Universidade de São Paulo
Instituto de Física de São Carlos

High Performance Programming-Parallel MPI

Éverton Luís Mendes da Silva (10728171)

Contents

1 Resume

Currently, there are several applications in which, given a matrix A, the eigenvalues and eigenvectors are relevant information to solve the problem. With this in mind, there are several methods to find these values, however, most of them do not take into account the limitation of current computers in dealing with large memories and processing speed of current CPUs(approximately 10^8 iterations per second).As an example, if we have a problem with a large number of variables(N), algorithms that have a complexity of $N^2(N \rightarrow \infty, \Theta(N^2))$ are unfeasible for solution due to the high processing time. Therefore, in this project, the representation of a graph as a weighted adjacency matrix was discarded, thus the multiplication of a matrix by a vector was implemented with an weighted adjacency list(weighted linked list). Finally, the power iteration method for matrix multiplication was implemented in two ways: sequential and parallel.

2 Theoretical Introduction

The multiplication of a matrix by a vector can be done in two ways, one with matrix representation and the other with an adjacency list, as below:

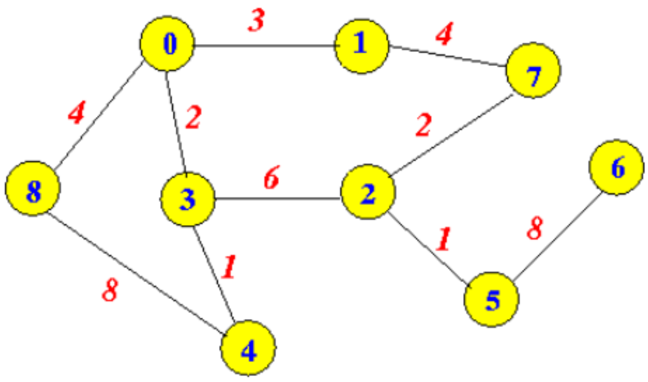

Figure 1: Graph

	0	1	2	3	4	5	6	7	8
0	0	3	0	2	0	0	0	0	4
1	3	0	0	0	0	0	0	4	0
2	0	0	0	6	0	1	0	2	0
3	2	0	6	0	1	0	0	0	0
4	0	0	0	1	0	0	0	0	8
5	0	0	1	0	0	0	8	0	0
6	0	0	0	0	0	8	0	0	0
7	0	4	2	0	0	0	0	0	0
8	4	0	0	0	8	0	0	0	0

Figure 2: Adjacency Matrix

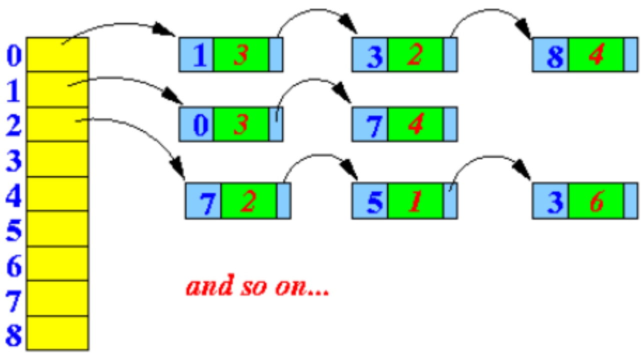

Figure 3: Adjacency List

3 Description of files

This project was divided into four main files that will be presented below. The first consists of a file, named 'AdjList', which reads a file in Pajek format and creates an adjacency list with the respective values. In addition, we have two more files to perform the power iteration method, one sequential and the two other parallels.

3.1 Code Linked List

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

struct Graph
{
    struct Node *head[1];
};

struct Edge
{
    int i, j;
    double weight;
};

struct Node
{
    int dest;
    double weight;
    struct Node *next;
};

struct Graph *createGraph(struct Edge edges[], int n, int
  n_vertexs)
{
    struct Graph *graph = (struct Graph *)malloc(sizeof(
  struct Graph) + sizeof(struct Node) * (n_vertexs - 1));

    for (int i = 0; i < n_vertexs; i++)
    {
        graph->head[i] = NULL;
    }

    for (int i = 0; i < n; i++)
    {
        int src = edges[i].i;
        int dest = edges[i].j;
        double weight = edges[i].weight;

        struct Node *newNode = (struct Node *)malloc(sizeof(
  struct Node));
        newNode->dest = dest;
        newNode->weight = weight;

