluster\_id, point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b))

{

int return\_value = NOT\_CORE\_POINT;

epsilon\_neighbours\_t \*seeds = get\_epsilon\_neighbours(index, points, num\_points, epsilon, dist);

if (seeds == NULL)

return FAILURE;

if (seeds->num\_members < minpts)

points[index].cluster\_id = NOISE;

else {

points[index].cluster\_id = cluster\_id;

node\_t \*h = seeds->head;

while (h) {

points[h->index].cluster\_id = cluster\_id;

h = h->next;

}

h = seeds->head;

while (h) {

spread(h->index, seeds, cluster\_id, points, num\_points, epsilon, minpts, dist);

h = h->next;

}

return\_value = CORE\_POINT;

}

destroy\_epsilon\_neighbours(seeds);

return return\_value;

}

int spread(unsigned int index, epsilon\_neighbours\_t \*seeds, unsigned int cluster\_id, point\_t \*points, unsigned int num\_points, double epsilon, unsigned int minpts, double(\*dist)(point\_t \*a, point\_t \*b))

{

epsilon\_neighbours\_t \*spread = get\_epsilon\_neighbours(index, points, num\_points, epsilon, dist);

if (spread == NULL)

return FAILURE;

if (spread->num\_members >= minpts) {

node\_t \*n = spread->head;

point\_t \*d;

while (n) {

d = &points[n->index];

if (d->cluster\_id == NOISE || d->cluster\_id == UNCLASSIFIED) {

if (d->cluster\_id == UNCLASSIFIED) {

if (append\_at\_end(n->index, seeds) == FAILURE) {

destroy\_epsilon\_neighbours(spread);

return FAILURE;

}

}

d->cluster\_id = cluster\_id;

}

n = n->next;

}

}

destroy\_epsilon\_neighbours(spread);

return SUCCESS;

}

// Euclidean Distance

double euclidean\_dist(point\_t \*a, point\_t \*b)

{

return sqrt(pow(a->x - b->x, 2) + pow(a->y - b->y, 2));

}

//Print Results

void print\_points(point\_t \*points, unsigned int num\_points)

{

unsigned int i = 0;

printf("Number of points: %u\n"

" x y cluster\_id\n"

"-----------------------\n"

, num\_points);

while (i < num\_points) {

printf("%5.2lf %5.2lf: %d\n", points[i].x, points[i].y, points[i].cluster\_id);

++i;

}

printf("\n");

}

//Calculate CenterOfMass per cluster, Radius and Count of points per cluster

cluster\_t \*clusters\_init(point\_t \*points, unsigned int num\_points\_per\_proc, int clusters\_count, int rank)

{

int i, k, j = 0;

point\_t clusterCOM, currentPoint, current\_c\_point;

double \*maxDistanceFromCOM = (double \*)calloc(clusters\_count, sizeof(double));

cluster\_t \*clusters = (cluster\_t \*)calloc(clusters\_count, sizeof(cluster\_t));

int \*clusterPointsCount = (int \*)calloc(clusters\_count, sizeof(int));

point\_t \*borderPoint = (point\_t \*)calloc(clusters\_count, sizeof(point\_t));

for (i = 0; i < clusters\_count; i++)

{

clusters[i].x = 0;

clusters[i].y = 0;

clusters[i].gravity = 1;

clusters[i].exists = true;

maxDistanceFromCOM[i] = 0;

clusterPointsCount[i] = 0;

}

for (i = 0; i < num\_points\_per\_proc; i++)

{

int currCluster = points[i].cluster\_id;

if (currCluster >= 0)

{

if (clusters[currCluster].x > points[i].x)

clusters[currCluster].x = clusters[currCluster].x - (clusters[currCluster].x - points[i].x) / (clusters[currCluster].gravity);

else if (clusters[currCluster].x < points[i].x)

clusters[currCluster].x = clusters[currCluster].x + (points[i].x - clusters[currCluster].x) / (clusters[currCluster].gravity);

if (clusters[currCluster].y > points[i].y)

clusters[currCluster].y = clusters[currCluster].y - (clusters[currCluster].y - points[i].y) / (clusters[currCluster].gravity);

else if (clusters[currCluster].y < points[i].y)

clusters[currCluster].y = clusters[currCluster].y + (points[i].y - clusters[currCluster].y) / (clusters[currCluster].gravity);

clusters[currCluster].gravity++;

