Homework11

cat1997

December 8, 2019

1 Source files

1.1 centroids.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "data.h"
#include "clusters.h"

// perform kaverage clustering of data from file
double mpiStartTimer(){
  MPI_Barrier(MPI_COMM_WORLD);
  return MPI_Wtime();
}

double mpiEndTimer(double start){
  double end = MPI_Wtime();
  int N = 1;
  double *message = (double*) calloc(N, sizeof(double));
  double *elapsed = (double*) calloc(N, sizeof(double));
  message[0] = end-start;
  MPI_Allreduce(message, elapsed, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

  double res = elapsed[0];
  free(elapsed);
  free(message);

  return res;
}



int main(int argc, char **argv){
  // initialize MPI
  MPI_Init(&argc, &argv);

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


  // parse command line arguments
  if(argc<2){
    printf("usage: ./kaverage fileName.dat\n");
    return 0;
  }

  // initialize random seed
  srand48(123456);

  // read data from file
  data_t data = dataReadWithClusterIds(argv[1]);

  double start = mpiStartTimer();
  // compute cluster centroids
  clusters_t clusters = clusterSetupFromData(data);

  // compute centroids
  clusterComputeCentroids(clusters, data);
  double time = mpiEndTimer(start);

 if(rank == 0){
    printf("Size: %d Time: %f\n", size, time);
  }

  // output data
  clusterOutput(clusters, "scatterdata.txt");

  // finalize MPI
  MPI_Finalize();