        newNode->next = graph->head[src];

        graph->head[src] = newNode;
    }
    return graph;
}
```

```
void printGraph(struct Graph *graph, int n_vertexs)
{
    int i;
    for (i = 0; i < n_vertexs; i++)
    {
        struct Node *ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("%d    > %d ", i, ptr->dest);
            ptr = ptr->next;
        }

        printf("\n");
    }
}

void returnNumberNeighbors(struct Graph *graph, int n_vertexs, int *n_neighbors)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        struct Node *ptr = graph->head[i];
        while (ptr != NULL)
        {
            ptr = ptr->next;
            n_neighbors[i]+=1;
        }
    }
}

void neighborsWeight(struct Graph *graph, int vertex, int n_neighbors, double *weights)
{

    struct Node *ptr = graph->head[vertex];
    for (int i=0; i<n_neighbors; ++i)
    {
        weights[i]=ptr->weight;
        ptr = ptr->next;

    }
}

struct File_data
{
    int n_vertexs;
    struct Graph *graph;
};
```

```
struct File_data *ReadPajek(char *filename)
{
    FILE *fp;
    fp = fopen(filename, "r");

    int n_vertexs, n_edges;
    int i_element, j_element;
    double weight;

    if (fscanf(fp, "%d", &n_vertexs))
    {
    }
    if (fscanf(fp, "%d ", &n_edges))
    {
    }

    struct Edge *Edges = malloc(2 * n_edges * sizeof(struct Edge));

    int count_equal_ij = 0;
    for (int edge = 0; edge < n_edges; edge++)
    {
        if (fscanf(fp, "%d %d %lf", &i_element, &j_element, &weight))
        {
        }

        if (i_element != j_element)
        {
            Edges[2 * edge - count_equal_ij].i = i_element;
            Edges[2 * edge - count_equal_ij].j = j_element;
            Edges[2 * edge - count_equal_ij].weight = weight;
            Edges[2 * edge + 1 - count_equal_ij].i = j_element;
            Edges[2 * edge + 1 - count_equal_ij].j = i_element;
            Edges[2 * edge + 1 - count_equal_ij].weight = weight;
        }
        else
        {
            Edges[2 * edge - count_equal_ij].i = i_element;
            Edges[2 * edge - count_equal_ij].j = j_element;
            Edges[2 * edge - count_equal_ij].weight = weight;
            count_equal_ij += 1;
        }
    }

```

```
    fclose(fp);

    struct Graph *graph = createGraph(Edges, 2 * n_edges - count_equal_ij, n_vertexs);

    struct File_data *file_data = malloc(sizeof(int) + sizeof(struct Graph) + sizeof(struct Node) * (n_vertexs - 1));
    file_data->n_vertexs = n_vertexs;
    file_data->graph = graph;

    return file_data;
}
```

Listing 1: Read Pajek file and Linked List Representation

3.2 Code FindEigen - Sequential

```
#include "AdjLisT.c"
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

void read_arguments_or_abort(int argc, char *argv[]);
double normalize_vec(int n_vertexs, double *vector);
void mat_mult_AdjList(struct Graph *graph, double *vector, double *new_vector, int n_vertexs);
void printfvector(double *vector, int n_vertexs);
void cleanVector(double *vector, int n_vertexs);
double mult_pointers(double num1, double num2);
void copy_vec(double *vector, double *new_vector, int n_vertexs);

int main(int argc, char *argv[])
{
    read_arguments_or_abort(argc, argv);
    char *input_filename = argv[1];

    double precision;
    sscanf(argv[2], "%lf", &precision);
    char *output_filename = argv[3];

    struct File_data *file_data = ReadPajek(input_filename);

    double *vec = (double *)malloc(file_data->n_vertexs * sizeof(double));
    double *new_vec = (double *)malloc(file_data->n_vertexs *
```

