Image segmentation with Parallel Kmeans using MPI and OpenMP

Author: Gautam Shende

CSE 702: Programming Massively Parallel Systems

Instructor: Dr. Russ Miller

Date: 12/06/2018

OVERVIEW

1. Clustering
2. K means
3. Parallel MPI Model
4. Parallel OpenMP Model
5. Results
6. Inferences
7. References

1) CLUSTERING

CLUSTERING

1. Partitioning of data into subsets called clusters

2. Similar elements placed in same cluster. Similarity is calculated based on some distance metric such as euclidean distance or hamming distance.

3. Example :
   Dataset = {US, CHN, IN, CA}
   No of clusters = 2

   Cluster 1: US, CA
   Cluster 2: CHN, IN

2) K-Means

Sequential K means

1. Select k i.e. the number of clusters
2. Use any strategy* to select k points to be cluster centers.
3. Put each point in the data set in the cluster which has its center closest to the point
4. Calculate new cluster centers by taking means of all points in a cluster
5. Repeat 3 and 4 until convergence

EXAMPLE

- U = {1,6,10,18,3,14} , K=2

- ASSUME CLUSTER CENTERS TO BE C1 = 1, C2 = 6

- CLUSTER C1: {1,3} , CLUSTER C2: {6,10,18,14}

- UPDATE CENTRE C1 = AVG {1,3} = 2
  UPDATE CENTRE C2 = AVG {6,10,18,14} = 12

- UPDATED CLUSTER C1: {1,3,6}
  UPDATED CLUSTER C2: {10,18,14}

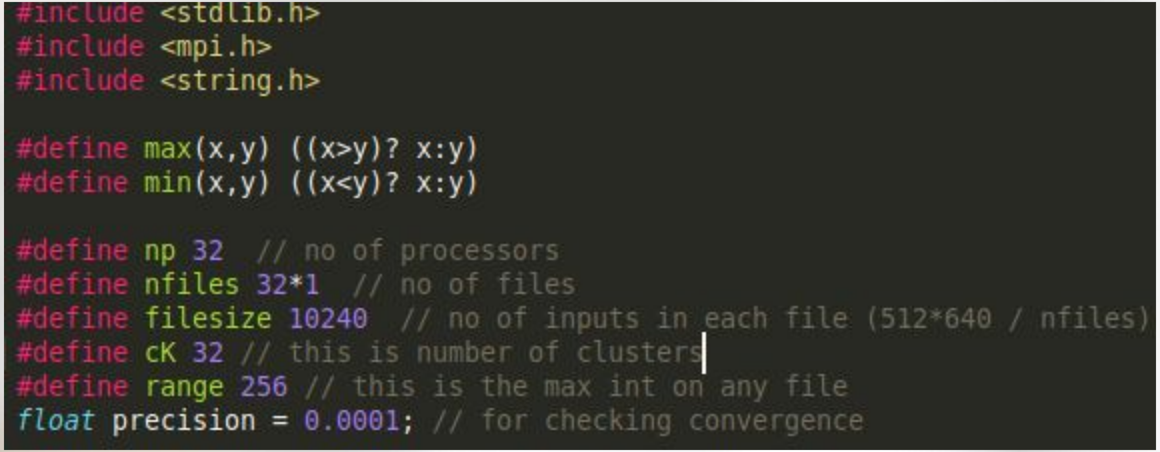
- UPDATE CENTRE C1 = AVG {1,3,6} = 3.333
  UPDATE CENTRE C2 = AVG {10,18,14} = 14

- UPDATED CLUSTER C1: {1,3,6}
  UPDATED CLUSTER C2: {10,18,14}

- NO CHANGE IN CLUSTER CONFIGURATION (CONVERGENCE)
  -> STOP <-

3）Parallel MPI Model

MPI MODEL PARAMETERS



```
#include <stdlib.h>
#include <mpi.h>
#include <string.h>

#define max(x,y) ((x>y)? x:y)
#define min(x,y) ((x<y)? x:y)

#define np 32  // no of processors
#define nfiles 32*1  // no of files
#define filesize 10240  // no of inputs in each file (512*640 / nfiles)
#define cK 32 // this is number of clusters
#define range 256 // this is the max int on any file
float precision = 0.0001; // for checking convergence
```

*local parameter = Max iterations (=25)

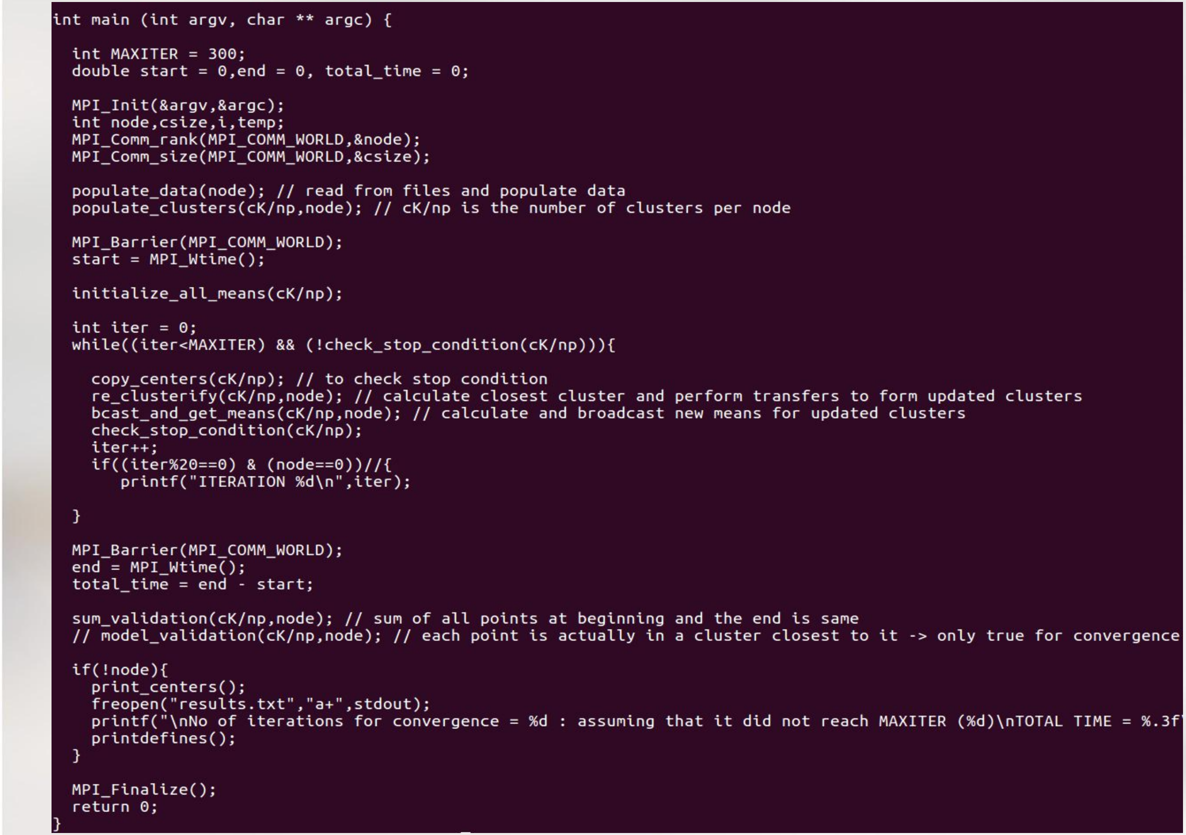
Complexity: O(input*K*iterations*dimensions)