  return 0;
}
```

1.2 clusters.c

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "data.h"
#include "clusters.h"

// set up random centroids for clusters
clusters_t clusterRandomSetup(int clusterK, data_t data){

  clusters_t clusters;

  clusters.K = clusterK;
  clusters.x = (float*) calloc(clusterK, sizeof(float));
  clusters.y = (float*) calloc(clusterK, sizeof(float));

  // work space for building new cluster centers
  clusters.tmpDegree = (int*) calloc(clusterK, sizeof(int));
  clusters.tmpx = (float*) calloc(clusterK, sizeof(float));
  clusters.tmpy = (float*) calloc(clusterK, sizeof(float));

  // initialize cluster centers
  for(int k=0;k<clusterK;++k){
    clusters.x[k] = drand48();
    clusters.y[k] = drand48();
  }

  // MPI buffer space for building new cluster centers
  clusters.bufDegree = (int*) calloc(clusterK, sizeof(int));
  clusters.bufx = (float*) calloc(clusterK, sizeof(float));
  clusters.bufy = (float*) calloc(clusterK, sizeof(float));

  return clusters;
}

// set up centroids from data
clusters_t clusterSetupFromData(data_t data){

  // find maximum cluster ID
  int clusterK = 0;
  for(int n=0;n<data.N;++n){
    int c = data.clusterIds[n];
    if(c>clusterK){
      clusterK = c;
    }
  }
  ++clusterK;

  // find maximum cluster idx
  int maxK;
  MPI_Allreduce(&clusterK, &maxK, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

  clusters_t clusters;

  clusters.K = maxK;
  clusters.x = (float*) calloc(clusterK, sizeof(float));
  clusters.y = (float*) calloc(clusterK, sizeof(float));

  // work space for building new cluster centers
  clusters.tmpDegree = (int*) calloc(clusterK, sizeof(int));
  clusters.tmpx = (float*) calloc(clusterK, sizeof(float));
  clusters.tmpy = (float*) calloc(clusterK, sizeof(float));

  // MPI buffer space for building new cluster centers
  clusters.bufDegree = (int*) calloc(clusterK, sizeof(int));
  clusters.bufx = (float*) calloc(clusterK, sizeof(float));
  clusters.bufy = (float*) calloc(clusterK, sizeof(float));

  return clusters;
}


// compute centers of clusters
float clusterComputeCentroids(clusters_t clusters, data_t data){

  int K = clusters.K;

  float totalChange = 0;

  // zero accumulators for x,y,degree for all clusters
  for(int k=0;k<K;++k){
    clusters.bufx[k] = 0;
    clusters.bufy[k] = 0;
    clusters.bufDegree[k] = 0;
  }

  // sum up coordinates of all data points in each cluster
  for(int n=0;n<data.N;++n){
    int k = data.clusterIds[n];
    clusters.bufx[k] += data.x[n];
    clusters.bufy[k] += data.y[n];
    ++(clusters.bufDegree[k]);
  }

  // sum up cluster x,y,counts globally
  MPI_Allreduce(clusters.bufx, clusters.tmpx, clusters.K, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD
  MPI_Allreduce(clusters.bufy, clusters.tmpy, clusters.K, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD
  MPI_Allreduce(clusters.bufDegree, clusters.tmpDegree, clusters.K, MPI_FLOAT, MPI_SUM, MPI_

  // compute centroids for clusters
  for(int k=0;k<clusters.K;++k){
    int degree = clusters.tmpDegree[k];
    if(degree>0){
      clusters.tmpx[k] /= (float) degree;
      clusters.tmpy[k] /= (float) degree;
    }else{
      // if empty cluster then choose random center
      clusters.tmpx[k] = drand48();
      clusters.tmpy[k] = drand48();
    }

    // compute change in cluster centroid
#if USE_POW==1
    float change =
      pow(clusters.x[k]-clusters.tmpx[k], 2) +
      pow(clusters.y[k]-clusters.tmpy[k], 2);
#else
    float dx = clusters.x[k]-clusters.tmpx[k];
    float dy = clusters.y[k]-clusters.tmpy[k];

    float change = dx*dx+dy*dy;
#endif

    totalChange += change;

    clusters.x[k] = clusters.tmpx[k];
    clusters.y[k] = clusters.tmpy[k];
  }

  return totalChange;
}

// label data points with cluster index
void clusterAssignDataPoints(clusters_t clusters, data_t data){

  //for each data point
  for(int n=0;n<data.N;++n){

    // find its old cluster
    int oldk = data.clusterIds[n];

    float xn = data.x[n];
    float yn = data.y[n];
    data.clusterIds[n] = 0;

#if USE_POW==1
    float minDistanceSquared = pow(xn-clusters.x[0],2) + pow(yn-clusters.y[0],2);
#else
    float dx = xn-clusters.x[0];
    float dy = yn-clusters.y[0];
    float minDistanceSquared = dx*dx+dy*dy;
#endif
    // find closest centroid
    for(int k=1;k<clusters.K;++k){

#if USE_POW==1
      float distanceSquared = pow(xn-clusters.x[k],2) + pow(yn-clusters.y[k],2);
#else
      dx = xn-clusters.x[k];
      dy = yn-clusters.y[k];
      float distanceSquared = dx*dx+dy*dy;
#endif
      if(distanceSquared<minDistanceSquared){
        data.clusterIds[n] = k;
        minDistanceSquared = distanceSquared;
      }
    }
  }
}

// output clusters to file from rank 0
void clusterOutput(clusters_t clusters, const char *fileName){

  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank==0){
    FILE *fp = fopen(fileName, "w");

    fprintf(fp, "x y cluster\n");
    for(int k=0;k<clusters.K;++k){
      fprintf(fp, "%f %f %d\n",
              clusters.x[k],
              clusters.y[k],
              k);
    }
    fclose(fp);
  }
}
```

1.3 kmeans.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "omp.h"

#include "data.h"
#include "clusters.h"

// some comment

/*

   kmeans.c

   Purpose: read a data file and use kmeans clustering to assign data points to clusters

   Usage: ./kmeans dataFile.dat numberClusters

   To compile: gcc -o kmeans kmeans.c clusters.c data.c -lm

*/


// perform kmeans clustering using naive LLoyd's algorithm
int kmeans(data_t data, clusters_t clusters){

  float totalChange = 0;
  float tol = 1e-5;

  int iterations = 0;
  do{

    clusterAssignDataPoints(clusters, data);

    totalChange = clusterComputeCentroids(clusters, data);
    ++iterations;

    printf("totalChange = %g\n", totalChange);

  }while(totalChange>tol*tol);

  printf("iterations = %d\n", iterations);

  return iterations;
}

// output data to file
void kmeansOutput(data_t data, clusters_t clusters, const char *fileName){

