NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL
DEPARTMENT OF INFORMATION TECHNOLOGY
IT 301 Parallel Computing LAB 10

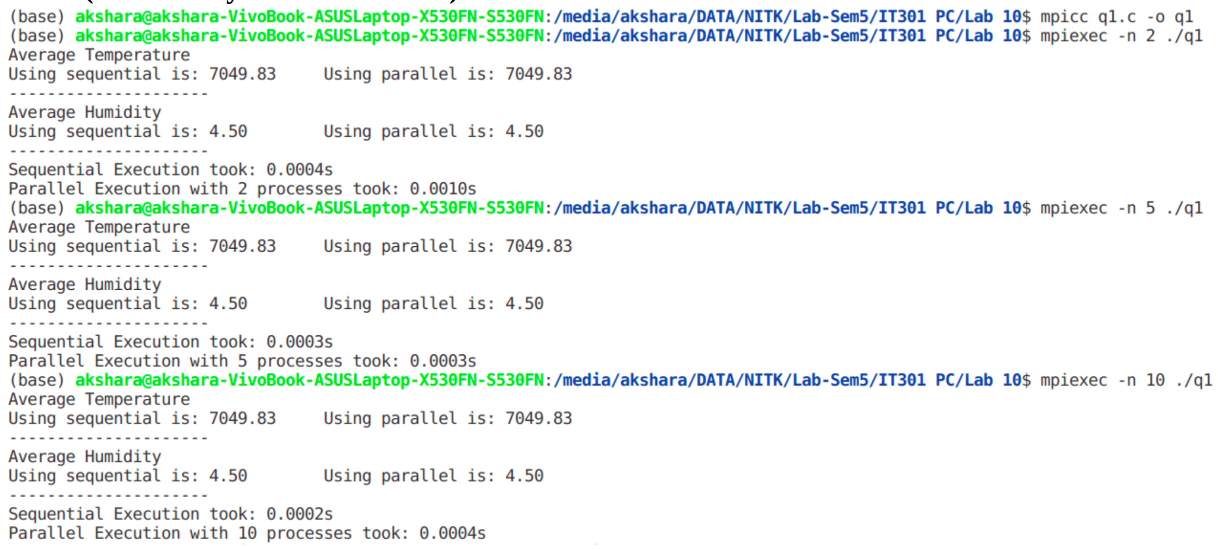
P Akshara - 181IT132

1. In a smart agriculture system in a large area like a state, sensors are deployed to collect temperature and humidity. The sensed information are stored in a server in the cloud. A query on calculating the average temperature and average humidity of the complete state needs the processing of 10 lakh data elements. Write a parallel program using MPI in which N number of processes run in parallel to calculate the average of 10 lakh elements stored in an array, in order to improve response time. Compare the execution time with sequential code.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include <assert.h>

const int N = 1e5;

/*
N: No. of array elements
temp_arr: Temperature array in server
humid_arr: Humidity array in server
start_val: First array index for a process
end_val: Last array index for a process
Every processor finds local sum for array elements with index between start_val & end_val
local_sum: Stores local sum calculated by each process
global_sum: Obtained through reduction -> Stores sum of all array elements
seq_sum: Sum of all elements obtained sequentially
*/

int main(int argc, char *argv[])
{
int rank, size, n, temp_arr[N], humid_arr[N], start_val, end_val;
int temp_seq_sum, temp_local_sum, temp_global_sum, humid_seq_sum, humid_local_sum, humid_global_sum;
double start, fin;
temp_seq_sum = temp_local_sum = temp_global_sum = humid_seq_sum = humid_local_sum =
humid_global_sum = 0;
// Initializing array elements
srand(rank+1);
for (int i = 0; i < N; i++)
{
temp_arr[i] = i;
humid_arr[i] = i%10;
}
// Sequential execution
clock_t begin = clock();
for (int i = 0; i < N; i++)
{
temp_seq_sum = temp_seq_sum + temp_arr[i];
humid_seq_sum = humid_seq_sum + humid_arr[i];
}
clock_t end = clock();
// MPI initialization
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Starting clock for recording time
start = MPI_Wtime();
start_val = N * rank / size + 1;
end_val = N * (rank + 1) / size;
n = N / size;
// Calculation of local sum at processor level
for (int i = start_val; i <= end_val; i++)
{ temp_local_sum = temp_local_sum + temp_arr[i];
humid_local_sum = humid_local_sum + humid_arr[i];
}
// Reduction to get global sum of array elements
MPI_Reduce(&temp_local_sum, &temp_global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&humid_local_sum, &humid_global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
// Barrier to synchronize all processes before calculating time
MPI_Barrier(MPI_COMM_WORLD);
fin = MPI_Wtime();