= O(nfiles*filesize*cK*max_iterations*dimensions)

= ~(32*10240*32*25*3) = ~1 billion calculations

Repository: https://github.com/thezodiac1994/Parallel-Alogrithms

MPI MODEL FLOW



```
int main (int argv, char ** argc) {

  int MAXITER = 300;
  double start = 0,end = 0, total_time = 0;

  MPI_Init(&argv,&argc);
  int node,csize,i,temp;
  MPI_Comm_rank(MPI_COMM_WORLD,&node);
  MPI_Comm_size(MPI_COMM_WORLD,&csize);

  populate_data(node); // read from files and populate data
  populate_clusters(cK/np,node); // cK/np is the number of clusters per node

  MPI_Barrier(MPI_COMM_WORLD);
  start = MPI_Wtime();

  initialize_all_means(cK/np);

  int iter = 0;
  while((iter<MAXITER) && (!check_stop_condition(cK/np))){

    copy_centers(cK/np); // to check stop condition
    re_clusterify(cK/np,node); // calculate closest cluster and perform transfers to form updated clusters
    bcast_and_get_means(cK/np,node); // calculate and broadcast new means for updated clusters
    check_stop_condition(cK/np);
    iter++;
    if((iter%20==0) & (node==0))//{
       printf("ITERATION %d\n",iter);

  }

  MPI_Barrier(MPI_COMM_WORLD);
  end = MPI_Wtime();
  total_time = end - start;

  sum_validation(cK/np,node); // sum of all points at beginning and the end is same
  // model_validation(cK/np,node); // each point is actually in a cluster closest to it -> only true for convergence

  if(!node){
    print_centers();
    freopen("results.txt","a+",stdout);
    printf("\nNo of iterations for convergence = %d : assuming that it did not reach MAXITER (%d)\nTOTAL TIME = %.3f
    printdefines();
  }

  MPI_Finalize();
  return 0;
}
```

1. Allot k cluster centers to the nodes(n) equally such that each node is responsible for (k/n) clusters
   < populate_data(), populate_clusters() >

2. Now Each node does the following
   a. Calculate centers of (k/n) clusters by mean
   b. Broadcast (k/n) centers to all other nodes
   c. Receive (k/n) centers from every other node
   d. Calculate distance of all points from all centers and find closest cluster
   e. Send and receive points (internal and external transfers)

   < re_clusterify(), bcast_and_check_means() >

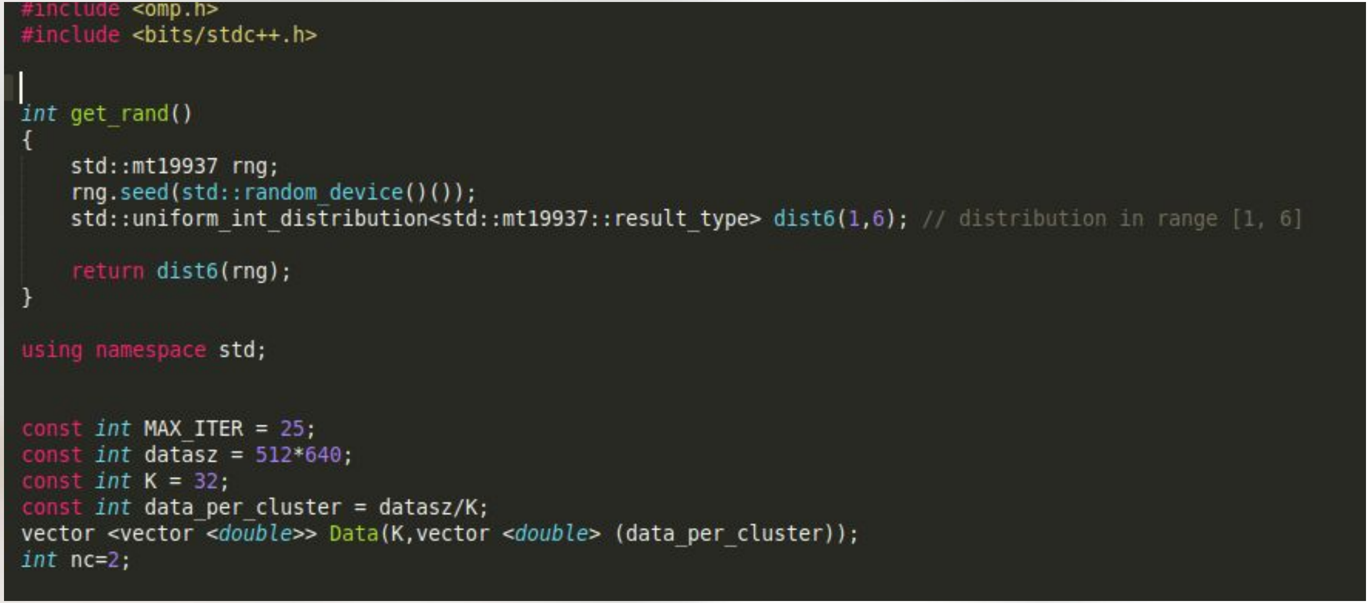
3. Repeat until Convergence (stopping condition)
   - No internal/external transfers i.e Centers remain constant