```c
    sizeof(double));

    for (int i = 0; i < file_data->n_vertexs; i++)
    {
        vec[i] = rand() / (RAND_MAX + 1.0);
        if (rand() / (RAND_MAX + 1.0) >= 0.5)
        {
            vec[i] *= -1;
        }
    }

    int stop_iter = 0;
    double norm_vec, new_norm_vec;

    norm_vec = normalize_vec(file_data->n_vertexs, vec);

    struct timeval t1, t2;
    gettimeofday(&t1, NULL);
    while (stop_iter < 3)
    {
        cleanVector(new_vec, file_data->n_vertexs);
        mat_mult_AdjList(file_data->graph, vec, new_vec, file_data->n_vertexs);
        new_norm_vec = normalize_vec(file_data->n_vertexs, new_vec);
        copy_vec(vec, new_vec, file_data->n_vertexs);

        if (fabs(new_norm_vec - norm_vec) / new_norm_vec < precision)
        {
            stop_iter += 1;
        }
        else
        {
            stop_iter = 0;
        }

        norm_vec = new_norm_vec;
    }
    gettimeofday(&t2, NULL);

    printf("It took %.17lf milliseconds.\n", (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);

    FILE *output_file;
    output_file = fopen(output_filename, "w");
    fprintf(output_file, "%lf\n", new_norm_vec);
    fprintf(output_file, "%d\n", file_data->n_vertexs);

```

```
    for (int i = 0; i < file_data->n_vertexs; i++)
    {
        fprintf(output_file, "%lf\n", vec[i]);
    }

    fclose(output_file);

    cleanVector(new_vec, file_data->n_vertexs);
    mat_mult_AdjList(file_data->graph, vec, new_vec, file_data->n_vertexs);
    new_norm_vec = normalize_vec(file_data->n_vertexs, new_vec);

    if (fabs(new_norm_vec - norm_vec) / new_norm_vec < precision)
    {
        printf("The Method works well\n");
    }
    else
    {
        printf("The Method don't work so well\n");
        printf("method: %.17lf precision: %.17lf\n", fabs(new_norm_vec - norm_vec) / new_norm_vec, precision);
    }

    FILE *time_record_file;
    char timefilename[100] = "time_";
    strcat(timefilename, input_filename);
    printf("%s\n", timefilename);
    time_record_file = fopen(timefilename, "a+");
    fprintf(time_record_file, "%.10lf\n", (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);
    fclose(time_record_file);

    return 0;
}

void read_arguments_or_abort(int argc, char *argv[])
{
    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s <number of elements> <number of arrays>\n",
                argv[0]);
        exit(505);
    }
}

double normalize_vec(int n_vertexs, double *vector)
```

```c
{

    double sum_elements = 0;
    for (int i = 0; i < n_vertexs; i++)
    {
        sum_elements += pow(vector[i], 2);
    }

    for (int i = 0; i < n_vertexs; i++)
    {
        vector[i] /= sqrt(sum_elements);
    }

    return sqrt(sum_elements);
}

void mat_mult_AdjList(struct Graph *graph, double *vector, double *new_vector, int n_vertexs)
{

    for (int i = 0; i < n_vertexs; i++)
    {
        struct Node *ptr = graph->head[i];

        if (ptr == NULL)
        {
        }
        else
        {
            while (ptr != NULL)
            {
                new_vector[i] += mult_pointers(ptr->weight, vector[ptr->dest]);
                ptr = ptr->next;
            }
        }
    }
}

void printfvector(double *vector, int n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        printf("%lf ", vector[i]);
    }
    printf("\n");
}

void cleanVector(double *Clean_vector, int n_vertexs)
```

```
{
    double zero = 0;
    for (int i = 0; i < n_vertexs; i++)
    {
        Clean_vector[i] = zero;
    }
}

double mult_pointers(double num1, double num2)
{
    double aux1 = num1;
    double aux2 = num2;
    double mult_value = aux1 * aux2;
    return mult_value;
}

void copy_vec(double *vector, double *new_vector, int
   n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        double aux = new_vector[i];
        vector[i] = aux;
    }
}
```

Listing 2: Power iteratrion - Sequential

3.3 Code FindEigen_omp - Parallel

```
#include "AdjLisT.c"
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

void read_arguments_or_abort(int argc, char *argv[]);
double normalize_vec(int n_vertexs, double *vector);
void mat_mult_AdjList(struct Graph *graph, double *vector,
   double *new_vector, int n_vertexs);
void printfvector(double *vector, int n_vertexs);
void cleanVector(double *vector, int n_vertexs);
double mult_pointers(double num1, double num2);
void copy_vec(double *vector, double *new_vector, int
   n_vertexs);