// Display statistics at root
if (rank == 0)
{
printf("Average Temperature\n");
printf("Using sequential is: %.2f\t Using parallel is: %.2f", temp_seq_sum / (1. * N), temp_global_sum / (1. * N));
printf("\n---------------------\n");
printf("Average Humidity\n");
printf("Using sequential is: %.2f\t Using parallel is: %.2f", humid_seq_sum / (1. * N), humid_global_sum / (1. * N));
printf("\n---------------------\n");
printf("Sequential Execution took: %.4fs\n", (double)(end - begin) / CLOCKS_PER_SEC);
printf("Parallel Execution with %d processes took: %.4fs\n", size, fin-start);
}

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}
```

Output

For N (no. of array elements = 10^5)



```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpicc q1.c -o q1
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpiexec -n 2 ./q1
Average Temperature
Using sequential is: 7049.83     Using parallel is: 7049.83
---------------------
Average Humidity
Using sequential is: 4.50        Using parallel is: 4.50
---------------------
Sequential Execution took: 0.0004s
Parallel Execution with 2 processes took: 0.0010s
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpiexec -n 5 ./q1
Average Temperature
Using sequential is: 7049.83     Using parallel is: 7049.83
---------------------
Average Humidity
Using sequential is: 4.50        Using parallel is: 4.50
---------------------
Sequential Execution took: 0.0003s
Parallel Execution with 5 processes took: 0.0003s
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpiexec -n 10 ./q1
Average Temperature
Using sequential is: 7049.83     Using parallel is: 7049.83
---------------------
Average Humidity
Using sequential is: 4.50        Using parallel is: 4.50
---------------------
Sequential Execution took: 0.0002s
Parallel Execution with 10 processes took: 0.0004s
```

Analysis

Using 10^5 elements for both temperature and humidity arrays, it is observed that using around 5 processors with the chunksize used scales well. Beyond 5, time taken is more in parallel version due to process spawning and communication overheads.

All routines used have been explained with comments in the code

2. Consider random deployment of sensor nodes in field to sense the environment. The nodes are deployed randomly and the position of each sensor node is sent to centralised server. The server would like to cluster these nodes. Use K-means algorithm to cluster the nodes. Write an MPI program to cluster the sensor nodes and compare the result with sequential and OPENMP approach.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>

// Creates an array of random floats. Each number has a value from 0 - 1
float *create_rand_nums(const int num_elements)
{
float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
assert(rand_nums != NULL);
for (int i = 0; i < num_elements; i++)
{
rand_nums[i] = (rand() / (float)RAND_MAX);
}
return rand_nums;
}

float distance2(const float *v1, const float *v2, const int d)
{
float dist = 0.0;
for (int i = 0; i < d; i++)
{
float diff = v1[i] - v2[i];
dist += diff * diff;
}
return dist;
}

// Assign a site to the correct cluster by computing its distances to each cluster centroid.
int assign_site(const float *site, float *centroids, const int k, const int d)
{
int best_cluster = 0;
float best_dist = distance2(site, centroids, d);
float *centroid = centroids + d;
for (int c = 1; c < k; c++, centroid += d)
{
float dist = distance2(site, centroid, d);
if (dist < best_dist)
{
best_cluster = c;
best_dist = dist;
}
}
return best_cluster;
}

void add_site(const float *site, float *sum, const int d)
{
for (int i = 0; i < d; i++)
sum[i] += site[i];
}

void print_centroids(float *centroids, const int k, const int d)
{
float *p = centroids;
```

```
printf("Centroids:\n");
for (int i = 0; i < k; i++)
{
for (int j = 0; j < d; j++, p++)
{
printf("%f ", *p);
}
printf("\n");
}
}

int main(int argc, char **argv)
{
int sites_per_proc, d, rank, nprocs, k;
sites_per_proc = 2000;
d = 2;
if (argc != 2)
{
fprintf(stderr,"Incorrect args\n");
exit(1);
}
k = atoi(argv[1]);

srand(31359);
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

float *sites;
assert(sites = malloc(sites_per_proc * d * sizeof(float)));
float *sums;
assert(sums = malloc(k * d * sizeof(float)));
// The number of sites assigned to each cluster by this process. k integers.
int *counts;
assert(counts = malloc(k * sizeof(int)));
// The current centroids against which sites are being compared.
float *centroids;
assert(centroids = malloc(k * d * sizeof(float)));
// The cluster assignments for each site.
int *labels;
assert(labels = malloc(sites_per_proc * sizeof(int)));

// All the sites for all the processes.
float *all_sites = NULL;
// Sum of sites assigned to each cluster by all processes.
float *grand_sums = NULL;
// Number of sites assigned to each cluster by all processes.
int *grand_counts = NULL;
int *all_labels;
if (rank == 0)
{
printf("Starting random initialization..\n");
all_sites = create_rand_nums(d * sites_per_proc * nprocs);
for (int i = 0; i < k * d; i++)
{
centroids[i] = all_sites[i];
}
assert(grand_sums = malloc(k * d * sizeof(float)));
assert(grand_counts = malloc(k * sizeof(int)));
assert(all_labels = malloc(nprocs * sites_per_proc * sizeof(int)));
printf("Nodes initialized \n");
}
double start_time = MPI_Wtime();
```