}

}

for (j = 0; j < num\_points\_per\_proc; j++)

{

int currCluster = points[j].cluster\_id;

if (currCluster >= 0)

{

currentPoint.x = points[j].x;

currentPoint.y = points[j].y;

current\_c\_point.x = clusters[currCluster].x;

current\_c\_point.y = clusters[currCluster].y;

double currDist = euclidean\_dist(&currentPoint, &current\_c\_point);

if (currDist > maxDistanceFromCOM[currCluster]) {

maxDistanceFromCOM[currCluster] = currDist;

borderPoint[currCluster] = currentPoint;

}

}

}

if (clusters\_count != 0)

{

for (i = 0; i < clusters\_count; i++)

{

clusters[i].cluster\_id = i;

clusters[i].points\_count = clusters[i].gravity - 1;

clusters[i].radius = maxDistanceFromCOM[i];

//printf("Cluster %d info: COM:(%lf,%lf), Count:%d, Radius:%lf from COM to point:(%lf,%lf), from rank %d\n", clusters[i].cluster\_id, clusters[i].x, clusters[i].y, clusters[i].points\_count, clusters[i].radius, borderPoint[i].x, borderPoint[i].y, rank);

}

}

else

{

printf("Rank %d has no clusters", rank);

}

return clusters;

}

//Compare Clusters

cluster\_t \*compare\_clusters(cluster\_t \*clusters, cluster\_t \*send\_cluster, point\_t \*points, point\_t \*recv\_points, int cluster\_count, int cluster\_count\_send, int minpts, double eps)

{

point\_t a\_1, a\_2;

int i = 0, index = 0, j = 0, c = 0, count\_intersects = 0, points\_c = 0, send\_points\_c = 0;

double maxdist;

bool flag = false;

cluster\_t \*merged\_cl = (cluster\_t \*)calloc(cluster\_count\_send + cluster\_count, sizeof(cluster\_t));

for (i = 0; i < cluster\_count\_send + cluster\_count; i++)

merged\_cl[i].exists = false;

int \*send\_cluster\_note = (int \*)calloc(cluster\_count\_send, sizeof(int));

for (i = 0; i < cluster\_count\_send; i++)

send\_cluster\_note[i] = 0;

for (index = 0; index < cluster\_count; index++)

{

flag = false;

for (i = 0; i < cluster\_count\_send; i++)

{

point\_t \*merged\_cl\_points = (point\_t \*)calloc(clusters[index].points\_count + send\_cluster[i].points\_count, sizeof(point\_t));

a\_1.x = clusters[index].x;

a\_1.y = clusters[index].y;

a\_2.x = send\_cluster[i].x;

a\_2.y = send\_cluster[i].y;

if (fabs(clusters[index].radius - send\_cluster[i].radius) >= euclidean\_dist(&a\_1, &a\_2)) // one circle lies completely inside another

{

if (clusters[index].radius > send\_cluster[i].radius)

{

printf("Cluster %d lies completely inside cluster %d\n", send\_cluster[i].cluster\_id, clusters[index].cluster\_id);

merged\_cl[c].x = (clusters[index].x + send\_cluster[i].x) / 2;

merged\_cl[c].y = (clusters[index].y + send\_cluster[i].y) / 2;

merged\_cl[c].gravity = clusters[index].gravity + send\_cluster[i].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].radius = clusters[index].radius;

merged\_cl[c].points\_count = clusters[index].points\_count + send\_cluster[i].points\_count;

merged\_cl[c].exists = true;

c++;

flag = true;