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

```
{

    read_arguments_or_abort(argc, argv);
    char *input_filename = argv[1];

    double precision;
    sscanf(argv[2], "%lf", &precision);

    char *output_filename = argv[3];

    struct File_data *file_data = ReadPajek(input_filename);

    double *vec = (double *)malloc(file_data->n_vertexs * sizeof(double));
    double *new_vec = (double *)malloc(file_data->n_vertexs * sizeof(double));

    for (int i = 0; i < file_data->n_vertexs; i++)
    {
        vec[i] = rand() / (RAND_MAX + 1.0);
        if (rand() / (RAND_MAX + 1.0) >= 0.5)
        {
            vec[i] *= -1;
        }
    }

    int stop_iter = 0;
    double norm_vec, new_norm_vec;

    norm_vec = normalize_vec(file_data->n_vertexs, vec);

    struct timeval t1, t2;
    gettimeofday(&t1, NULL);
    while (stop_iter < 3)
    {

        cleanVector(new_vec, file_data->n_vertexs);
        mat_mult_AdjList(file_data->graph, vec, new_vec, file_data->n_vertexs);
        new_norm_vec = normalize_vec(file_data->n_vertexs, new_vec);
        copy_vec(vec, new_vec, file_data->n_vertexs);

        if (fabs(new_norm_vec - norm_vec) / new_norm_vec < precision)
        {
            stop_iter += 1;
        }
        else
```

```
        {
            stop_iter = 0;
        }

        norm_vec = new_norm_vec;
    }
    gettimeofday(&t2, NULL);

    printf("It took %.17lf milliseconds.\n", (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);

    FILE *output_file;
    output_file = fopen(output_filename, "w");
    fprintf(output_file, "%lf\n", new_norm_vec);
    fprintf(output_file, "%d\n", file_data->n_vertexs);

    for (int i = 0; i < file_data->n_vertexs; i++)
    {
        fprintf(output_file, "%lf\n", vec[i]);
    }

    fclose(output_file);

    cleanVector(new_vec, file_data->n_vertexs);
    mat_mult_AdjList(file_data->graph, vec, new_vec, file_data->n_vertexs);
    new_norm_vec = normalize_vec(file_data->n_vertexs, new_vec);

    if (fabs(new_norm_vec - norm_vec) / new_norm_vec < precision)
    {
        printf("The Method works well\n");
    }
    else
    {
        printf("The Method don't work so well\n");
        printf("method: %.17lf precision: %.17lf\n", fabs(new_norm_vec - norm_vec) / new_norm_vec, precision);
    }

    FILE *time_record_file;
    char timefilename[100] = "time_omp_";
    strcat(timefilename, input_filename);
    printf("%s\n", timefilename);
    time_record_file = fopen(timefilename, "a+");
    fprintf(time_record_file, "%.10lf\n", (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);
    fclose(time_record_file);
```

```
    return 0;
}

void read_arguments_or_abort(int argc, char *argv[])
{
    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s <number of elements> <number of arrays>\n",
                argv[0]);
        exit(505);
    }
}

double normalize_vec(int n_vertexs, double *vector)
{

    double sum_elements = 0;
#pragma omp parallel for default(none) shared(vector, n_vertexs) reduction(+ \
                                                                          : sum_elements) schedule(static)
    for (int i = 0; i < n_vertexs; i++)
    {
        sum_elements += pow(vector[i], 2);
    }

#pragma omp parallel for default(none) shared(vector, n_vertexs, sum_elements) schedule(static)
    for (int i = 0; i < n_vertexs; i++)
    {
        vector[i] /= sqrt(sum_elements);
    }
    return sqrt(sum_elements);
}

void mat_mult_AdjList(struct Graph *graph, double *vector, double *new_vector, int n_vertexs)
{

#pragma omp parallel for default(none) shared(new_vector, n_vertexs, vector, graph) schedule(dynamic)
    for (int i = 0; i < n_vertexs; i++)
    {
        struct Node *ptr = graph->head[i];
        if (ptr == NULL)
        {
        }
        else
```

```
        {
            while (ptr != NULL)
            {
                new_vector[i] += mult_pointers(ptr->weight, vector[ptr->dest]);
                ptr = ptr->next;
            }
        }
    }
}

void printfvector(double *vector, int n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        printf("%lf ", vector[i]);
    }
    printf("\n");
}

void cleanVector(double *Clean_vector, int n_vertexs)
{
    double zero = 0;
#pragma omp parallel for default(none) shared(Clean_vector, n_vertexs, zero) schedule(static)
    for (int i = 0; i < n_vertexs; i++)
    {
        Clean_vector[i] = zero;
    }
}

double mult_pointers(double num1, double num2)
{
    double aux1 = num1;
    double aux2 = num2;
    double mult_value = aux1 * aux2;
    return mult_value;
}

void copy_vec(double *vector, double *new_vector, int n_vertexs)
{
#pragma omp parallel for default(none) shared(new_vector, vector, n_vertexs) schedule(static)
    for (int i = 0; i < n_vertexs; i++)
    {
        double aux = new_vector[i];
        vector[i] = aux;
    }
```

```
}
```