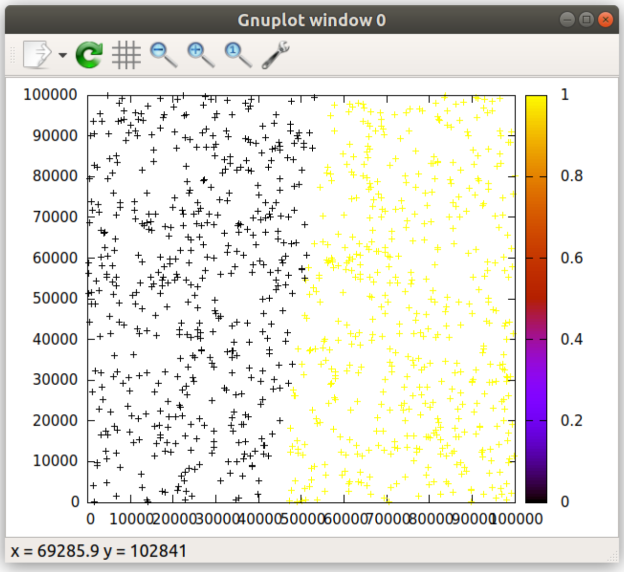
```
// Root sends each process its share of sites.
MPI_Scatter(all_sites, d * sites_per_proc, MPI_FLOAT, sites, d * sites_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

float norm = 1.0;
// Will tell if centroids have changed
while (norm > 0.00001)
{ // Broadcast the current cluster centroids to all processes.
MPI_Bcast(centroids, k * d, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Each process reinitializes its cluster accumulators.
for (int i = 0; i < k * d; i++)
sums[i] = 0.0;
for (int i = 0; i < k; i++)
counts[i] = 0;

// Find the closest centroid to each site and assign to cluster.
float *site = sites;
for (int i = 0; i < sites_per_proc; i++, site += d)
{
int cluster = assign_site(site, centroids, k, d);
// Record the assignment of the site to the cluster.
counts[cluster]++;
add_site(site, &sums[cluster * d], d);
}

// Gather and sum at root all cluster sums for individual processes.
MPI_Reduce(sums, grand_sums, k * d, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(counts, grand_counts, k, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
{
// Root process computes new centroids by dividing sums per cluster
// by count per cluster.
for (int i = 0; i < k; i++)
{
for (int j = 0; j < d; j++)
{
int dij = d * i + j;
grand_sums[dij] /= grand_counts[i];
}
}
norm = distance2(grand_sums, centroids, d * k);
printf("norm: %f\n", norm);
// Copy new centroids from grand_sums into centroids.
for (int i = 0; i < k * d; i++)
{
centroids[i] = grand_sums[i];
}
print_centroids(centroids, k, d);
}
// Broadcast the norm. to all processes
MPI_Bcast(&norm, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
}
// Centroids are fixed, so compute a final label for each site.
float *site = sites;
for (int i = 0; i < sites_per_proc; i++, site += d)
{
labels[i] = assign_site(site, centroids, k, d);
}
// Gather all labels into root process.
MPI_Gather(labels, sites_per_proc, MPI_INT, all_labels, sites_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
// Root can print out all sites and labels.
MPI_Barrier(MPI_COMM_WORLD);
double end_time = MPI_Wtime() - start_time;
```

```
if (rank == 0)
{
float *site = all_sites;
printf("Plotting nodes...\n");
printf("Time: %fs\n",end_time);
}
MPI_Finalize();
}
```