   < check_stop_condition() >

Image segmentation with parallel K means

4）Parallel OpenMP Model

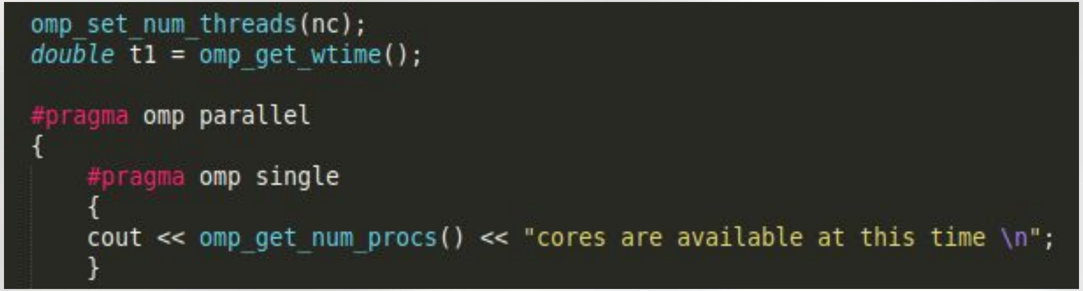
OpenMP MODEL PARAMETERS



```
#include <omp.h>
#include <bits/stdc++.h>


int get_rand()
{
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<std::mt19937::result_type> dist6(1,6); // distribution in range [1, 6]

    return dist6(rng);
}

using namespace std;


const int MAX_ITER = 25;
const int datasz = 512*640;
const int K = 32;
const int data_per_cluster = datasz/K;
vector <vector <double>> Data(K,vector <double> (data_per_cluster));
int nc=2;
```

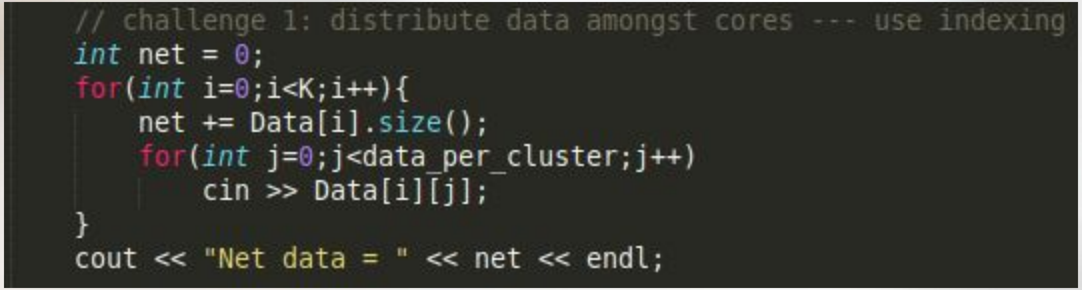
Repository: https://github.com/thezodiac1994/Parallel-Alogrithms

OpenMP MODEL FLOW



```
omp_set_num_threads(nc);
double t1 = omp_get_wtime();

#pragma omp parallel
{
    #pragma omp single
    {
    cout << omp_get_num_procs() << "cores are available at this time \n";
    }
```

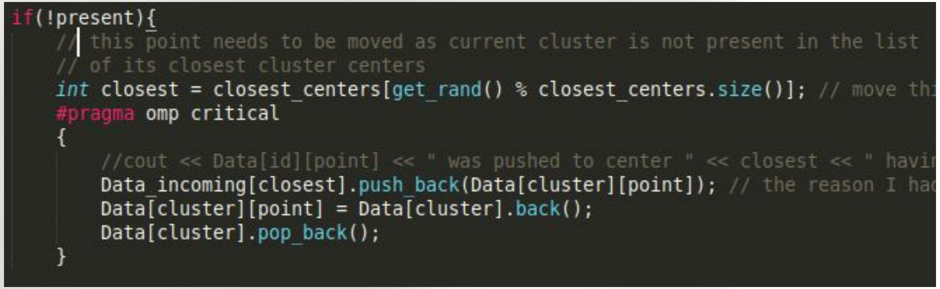
➔ Request for nc number of threads/cores and MAKE SURE we actually get them on the allotted machine.



```
    // challenge 1: distribute data amongst cores --- use indexing
    int net = 0;
    for(int i=0;i<K;i++){
        net += Data[i].size();
        for(int j=0;j<data_per_cluster;j++)
            cin >> Data[i][j];
    }
    cout << "Net data = " << net << endl;
```

➔ Allot equal (data/k) points to each cluster initially This Data is global and visible to all. However, to Handle accessing, we use indexing of Data array. Each core takes care of (k/nc) clusters depending on thread id.

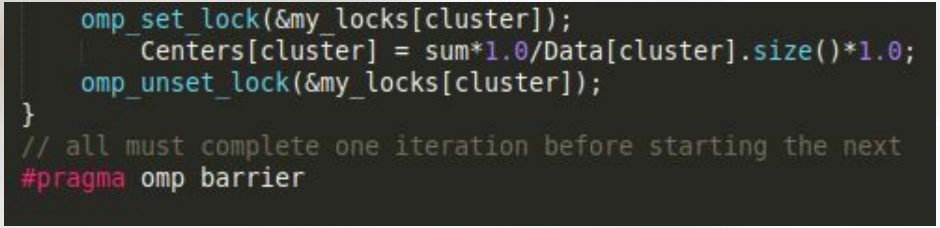
OpenMP MODEL FLOW



```
if(!present){
    // this point needs to be moved as current cluster is not present in the list
    // of its closest cluster centers
    int closest = closest_centers[get_rand() % closest_centers.size()]; // move thi
    #pragma omp critical
    {
        //cout << Data[id][point] << " was pushed to center " << closest << " havi
        Data_incoming[closest].push_back(Data[cluster][point]); // the reason I ha
        Data[cluster][point] = Data[cluster].back();
        Data[cluster].pop_back();
    }
```

➔ For each point, the respective thread will calculate the closest cluster and add the point to respective closest clusters incoming list of points.