Listing 3: Power iteratrion - Parallel OpenMP

3.4 Code FindEigen_mpi - Parallel

```
#include "AdjLisT.c"
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <mpi.h>

void read_arguments_or_abort(int argc, char *argv[]);
double normalize_vec(int n_vertexs, double *vector);
void mat_mult_AdjList(struct Graph *graph, double *vector,
    double *new_vector, int *jobs, int size_jobs);
void printfvector(double *vector, int n_vertexs);
void printfIntVector(int *vector, int n_vertexs);
void cleanVector(double *vector, int n_vertexs);
double mult_pointers(double num1, double num2);
void copy_vec(double *vector, double *new_vector, int
    n_vertexs);

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    //numbers of processors and rank
    int nprocs, rank;

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

    if (argc != 4)
    {
        if(rank==0){
            fprintf(stderr, "Usage: %s <number of elements> <
    number of arrays>\n",argv[0]);
        }
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    //Read Pajek File
    char *input_filename = argv[1];
    double precision;
    sscanf(argv[2], "%lf", &precision);
```

```
    char *output_filename = argv[3];
    struct File_data *file_data = ReadPajek(input_filename);

    //work division
    int complete_sections = (file_data->n_vertexs)/nprocs;
    int rest_sections = file_data->n_vertexs - nprocs*complete_sections;

    int aux_size;
    double *auxvec= NULL;

    int *indexJobs=NULL;

    if (rank==0){

        aux_size = complete_sections + rest_sections;
        auxvec = (double *)malloc(aux_size * sizeof(double));

        indexJobs= (int *)malloc((complete_sections+rest_sections)*sizeof(int));
        for (int i=0; i<complete_sections+rest_sections;  ++i){
            indexJobs[i]= i;
        }
    }else{
        aux_size = complete_sections;
        auxvec = (double *)malloc(aux_size * sizeof(double));

        indexJobs= (int *)malloc((complete_sections)*sizeof(int));
        for (int i=0; i<complete_sections; ++i){
            indexJobs[i]= complete_sections*rank+rest_sections + i;
        }
    }

    double *vec = (double *)malloc((file_data->n_vertexs) * sizeof(double));
    double *new_vec = (double *)malloc((file_data->n_vertexs) * sizeof(double));
    double norm_vec, new_norm_vec;

    if (rank == 0){
        for (int i = 0; i < file_data->n_vertexs; i++)
        {
            vec[i] = rand() / (RAND_MAX + 1.0);
            if (rand() / (RAND_MAX + 1.0) >= 0.5)
            {
                vec[i] *= -1;
```

```
            }
        }
        norm_vec = normalize_vec(file_data->n_vertexs, vec);
    }

    //BroadCast Random vector
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&vec[0], file_data->n_vertexs, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    MPI_Bcast(&norm_vec, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    //wait all threads arrive here
    MPI_Barrier(MPI_COMM_WORLD);

    int stop_iter = 0;


    struct timeval t1, t2;
    if (rank==0){
        gettimeofday(&t1, NULL);
    }

    while (stop_iter < 3)
    {

        if (rank == 0){
            cleanVector(new_vec, file_data->n_vertexs);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&new_vec[0], file_data->n_vertexs,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

        //multiplication
        cleanVector(auxvec, aux_size);
        mat_mult_AdjList(file_data->graph, vec, auxvec,
indexJobs, aux_size);
        //wait all threads arrive here
        MPI_Barrier(MPI_COMM_WORLD);


        if (rank == 0){
            MPI_Gather(&auxvec[0], aux_size, MPI_DOUBLE, &
new_vec[0], aux_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }else{
            if (rank !=0){
                MPI_Gather(&auxvec[0], aux_size, MPI_DOUBLE,
NULL, aux_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            }