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank==0){
    FILE *fp = fopen(fileName, "w");

    fprintf(fp, "x y cluster\n");
    for(int k=0;k<data.N;++k){
      fprintf(fp, "%f %f %d\n",
              data.x[k],
              data.y[k],
              data.clusterIds[k]+1);
    }
    fclose(fp);
  }
}


// perform kmeans clustering of data from file
int main(int argc, char **argv){

  // initialize MPI
  MPI_Init(&argc, &argv);

  // parse command line arguments
  if(argc<3){
    printf("usage: ./kmeans fileName.dat clusterCount\n");
    return 0;
  }

  int clusterK = atoi(argv[2]);
  // initialize random seed
  srand48(123456);

  // read data from file
  data_t data = dataRead(argv[1]);

  // initialize cluster centroids
  clusters_t clusters = clusterRandomSetup(clusterK, data);

  // perform kmeans clustering
  double tic = omp_get_wtime();
  int iterations = kmeans(data, clusters);
  double toc = omp_get_wtime();

  printf("elapsed time per iteration = %3.2g\n", (toc-tic)/iterations);

  // output data
  kmeansOutput(data, clusters, "scatterdata.txt");

  // shutdown MPI
  MPI_Finalize();

  return 0;
}
```

1.4 data.c

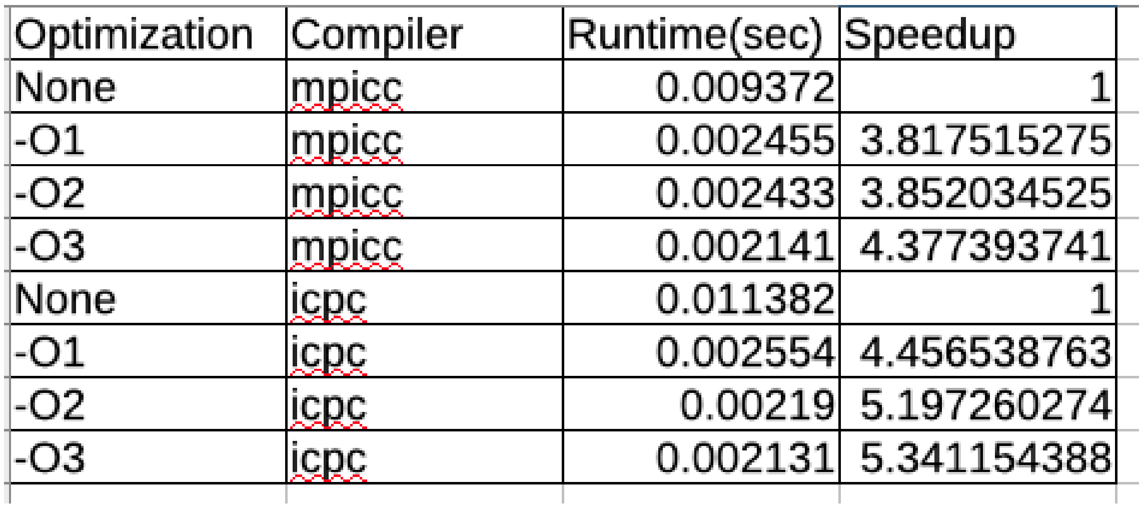
```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "data.h"
#include "clusters.h"

// read data from file
data_t dataRead(const char *fileName){

  FILE *fp = fopen(fileName, "r");

  if(fp==NULL){
    printf("Failed to open file named: %s\n", fileName);
    exit(0);
  }

  int dataN;
```

```
data_t data;
char buf[BUFSIZ];

// read header
fgets(buf, BUFSIZ, fp);

// read number of data points
fgets(buf, BUFSIZ, fp);
sscanf(buf, "%d", &dataN);

// partition into chunks
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int chunkN = dataN/size;
int remainder = dataN - (chunkN*size);
data.N = chunkN;
if(rank<remainder) data.N += 1; // add 1 to low ranks to make sure have all data points

// allocate local space for data
data.x = (float*) calloc(data.N, sizeof(float));
data.y = (float*) calloc(data.N, sizeof(float));
data.clusterIds = (int*) calloc(data.N, sizeof(int));