Since the list of incoming points is global and shared, it has to be done in the critical section.

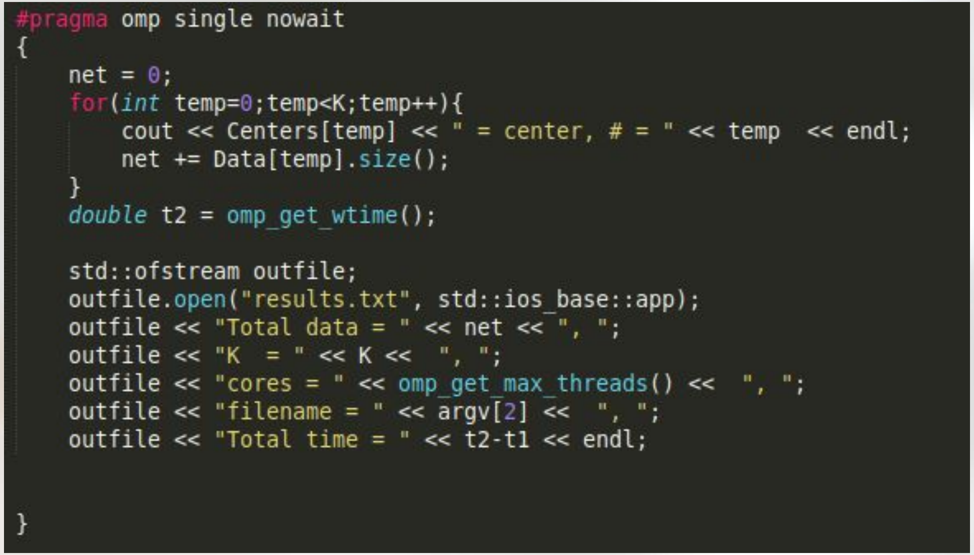


```
    omp_set_lock(&my_locks[cluster]);
        Centers[cluster] = sum*1.0/Data[cluster].size()*1.0;
    omp_unset_lock(&my_locks[cluster]);
}
// all must complete one iteration before starting the next
#pragma omp barrier
```

➔ Once points are rotated, the centers need to be updated.

The centers can be updated in the critical section but I made use of locks to make it more efficient since we only need to lock one value/index at a time.

OpenMP MODEL FLOW



```
#pragma omp single nowait
{
    net = 0;
    for(int temp=0;temp<K;temp++){
        cout << Centers[temp] << " = center, # = " << temp  << endl;
        net += Data[temp].size();
    }
    double t2 = omp_get_wtime();

    std::ofstream outfile;
    outfile.open("results.txt", std::ios_base::app);
    outfile << "Total data = " << net << ", ";
    outfile << "K  = " << K <<  ", ";
    outfile << "cores = " << omp_get_max_threads() <<  ", ";
    outfile << "filename = " << argv[2] <<  ", ";
    outfile << "Total time = " << t2-t1 << endl;


}
```

➔ Use just one core (omp single) to asynchronously (omp nowait) to do final calculations such as validation checks, time calculations and writing to file.

5) RESULTS

a) Constant Input Size

(512 x 640 = ~0.3 m data points)

i) MPI: Nodes vs Time

Nodes	Time
1	2.502
2	2.190
4	1.555
8	1.472
16	1.565
32	1.652

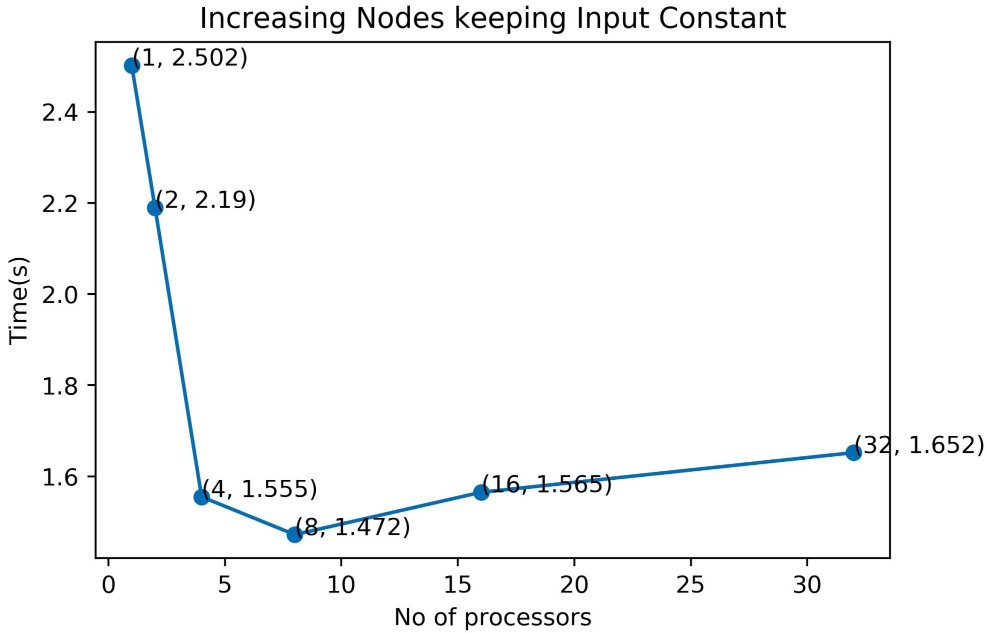
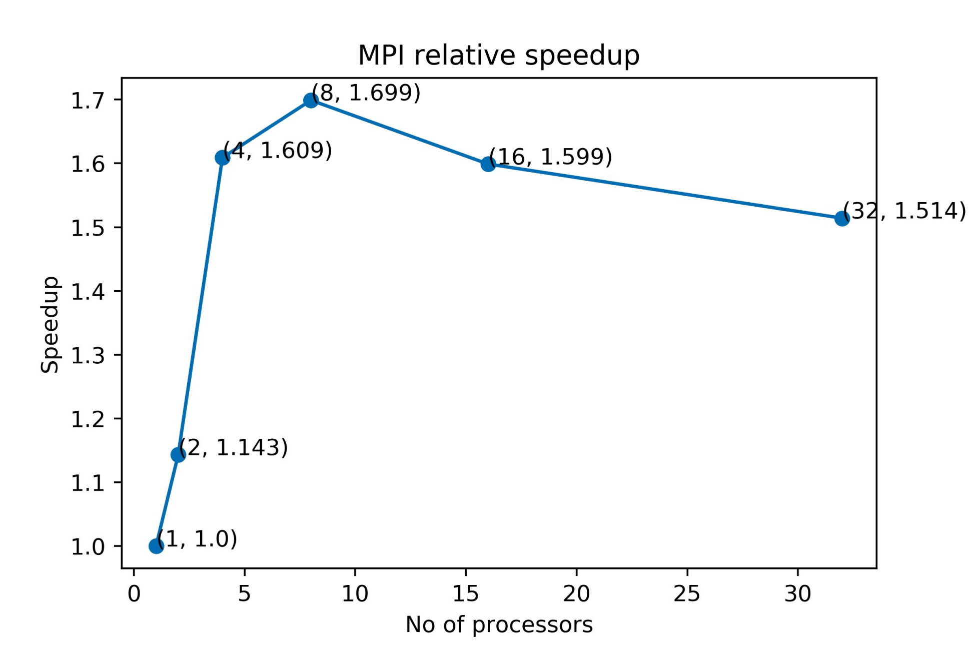