}

else

{

printf("Cluster %d lies completely inside cluster %d\n", clusters[index].cluster\_id, send\_cluster[i].cluster\_id);

merged\_cl[c].x = (clusters[index].x + send\_cluster[i].x) / 2;

merged\_cl[c].y = (clusters[index].y + send\_cluster[i].y) / 2;

merged\_cl[c].gravity = clusters[index].gravity + send\_cluster[i].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].radius = send\_cluster[i].radius;

merged\_cl[c].points\_count = clusters[index].points\_count + send\_cluster[i].points\_count;

merged\_cl[c].exists = true;

c++;

flag = true;

}

}

else if (fabs(clusters[index].radius - send\_cluster[i].radius) < euclidean\_dist(&a\_1, &a\_2)

&& (clusters[index].radius + send\_cluster[i].radius) > euclidean\_dist(&a\_1, &a\_2)) // intersect in two points

{

point\_t curr\_c\_pnt, currpnt;

curr\_c\_pnt.x = (send\_cluster[i].x + clusters[index].x) / 2;

curr\_c\_pnt.y = (send\_cluster[i].y + clusters[index].y) / 2;

maxdist = 0;

for (int j = 0; j < clusters[index].points\_count; j++)

{

if ((euclidean\_dist(&points[j + points\_c], &a\_2) <= send\_cluster[i].radius))

{

count\_intersects++;

}

currpnt.x = clusters[index].x;

currpnt.y = clusters[index].y;

if (euclidean\_dist(&currpnt, &curr\_c\_pnt) >= maxdist)

maxdist = euclidean\_dist(&currpnt, &curr\_c\_pnt);

}

for (int j = 0; j < send\_cluster[i].points\_count; j++)

{

if ((euclidean\_dist(&recv\_points[j + send\_points\_c], &a\_1) <= clusters[index].radius))

{

count\_intersects++;

}

currpnt.x = send\_cluster[i].x;

currpnt.y = send\_cluster[i].y;

if (euclidean\_dist(&currpnt, &curr\_c\_pnt) >= maxdist)

maxdist = euclidean\_dist(&currpnt, &curr\_c\_pnt);

}

if (count\_intersects > 0)

{

merged\_cl[c].x = (clusters[index].x + send\_cluster[i].x) / 2;

merged\_cl[c].y = (clusters[index].y + send\_cluster[i].y) / 2;

merged\_cl[c].gravity = clusters[index].gravity + send\_cluster[i].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].points\_count = clusters[index].points\_count + send\_cluster[i].points\_count;

merged\_cl[c].radius = maxdist;

merged\_cl[c].exists = true;

c++;

flag = true;

printf("Cluster %d intersects in two points with cluster %d and the intersected points are %d\n",

send\_cluster[i].cluster\_id, clusters[index].cluster\_id, count\_intersects);

}

else

{

printf("Cluster %d intersects in two points with cluster %d but have no intersected points and the clusters don't merge\n",

send\_cluster[i].cluster\_id, clusters[index].cluster\_id);

send\_cluster\_note[i] = send\_cluster\_note[i] + 1;

}

}

else if ((clusters[index].radius + send\_cluster[i].radius) == euclidean\_dist(&a\_1, &a\_2)) // intersect in one point

{

point\_t curr\_c\_pnt, currpnt, int\_pnt;

curr\_c\_pnt.x = (send\_cluster[i].x + clusters[index].x) / 2;

curr\_c\_pnt.y = (send\_cluster[i].y + clusters[index].y) / 2;

maxdist = 0;

for (int j = 0; j < clusters[index].points\_count; j++)

{

if ((euclidean\_dist(&points[j + points\_c], &a\_2) == send\_cluster[i].radius))

{

int\_pnt = points[j + points\_c];

count\_intersects++;

}

currpnt.x = clusters[index].x;

currpnt.y = clusters[index].y;

if (euclidean\_dist(&currpnt, &curr\_c\_pnt) >= maxdist)

maxdist = euclidean\_dist(&currpnt, &curr\_c\_pnt);

}

for (int j = 0; j < send\_cluster[i].points\_count; j++)