// read section header
fgets(buf, BUFSIZ, fp);

// rank 0 reads all data from file and sends to other ranks
int tag = 999;
if(rank==0){

  // read own stuff
  for(int n=0;n<data.N;++n){
    fgets(buf, BUFSIZ, fp);
    sscanf(buf, "%f %f", data.x+n, data.y+n);
  }

  // buffer space for data to send
  float *bufx = (float*) calloc(chunkN+1, sizeof(float));
  float *bufy = (float*) calloc(chunkN+1, sizeof(float));

  // loop over ranks
  for(int r=1;r<size;++r){
    int sendN = (r<remainder) ? chunkN+1:chunkN;

    // read chunk to send to rank r
```

```
      for(int n=0;n<sendN;++n){
        fgets(buf, BUFSIZ, fp);
        sscanf(buf, "%f %f", bufx+n, bufy+n);
      }

      MPI_Send(bufx, sendN, MPI_FLOAT, r, tag+1, MPI_COMM_WORLD);
      MPI_Send(bufy, sendN, MPI_FLOAT, r, tag+2, MPI_COMM_WORLD);
    }

    // this rank is done with file
    fclose(fp);
  }
  else{
    // this rank is done with file
    fclose(fp);

    MPI_Status status;
    int recvN = data.N;
    int root = 0;
    MPI_Recv(data.x, recvN, MPI_FLOAT, root, tag+1, MPI_COMM_WORLD, &status);
    MPI_Recv(data.y, recvN, MPI_FLOAT, root, tag+2, MPI_COMM_WORLD, &status);
  }


  // print out a message from each rank
  printf("Process rank %d owns %d data points\n", rank, data.N);

  return data;
}




// read data from file
data_t dataReadWithClusterIds(const char *fileName){

  FILE *fp = fopen(fileName, "r");

  if(fp==NULL){
    printf("Failed to open file named: %s\n", fileName);
    exit(0);
  }

  int dataN;
  data_t data;
  char buf[BUFSIZ];
```

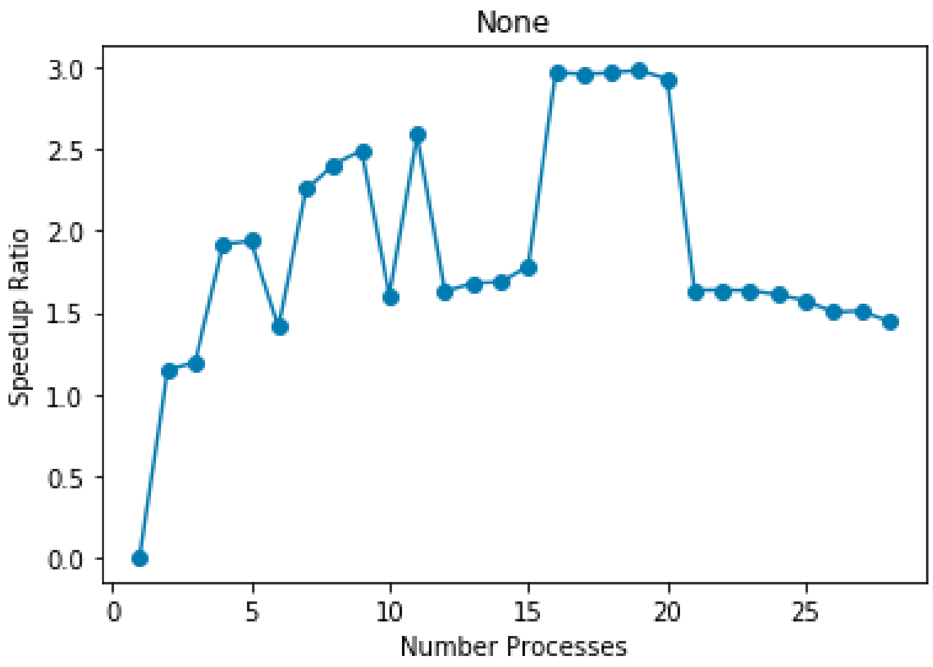
```
// read header
fgets(buf, BUFSIZ, fp);