Image segmentation with parallel K means

ii) MPI: Speedup

Nodes	Time
1	1.000
2	1.143
4	1.609
8	1.699
16	1.599
32	1.514



Relative to performance of single MPI node of same model

iii) OpenMP: Cores vs Time

Cores	Time
1	4.075
2	3.565
4	2.812
8	2.188
16	1.548
32	1.590

(1 thread per core)

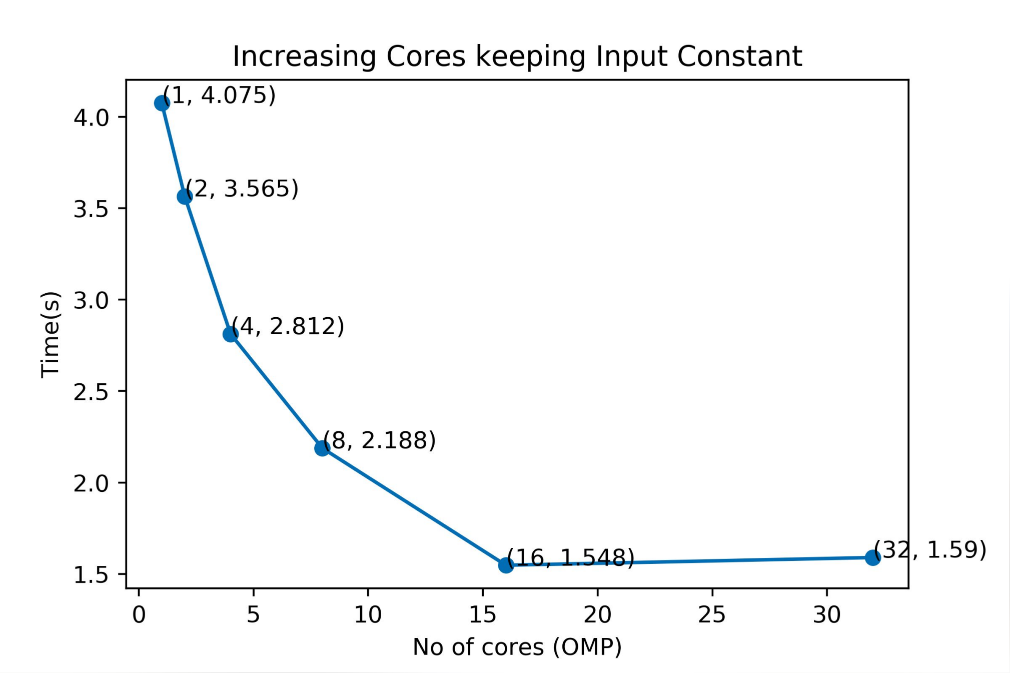
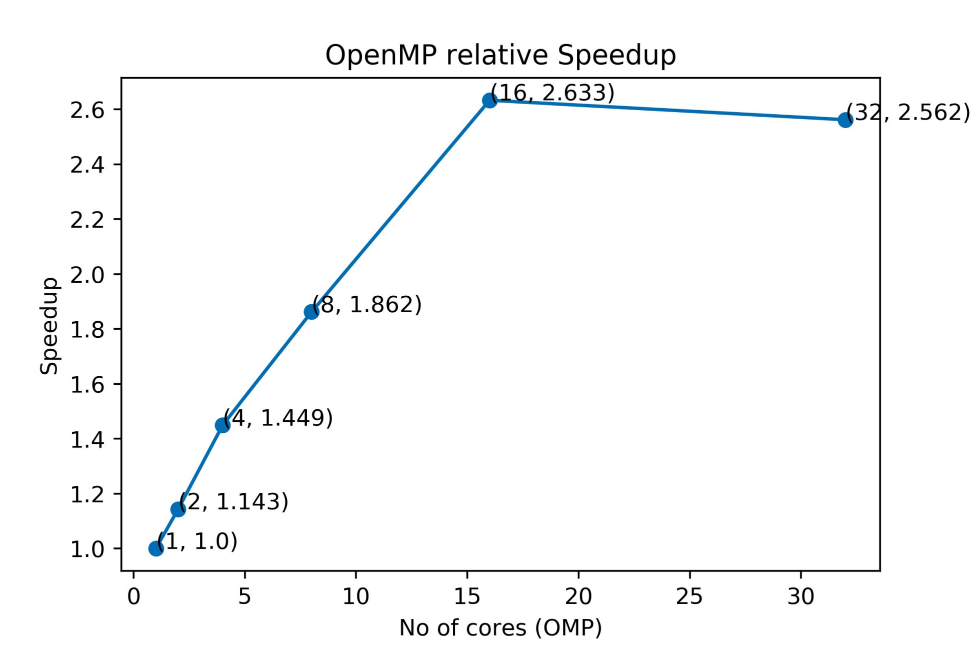