```

```
        }
        //wait all threads arrive here
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&new_vec[0], file_data->n_vertexs,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);

        if (rank==0){
            new_norm_vec = normalize_vec(file_data->n_vertexs
    , new_vec);
        }
        //wait all threads arrive here
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&new_vec[0], file_data->n_vertexs,
    MPI_DOUBLE, 0, MPI_COMM_WORLD);
        //wait all threads arrive here
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(&new_norm_vec, 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
        copy_vec(vec, new_vec, file_data->n_vertexs);

        if (fabs(new_norm_vec - norm_vec) / new_norm_vec <
    precision)
        {
            stop_iter += 1;
        }
        else
        {
            stop_iter = 0;
        }

        norm_vec = new_norm_vec;

        //wait all threads arrive here
        MPI_Barrier(MPI_COMM_WORLD);
    }



    if (rank == 0){
        gettimeofday(&t2, NULL);

        printf("It took %.17lf milliseconds.\n", (t2.tv_sec -
    t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);

        FILE *output_file;
        output_file = fopen(output_filename, "w");
        fprintf(output_file, "%lf\n", new_norm_vec);
        fprintf(output_file, "%d\n", file_data->n_vertexs);

```

```
        for (int i = 0; i < file_data->n_vertexs; i++)
        {
            fprintf(output_file, "%lf\n", vec[i]);
        }
        fclose(output_file);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    cleanVector(new_vec, file_data->n_vertexs);
    cleanVector(auxvec, aux_size);
    mat_mult_AdjList(file_data->graph, vec, auxvec, indexJobs, aux_size);

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank==0){
        new_norm_vec = normalize_vec(file_data->n_vertexs, new_vec);
    }


    if (rank == 0){
        if (fabs(new_norm_vec - norm_vec) / new_norm_vec < precision)
        {
            printf("The Method works well\n");
        }
        else
        {
            printf("The Method don't work so well\n");
            printf("method: %.17lf precision: %.17lf\n", fabs(new_norm_vec - norm_vec) / new_norm_vec, precision);
        }

        FILE *time_record_file;
        char timefilename[100] = "time_mpi_";
        strcat(timefilename, input_filename);
        printf("%s\n", timefilename);
        time_record_file = fopen(timefilename, "a+");
        fprintf(time_record_file, "%.10lf\n", (t2.tv_sec - t1.tv_sec) + (t2.tv_usec - t1.tv_usec) / 1e6);
        fclose(time_record_file);

    }

    MPI_Finalize();

    return 0;
```

```c
}


double normalize_vec(int n_vertexs, double *vector)
{

    double sum_elements = 0;
    for (int i = 0; i < n_vertexs; i++)
    {
        sum_elements += pow(vector[i], 2);
    }

    for (int i = 0; i < n_vertexs; i++)
    {
        vector[i] /= sqrt(sum_elements);
    }

    return sqrt(sum_elements);
}

void mat_mult_AdjList(struct Graph *graph, double *vector,
    double *new_vector, int *jobs, int size_jobs)
{

    for (int i = 0; i < size_jobs; i++)
    {
        struct Node *ptr = graph->head[jobs[i]];

        if (ptr == NULL)
        {
        }
        else
        {
            while (ptr != NULL)
            {
                new_vector[i] += mult_pointers(ptr->weight,
    vector[ptr->dest]);
                ptr = ptr->next;
            }
        }
    }
}

void printfvector(double *vector, int n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        printf("%lf ", vector[i]);
    }
```

```
    printf("\n");
}

void printfIntVector(int *vector, int n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        printf("%d ", vector[i]);
    }
    printf("\n");
}

void cleanVector(double *Clean_vector, int n_vertexs)
{
    double zero = 0;
    for (int i = 0; i < n_vertexs; i++)
    {
        Clean_vector[i] = zero;
    }
}

double mult_pointers(double num1, double num2)
{
    double aux1 = num1;
    double aux2 = num2;
    double mult_value = aux1 * aux2;
    return mult_value;
}

void copy_vec(double *vector, double *new_vector, int
   n_vertexs)
{
    for (int i = 0; i < n_vertexs; i++)
    {
        double aux = new_vector[i];
        vector[i] = aux;
    }
}
```

Listing 4: Power iteratrion - Parallel MPI

4 Performance analysis

For the performance analysis, the test files available in the compiled file powerit_test.tar.gz were used. With this in mind, error bar plots were performed to verify the processing time of each code.