// read number of data points
fgets(buf, BUFSIZ, fp);
sscanf(buf, "%d", &dataN);

// partition into chunks
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int chunkN = dataN/size;
int remainder = dataN - (chunkN*size);
data.N = chunkN;
if(rank<remainder) data.N += 1;

data.x = (float*) calloc(data.N, sizeof(float));
data.y = (float*) calloc(data.N, sizeof(float));
data.clusterIds = (int*) calloc(data.N, sizeof(int));

// read section header
fgets(buf, BUFSIZ, fp);

int tag = 999;
if(rank==0){

  // read own stuff
  for(int n=0;n<data.N;++n){
    fgets(buf, BUFSIZ, fp);
    sscanf(buf, "%f %f %d", data.x+n, data.y+n, data.clusterIds+n);
  }

  // buffer space for data to send
  float *bufx = (float*) calloc(chunkN+1, sizeof(float));
  float *bufy = (float*) calloc(chunkN+1, sizeof(float));
  int   *bufClusterIds = (int*) calloc(chunkN+1, sizeof(int));

  // loop over ranks
  for(int r=1;r<size;++r){
    int sendN = (r<remainder) ? chunkN+1:chunkN;

    // read chunk to send to rank r
    for(int n=0;n<sendN;++n){
      fgets(buf, BUFSIZ, fp);
      sscanf(buf, "%f %f %d", bufx+n, bufy+n, bufClusterIds+n);
      }

      MPI_Send(bufx, sendN, MPI_FLOAT, r, tag+1, MPI_COMM_WORLD);
      MPI_Send(bufy, sendN, MPI_FLOAT, r, tag+2, MPI_COMM_WORLD);
      MPI_Send(bufClusterIds, sendN, MPI_INT, r, tag+3, MPI_COMM_WORLD);
    }

    // root done with file
    fclose(fp);


  }
  else{
    // this rank done with file
    fclose(fp);

    MPI_Status status;
    int recvN = data.N;
    int root = 0;
    MPI_Recv(data.x, recvN, MPI_FLOAT, root, tag+1, MPI_COMM_WORLD, &status);
    MPI_Recv(data.y, recvN, MPI_FLOAT, root, tag+2, MPI_COMM_WORLD, &status);
    MPI_Recv(data.clusterIds, recvN, MPI_INT, root, tag+3, MPI_COMM_WORLD, &status);
  }

  // print out a message from each rank
  printf("Process rank %d owns %d data points\n", rank, data.N);