Image segmentation with parallel K means

iv) OpenMP: Speedup

Cores	Time
1	1.000
2	1.143
4	1.449
8	1.862
16	2.633
32	2.562

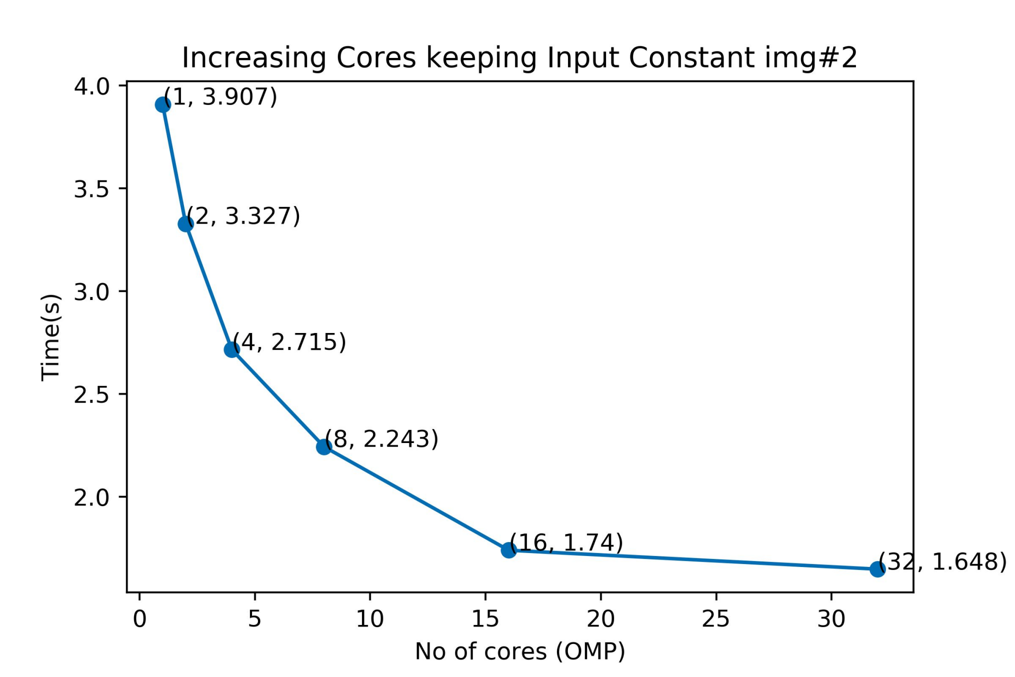
(1 thread per core)



v) OpenMP: Cores vs Time - img2

Cores	Time
1	3.907
2	3.327
4	2.715
8	2.243
16	1.740
32	1.648

(1 thread per core)



b) Scaling Input with processors/cores (~0.3m to ~10m data points)

i) MPI: Scaling input with processors

Nodes	Data (c = 512*768 =~0.3 mil)	Time
1	1c	2.65
2	2c	4.41
4	4c	6.41
8	8c	11.50
16	16c	24.66
32	32c	51.83

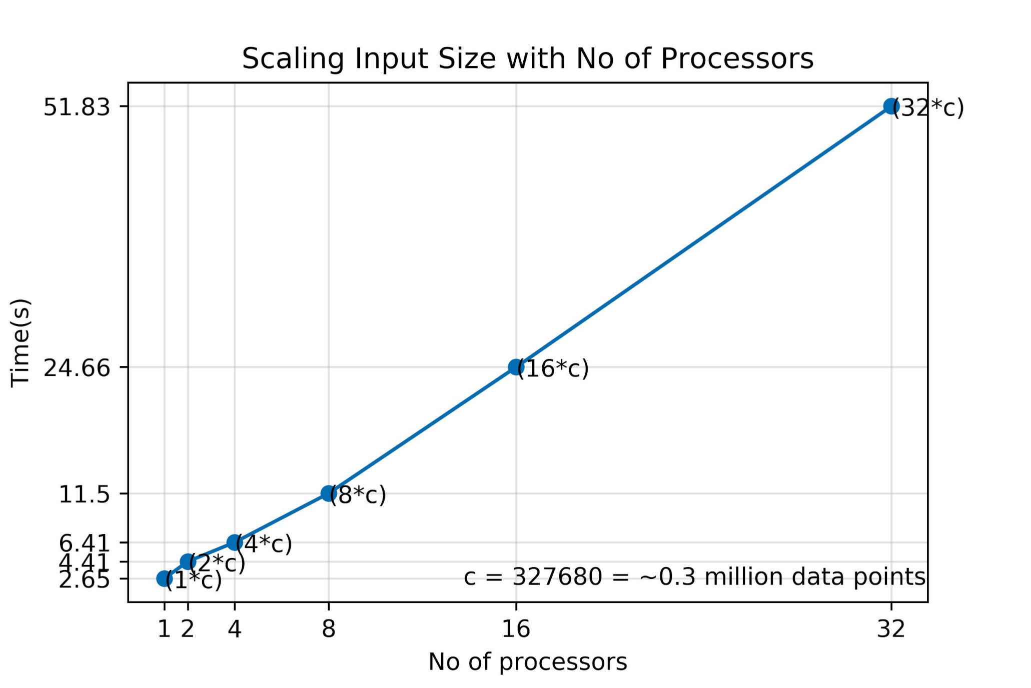


Image segmentation with parallel K means

ii) OpenMP: Scaling input with cores

Cores	Data (c = 512*768 = 0.3 mil)	Time
1	1c	4.17
2	2c	6.81
4	4c	10.34
8	8c	16.29
16	16c	37.22
32	32c	54.75