Output

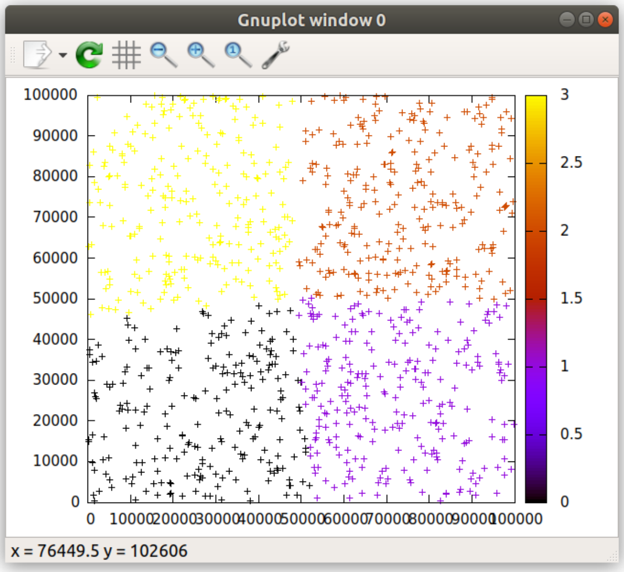
Taking 1000 nodes,
a) 2 clusters

```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpicc q21.c
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 10$ mpiexec -n 5 ./a.out 2
Starting random initialization..
Nodes initialized
norm: 0.190889
Centroids:
0.577627 0.181351
0.457397 0.662449
norm: 0.003480
Centroids:
0.550071 0.213315
0.461700 0.703439
norm: 0.001148
Centroids:
0.532949 0.230038
0.469994 0.725934
norm: 0.000337
Centroids:
0.523132 0.238331
0.476880 0.737103
norm: 0.000123
Centroids:
0.516458 0.242335
0.482575 0.742564
norm: 0.000057
Centroids:
0.511835 0.244895
0.486833 0.745909
norm: 0.000011
Centroids:
0.509745 0.245764
0.488855 0.747047
norm: 0.000006
Centroids:
0.508168 0.246335
0.490410 0.747781
Plotting nodes...
Time: 0.002962s
```