  return data;
}
```

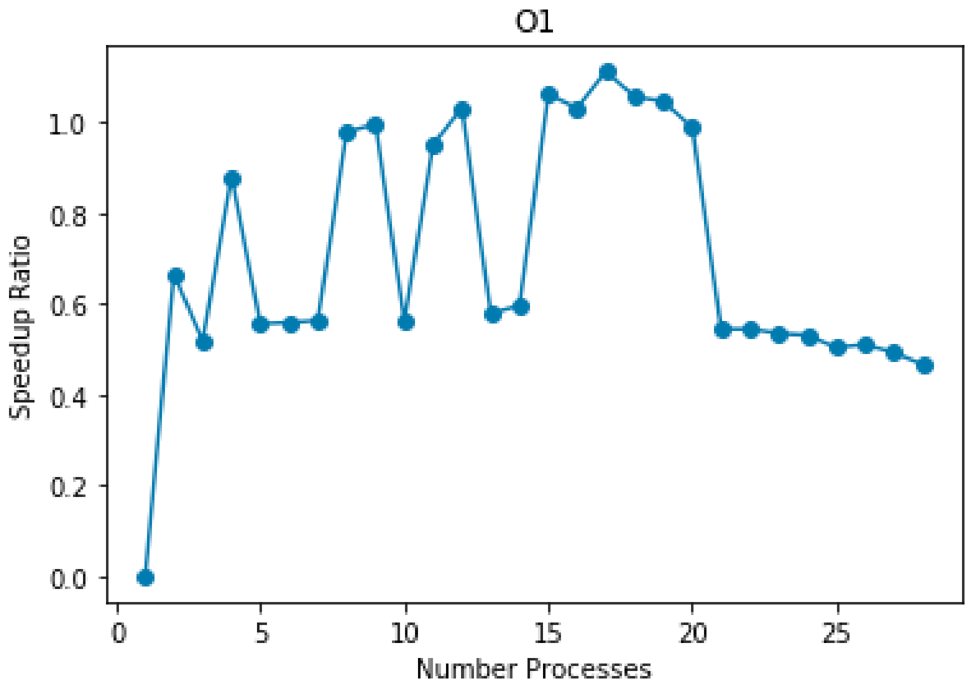
2 Task1

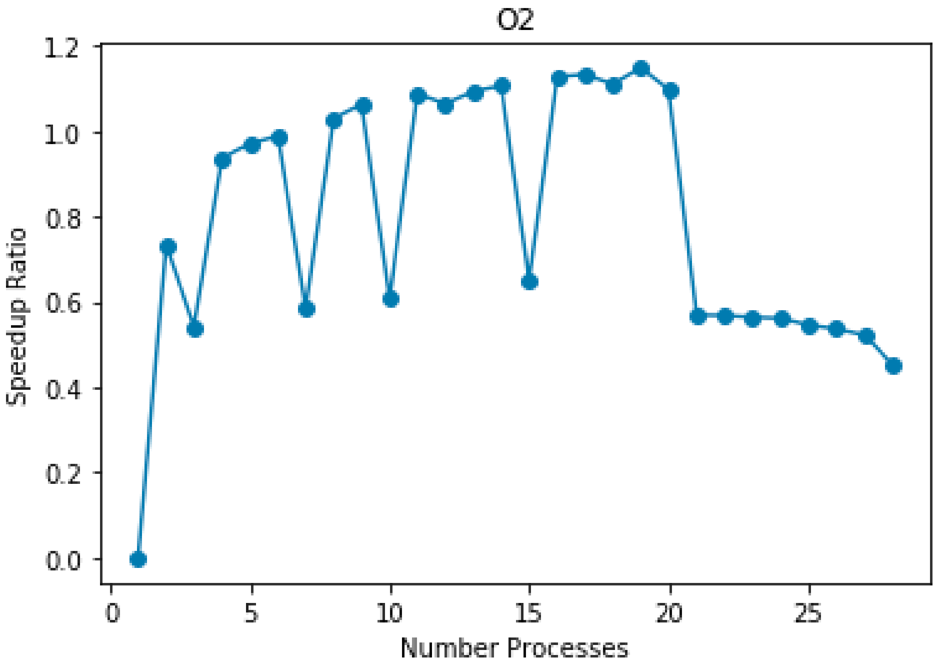


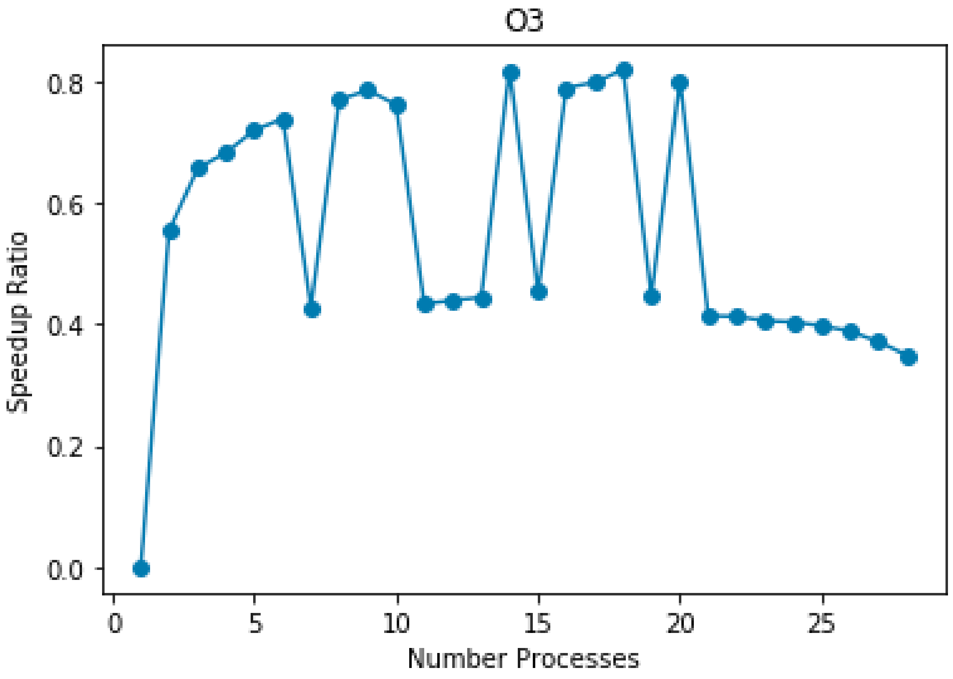
Optimization	Compiler	Runtime(sec)	Speedup
None	mpicc	0.009372	1
-O1	mpicc	0.002455	3.817515275
-O2	mpicc	0.002433	3.852034525
-O3	mpicc	0.002141	4.377393741
None	icpc	0.011382	1
-O1	icpc	0.002554	4.456538763
-O2	icpc	0.00219	5.197260274
-O3	icpc	0.002131	5.341154388

3 Task2

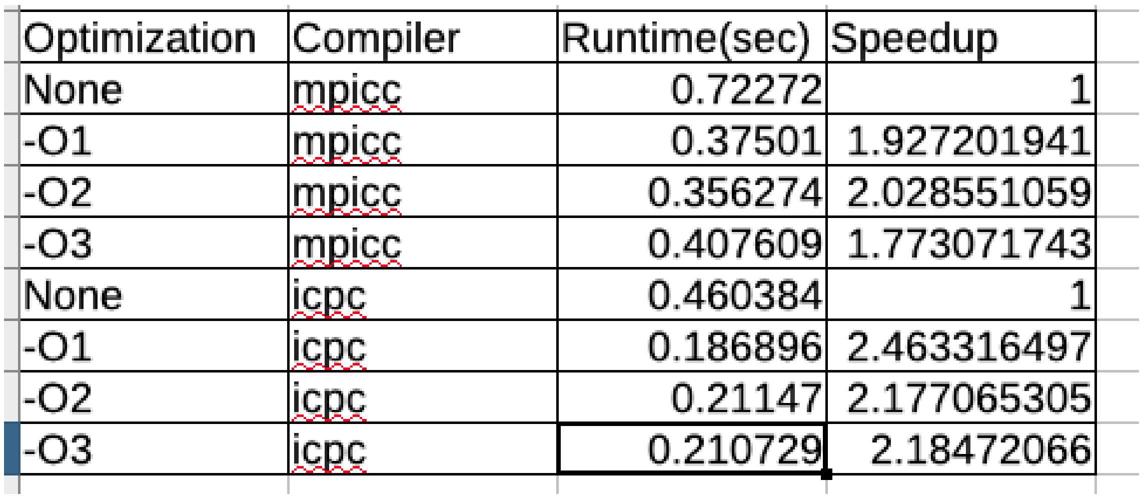

None
Speedup Ratio
Number Processes
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25


O1
Speedup Ratio
Number Processes
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25


O2
Speedup Ratio
Number Processes
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
5
10
15
20
25


O3
Speedup Ratio
Number Processes
0.0
0.2
0.4
0.6
0.8
0
5
10
15
20
25

4 Kmeans chart



Optimization	Compiler	Runtime(sec)	Speedup
None	mpicc	0.72272	1
-O1	mpicc	0.37501	1.927201941
-O2	mpicc	0.356274	2.028551059
-O3	mpicc	0.407609	1.773071743
None	icpc	0.460384	1
-O1	icpc	0.186896	2.463316497
-O2	icpc	0.21147	2.177065305
-O3	icpc	0.210729	2.18472066

5 Discussion

I used exactly what was in the HW10 MPI example code to test the speedup ratios as well as the mpiStart and mpiStop time functions in the lesson 21 repositories. I couldn't figure out why the speedup ratio wouldn't reach higher than around 3. It seemed to speed up more or less as it was supposed to for 1-4 processes then plateau and actually start to get slower as more processes were added.