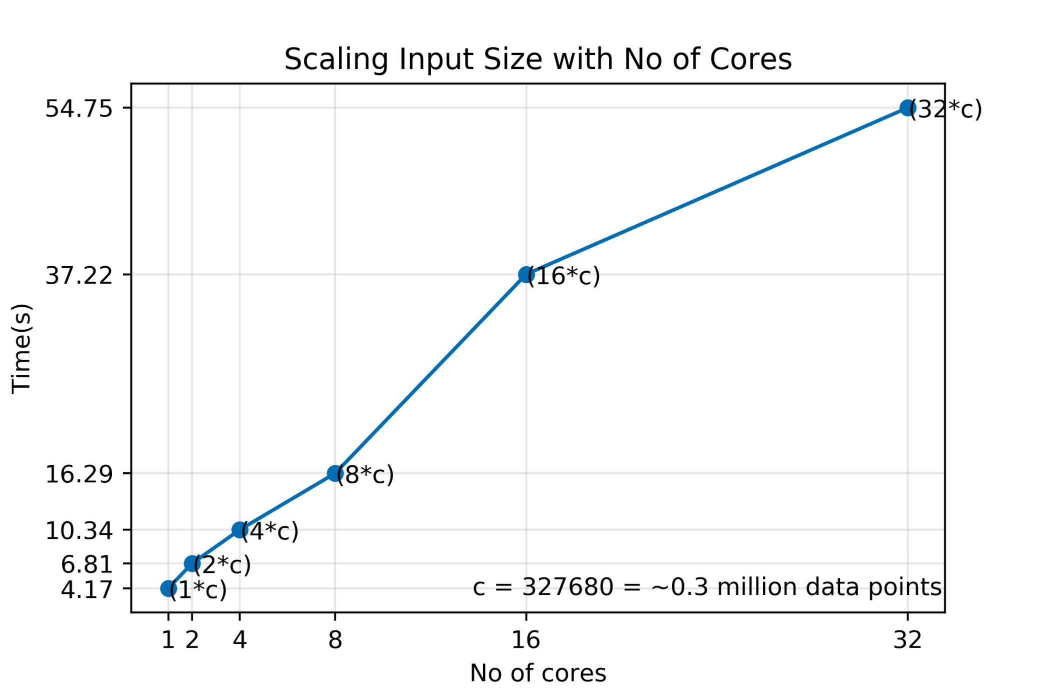


Image segmentation with parallel K means

b) Comparing MPI with OpenMP (leaving hardware specifics)

i) constant input size: cores/nodes vs time

(*separate run)

Nodes/ Cores log2	Time MPI	Time OMP
0	2.502	4.075
2	2.190	3.565
4	1.555	2.812
8	1.472	2.188
16	1.565	1.558
32	1.652	1.590

Inpsize = ~0.3 mil (512x640)

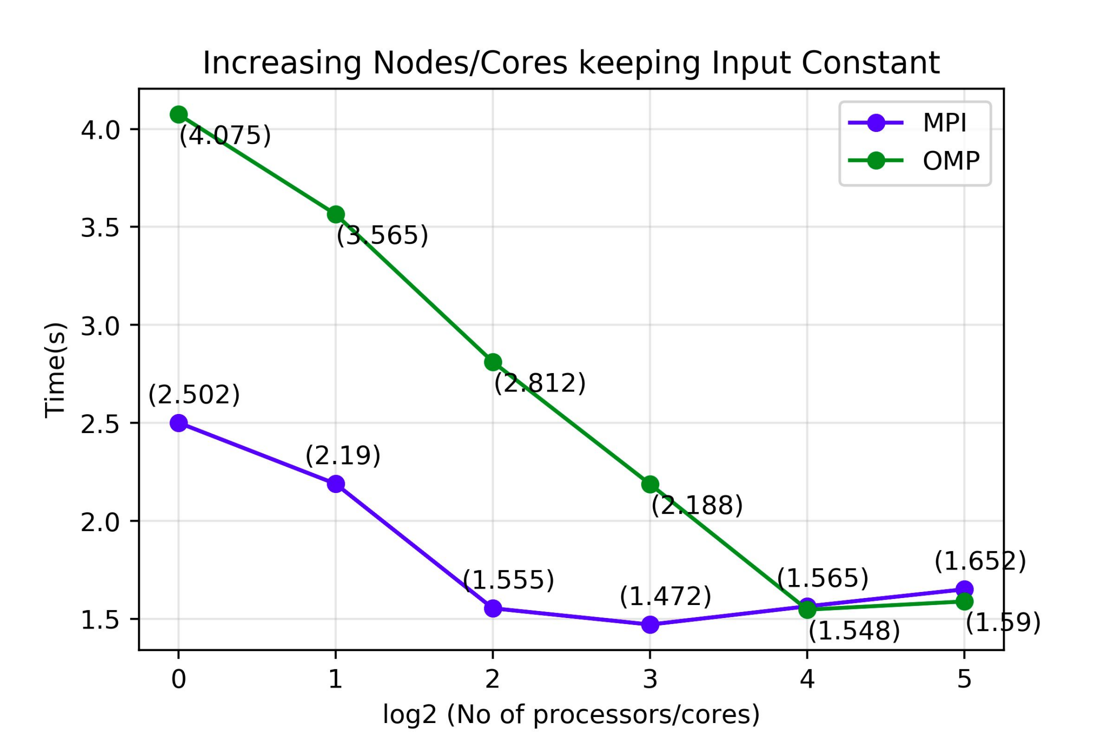


Image segmentation with parallel K means

ii) MPI vs OpenMP

scaling input size (*separate run)

Nodes/ Cores (log2)	Data (c=~0.3mil)	Time MPI	Time OMP
0	1c	2.642	4.170
1	2c	4.413	6.807
2	4c	6.413	10.340
3	8c	11.499	16.291
4	16c	24.655	37.216
5	32c	51.832	54.751