{

if ((euclidean\_dist(&recv\_points[j + send\_points\_c], &a\_1) == clusters[index].radius))

{

int\_pnt = recv\_points[j + send\_points\_c];

count\_intersects++;

}

currpnt.x = send\_cluster[i].x;

currpnt.y = send\_cluster[i].y;

if (euclidean\_dist(&currpnt, &curr\_c\_pnt) >= maxdist)

maxdist = euclidean\_dist(&currpnt, &curr\_c\_pnt);

}

if (count\_intersects > 0)

{

merged\_cl[c].x = (clusters[index].x + send\_cluster[i].x) / 2;

merged\_cl[c].y = (clusters[index].y + send\_cluster[i].y) / 2;

merged\_cl[c].gravity = clusters[index].gravity + send\_cluster[i].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].points\_count = clusters[index].points\_count + send\_cluster[i].points\_count;

merged\_cl[c].radius = maxdist;

merged\_cl[c].exists = true;

c++;

flag = true;

printf("Cluster %d intersects in one point with cluster %d and the intersected point is (%lf,%lf)\n",

send\_cluster[i].cluster\_id, clusters[index].cluster\_id, int\_pnt.x, int\_pnt.y);

}

else

{

printf("Cluster %d intersects in one point with cluster %d but the intersected point doesn't belong in the dataset, so the clusters don't merge\n",

send\_cluster[i].cluster\_id, clusters[index].cluster\_id);

send\_cluster\_note[i] = send\_cluster\_note[i] + 1;

}

}

else // do not intersect

{

printf("Clusters %d and %d have nothing in common\n", send\_cluster[i].cluster\_id, clusters[index].cluster\_id);

send\_cluster\_note[i] = send\_cluster\_note[i] + 1;

}

send\_points\_c = send\_points\_c + send\_cluster[i].points\_count;

}

//printf("\n\n");

if (!flag)

{

merged\_cl[c].x = clusters[index].x;

merged\_cl[c].y = clusters[index].y;

merged\_cl[c].gravity = clusters[index].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].points\_count = clusters[index].points\_count;

merged\_cl[c].radius = clusters[index].radius;

merged\_cl[c].exists = true;

c++;

}

points\_c = points\_c + clusters[index].points\_count;

}

for (i = 0; i < cluster\_count\_send; i++)

{

if (send\_cluster\_note[i] == cluster\_count)

{

merged\_cl[c].x = send\_cluster[i].x;

merged\_cl[c].y = send\_cluster[i].y;

merged\_cl[c].gravity = send\_cluster[i].gravity;

merged\_cl[c].cluster\_id = c;

merged\_cl[c].points\_count = send\_cluster[i].points\_count;

merged\_cl[c].radius = send\_cluster[i].radius;

merged\_cl[c].exists = true;

c++;

}

}

printf("No of cluster after merge: %d\n", c);

return merged\_cl;

}

cluster\_t \*find\_noise\_relation(cluster\_t \*clusters, point\_t \*points, int num\_points\_per\_proc, int c)

{

int i,j;

point\_t merged\_c;

for (i = 0; i < num\_points\_per\_proc \* 2; i++)

{

if (points[i].cluster\_id == -2)

{

for (j = 0; j < c; j++)

{

merged\_c.x = clusters[j].x;

merged\_c.y = clusters[j].y;

if (euclidean\_dist(&points[i], &merged\_c) <= clusters[j].radius)

{

clusters[j].points\_count += 1;

points[i].cluster\_id = clusters[j].cluster\_id;

}

}

}

}

return clusters;

}

void print\_cluster\_info(int clusters\_count, cluster\_t \* clusters, int rank, point\_t \* points)

{

int pts\_sum = 0, j = 0, jj = 0;

if (clusters\_count != 0)

{

for (int i = 0; i < clusters\_count; i++)

{

printf("Cluster %d info: COM:(%lf,%lf), Count:%d, Radius:%lf, rank: %d\n", clusters[i].cluster\_id, clusters[i].x, clusters[i].y, clusters[i].points\_count, clusters[i].radius, rank);

for (j = 0; j < clusters[i].points\_count; j++)