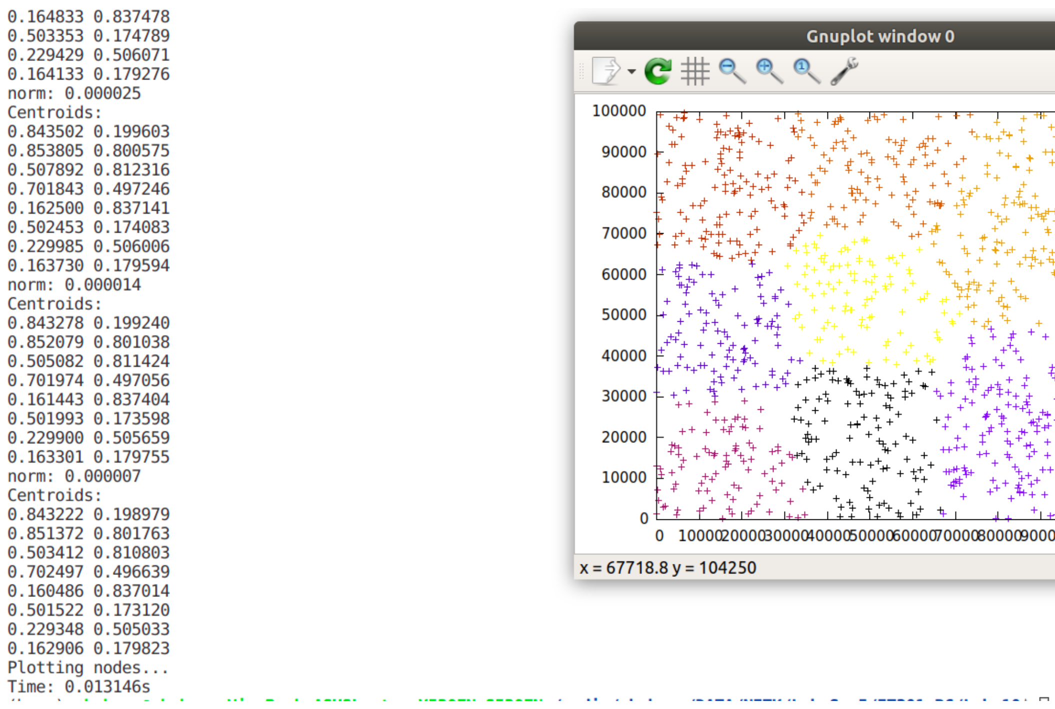


b) 4 clusters

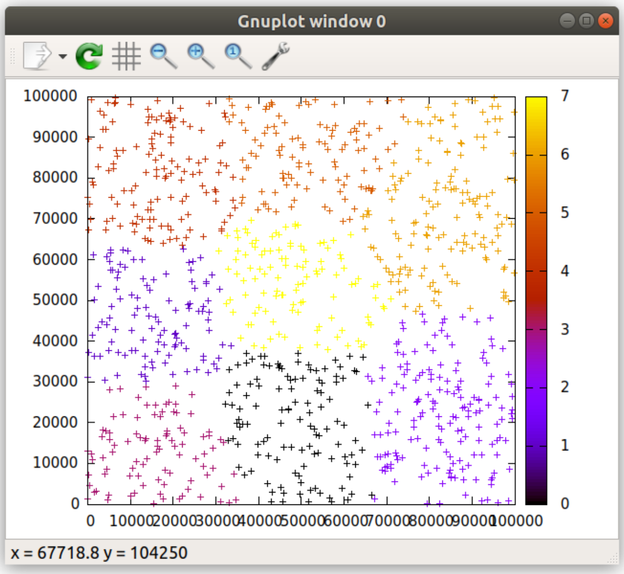
```
[illegible]
Centroids:
0.738899 0.231681
0.755843 0.730677
0.259811 0.772670
0.242324 0.270018
norm: 0.000202
Centroids:
0.742634 0.236893
0.751534 0.736670
0.254881 0.766799
0.245192 0.263773
norm: 0.000086
Centroids:
0.743491 0.239117
0.749152 0.739544
0.251373 0.761530
0.246558 0.258794
norm: 0.000028
Centroids:
0.744470 0.240151
0.747419 0.740701
0.249413 0.758638
0.247544 0.255877
norm: 0.000015
Centroids:
0.745257 0.241755
0.746365 0.742089
0.248563 0.756765
0.248011 0.253707
norm: 0.000005
Centroids:
0.745565 0.242679
0.745777 0.742896
0.247949 0.755658
0.248280 0.252454
Plotting nodes...
Time: 0.005035s
```



c) 8 clusters



```
0.164833 0.837478
0.503353 0.174789
0.229429 0.506071
0.164133 0.179276
norm: 0.000025
Centroids:
0.843502 0.199603
0.853805 0.800575
0.507892 0.812316
0.701843 0.497246
0.162500 0.837141
0.502453 0.174083
0.229985 0.506006
0.163730 0.179594
norm: 0.000014
Centroids:
0.843278 0.199240
0.852079 0.801038
0.505082 0.811424
0.701974 0.497056
0.161443 0.837404
0.501993 0.173598
0.229900 0.505659
0.163301 0.179755
norm: 0.000007
Centroids:
0.843222 0.198979
0.851372 0.801763
0.503412 0.810803
0.702497 0.496639
0.160486 0.837014
0.501522 0.173120
0.229348 0.505033
0.162906 0.179823
Plotting nodes...
Time: 0.013146s
```



Analysis

Sl. No.	No. of clusters	Toal Sequential Time (s)	Total Parallel Time (s)
1	2	0.0037	0.0029
2	4	0.0061	0.0050
3	8	0.0233	0.0131

Table: Comparing Sequential and Parallel version

As seen in the above table, the parallel version is much faster than the sequential one for all cluster values taken.
The following parts of the sequential code have been parallelized which leads to the improvement:
1. MPI_Scatter to scatter the points among 5 processes gave best results. Each process works on 200 nodes.
2. Main computational improvement is through parallelizing the Euclidean distance computation in each iteration. This is achieved using MPI_Bcast, MPI_Reduce, MPI_Gather.
3. Synchronization among iterations achieved through MPI_Barrier.

All routines used have been explained with comments in the code