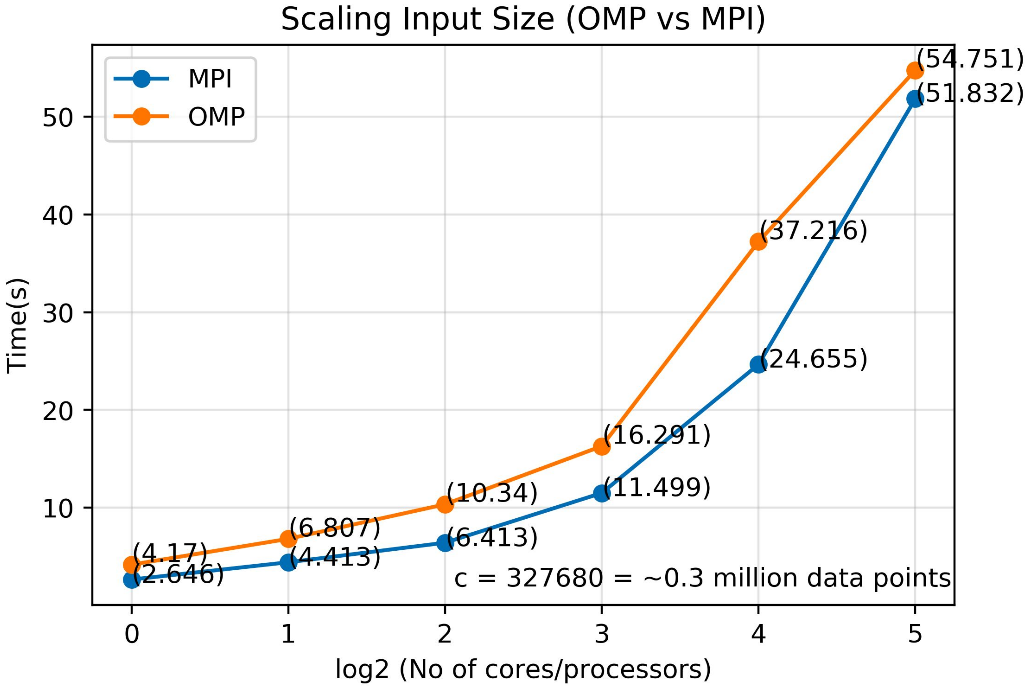
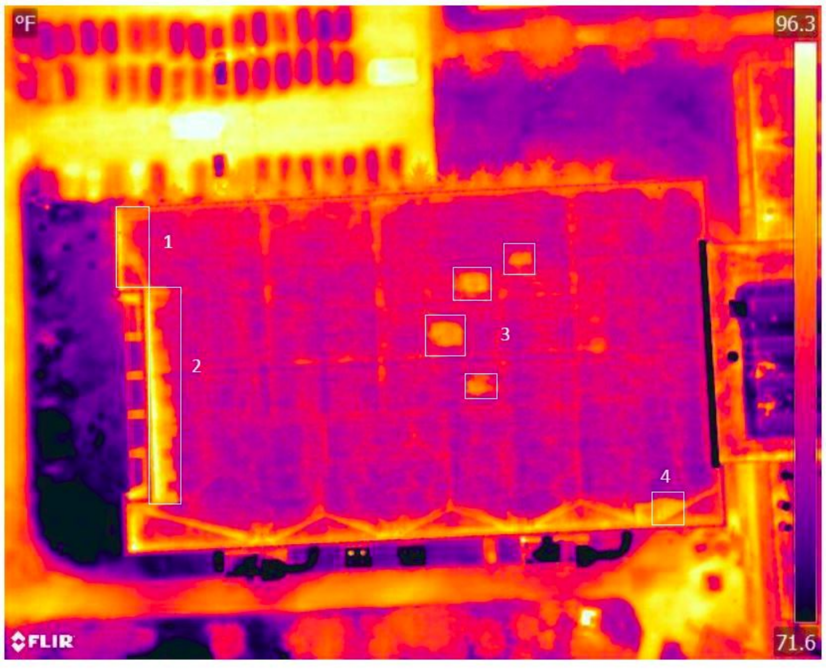
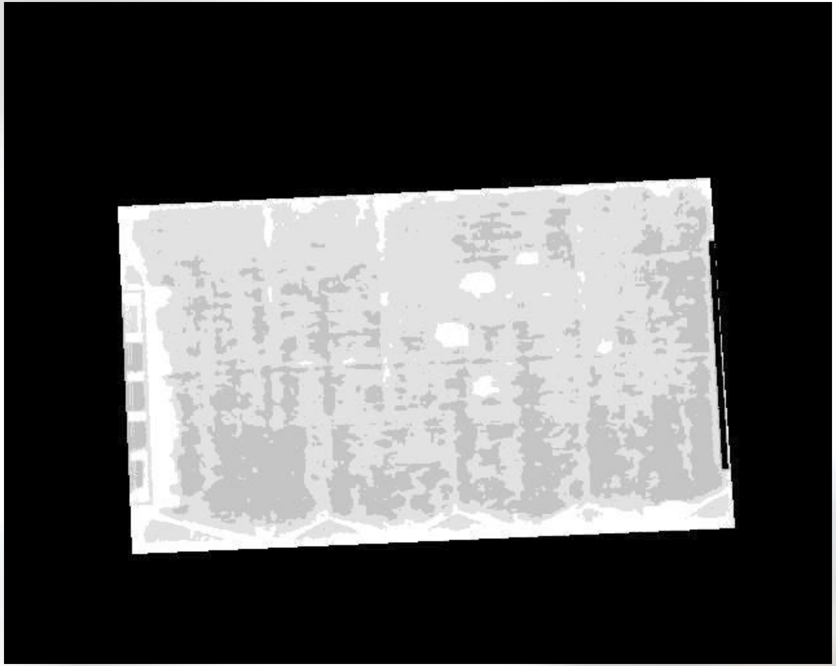


Image segmentation with parallel K means

c) A visualization of OpenMP output



Expected



K means (OMP)

➔ The highest intensity cluster closely represents the faults along with some false positives

Image segmentation with parallel K means

6) INFERENCES

Inferences

1) For my model, 8 nodes for MPI and 16 cores are ideal from performance perspective

2) 32 cores beats 32 nodes by a small margin ! (Hardware specifics not known)

3) OpenMP is far easier to code than MPI (150 lines vs 600 lines of code).

4) Problem is fairly parallelizable and scalable (advisable under 32 nodes/cores). I needed close to 1 minute on python for 25 iterations on a single image as compared to <2 seconds on MPI and OpenMp with 16 cores/nodes.

7) REFERENCES

REFERENCES

1) Algorithms Sequential & Parallel: A Unified Approach
(Dr. Russ Miller, Dr.Laurence Boxer)

2) https://ubccr.freshdesk.com/support/solutions/articles/13000026245-tutorials-and-training-documents
(Dr. Matthew Jones)

3) A Parallel K-Means Clustering Algorithm with MPI
(Jing Zhang, Gongqing Wu, Xuegang Hu, Shiying Li, Shuilong Hao)

4) https://www.buffalo.edu/ccr/support/ccr-help.html
(UB CCR help)

5) Stackoverflow (for general MPI questions)

QUESTIONS ?