{

jj = j;

while (points[jj + pts\_sum].cluster\_id == -2)

{

jj++;

}

printf("Cluster %d points: x=%lf, y=%lf\n", clusters[i].cluster\_id, points[jj + pts\_sum].x, points[jj + pts\_sum].y);

}

pts\_sum = pts\_sum + clusters[i].points\_count + jj;

}

}

else

printf("Rank %d has no clusters", rank);

}

//Main

int main(int argc, char \*argv[])

{

point\_t \*points;

double epsilon;

unsigned int minpts, num\_points, num\_points\_per\_proc, i = 0, k = 0, j = 0;

int rank, size, clusters\_count = 0;

cluster\_t \*clusters;

clock\_t t;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0)

{

t = clock();

printf("Begin Time Calculation...\n");

FILE \*file = fopen("input.txt", "r+");

fscanf(file, "%lf %u %u\n", &epsilon, &minpts, &num\_points);

printf("Epsilon: %lf\n", epsilon);

printf("Minimum points: %u\n", minpts);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank != 0)

{

int start, end;

double hi, psi;

FILE \*file = fopen("input.txt", "r+");

fscanf(file, "%lf %u %u\n", &epsilon, &minpts, &num\_points);

num\_points\_per\_proc = num\_points / (size - 1);

point\_t \*p = (point\_t \*)calloc(num\_points\_per\_proc, sizeof(point\_t));

if (p == NULL) {

perror("Failed to allocate points array");

return 0;

}

start = (num\_points\_per\_proc \* (rank - 1));

if (rank == size - 1)

end = num\_points;

else

end = num\_points\_per\_proc \* rank;

while (i < num\_points) {

if (i == start)

while (i < end)

{

fscanf(file, "%lf %lf\n", &(p[j].x), &(p[j].y));

p[j].cluster\_id = UNCLASSIFIED;

++i;

++j;

}

fscanf(file, "%lf %lf\n", &hi, &psi);

++i;

}

points = p;

fclose(file);

if (num\_points\_per\_proc)

{

clusters\_count = dbscan(points, num\_points\_per\_proc, epsilon, minpts, euclidean\_dist);

//print\_points(points, num\_points\_per\_proc);

}

/\* create a type for struct points\_t \*/

const int nitems\_points = 3;

int blocklengths\_points[3] = { 1,1,1 };

MPI\_Datatype types\_points[3] = { MPI\_DOUBLE, MPI\_DOUBLE, MPI\_INT };

MPI\_Datatype mpi\_points\_type;

MPI\_Aint offsets\_points[3];

offsets\_points[0] = offsetof(point\_t, x);

offsets\_points[1] = offsetof(point\_t, y);

offsets\_points[2] = offsetof(point\_t, cluster\_id);

MPI\_Type\_create\_struct(nitems\_points, blocklengths\_points, offsets\_points, types\_points, &mpi\_points\_type);

MPI\_Type\_commit(&mpi\_points\_type);

/\* create a type for struct clusters \*/

const int nitems\_cl = 7;

int blocklengths\_cl[7] = { 1,1,1,1,1,1,1 };

MPI\_Datatype types\_cl[7] = { MPI\_DOUBLE, MPI\_DOUBLE, MPI\_INT, MPI\_INT, MPI\_DOUBLE, MPI\_INT, MPI\_C\_BOOL };

MPI\_Datatype mpi\_cl\_type;

MPI\_Aint offsets\_cl[7];

offsets\_cl[0] = offsetof(cluster\_t, x);

offsets\_cl[1] = offsetof(cluster\_t, y);

offsets\_cl[2] = offsetof(cluster\_t, gravity);

offsets\_cl[3] = offsetof(cluster\_t, cluster\_id);

offsets\_cl[4] = offsetof(cluster\_t, radius);

offsets\_cl[5] = offsetof(cluster\_t, points\_count);

offsets\_cl[6] = offsetof(cluster\_t, exists);

MPI\_Type\_create\_struct(nitems\_cl, blocklengths\_cl, offsets\_cl, types\_cl, &mpi\_cl\_type);

MPI\_Type\_commit(&mpi\_cl\_type);

//printf("Clusters of rank %d\n", rank);

clusters = clusters\_init(points, num\_points\_per\_proc, clusters\_count, rank);

int n = 2, m = 0, c = 0;

while (m <= log(size - 1))

{

if (rank % n == n / 2)

{

MPI\_Send(&clusters\_count, 1, MPI\_INT, rank + n / 2, 10, MPI\_COMM\_WORLD);

MPI\_Send(points, num\_points\_per\_proc, mpi\_points\_type, rank + n / 2, 20, MPI\_COMM\_WORLD);

MPI\_Send(clusters, clusters\_count, mpi\_cl\_type, rank + n / 2, 30, MPI\_COMM\_WORLD);

//printf("Rank %d: sent struct clusters to rank %d\n", rank, rank + n / 2);

}

else if (rank % n == 0)

{

int recv\_cluster\_count, no\_of\_intersections;

c = 0;

MPI\_Status status;

MPI\_Recv(&recv\_cluster\_count, 1, MPI\_INT, rank - n / 2, 10, MPI\_COMM\_WORLD, &status);

cluster\_t \*recv\_cluster = (cluster\_t \*)calloc(recv\_cluster\_count, sizeof(cluster\_t));

point\_t \*recv\_points = (point\_t \*)calloc(num\_points\_per\_proc, sizeof(point\_t));

MPI\_Recv(recv\_points, num\_points\_per\_proc, mpi\_points\_type, rank - n / 2, 20, MPI\_COMM\_WORLD, &status);

MPI\_Recv(recv\_cluster, recv\_cluster\_count, mpi\_cl\_type, rank - n / 2, 30, MPI\_COMM\_WORLD, &status);

/\*print\_cluster\_info(clusters\_count, clusters, rank, points);

printf("\n");

print\_cluster\_info(recv\_cluster\_count, recv\_cluster, rank - n / 2, recv\_points);

printf("\n\n");\*/

cluster\_t \*merged\_cl = (cluster\_t \*)calloc(recv\_cluster\_count + clusters\_count, sizeof(cluster\_t));

merged\_cl = compare\_clusters(clusters, recv\_cluster, points, recv\_points, clusters\_count, recv\_cluster\_count, minpts, epsilon);

for (i = 0; i < recv\_cluster\_count + clusters\_count; i++)

{

if (merged\_cl[i].exists == true)

{

c += 1;

}

}

clusters = merged\_cl;

point\_t \*merged\_pts = (point\_t \*)calloc(num\_points\_per\_proc \* 2, sizeof(point\_t));

for (i = 0; i < num\_points\_per\_proc \* 2; i++)

{

if (i < num\_points\_per\_proc)

merged\_pts[i] = points[i];

else

merged\_pts[i] = recv\_points[i - num\_points\_per\_proc];

}

points = merged\_pts;

clusters\_count = c;

merged\_cl = find\_noise\_relation(clusters, points, num\_points\_per\_proc, c);

printf("Merge of ranks %d and %d\n", rank, rank - n / 2);

for (i = 0; i < c; i++)

{

printf("Merged Cluster: %d, com:(%lf,%lf), radius:%lf, CountOfPoints:%d\n",

merged\_cl[i].cluster\_id, merged\_cl[i].x, merged\_cl[i].y, merged\_cl[i].radius, merged\_cl[i].points\_count);

}

printf("\n\n");

}

n \*= 2; m += 1;

}

free(points);

free(clusters);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (rank == 0)

{

t = (clock() - t);

float timeSpentQS = ((float)t) / CLOCKS\_PER\_SEC;

printf("It took me %d clicks to calculate DBSCAN using %u processors (%.3f seconds).\n", t, size, timeSpentQS);

}

MPI\_Finalize();

return 0;

}