4.1 Small

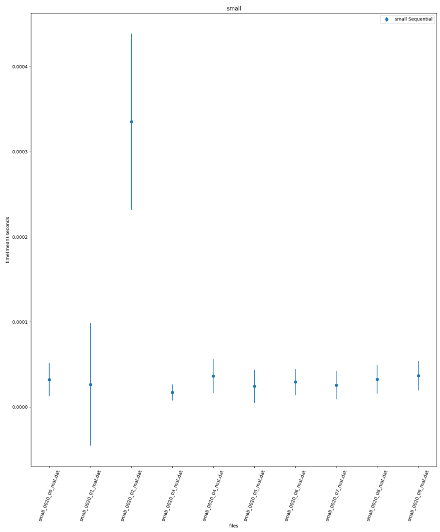

Figure 4: Sequential

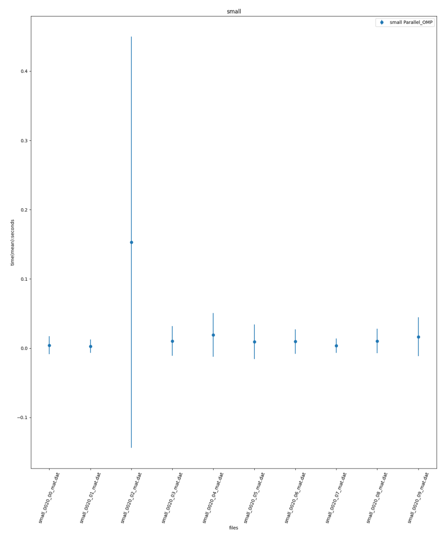

Figure 5: Parallel OpenMP

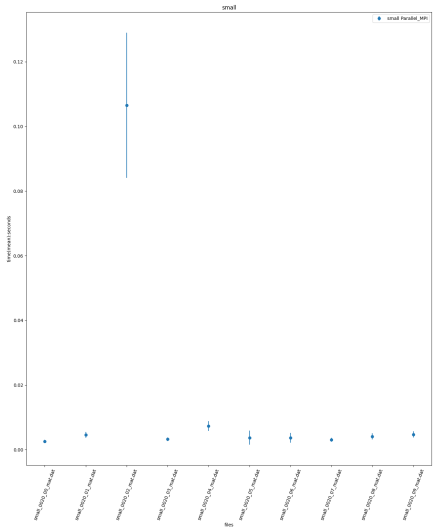

Figure 6: Parallel MPI

4.2 Medium

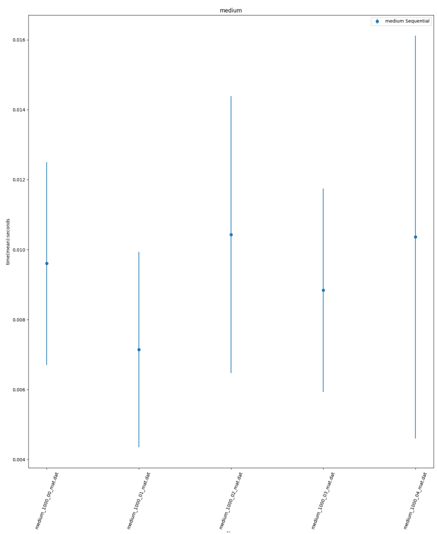

Figure 7: Sequential

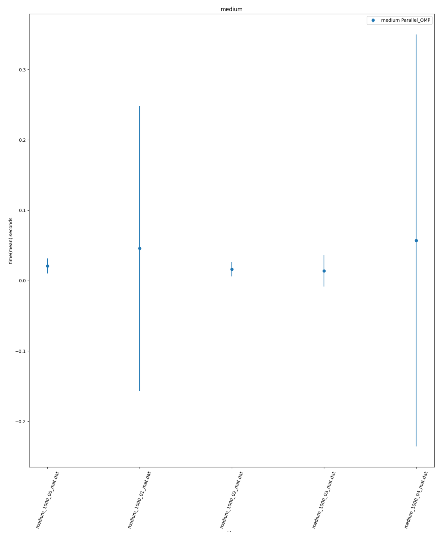

Figure 8: Parallel OpenMP

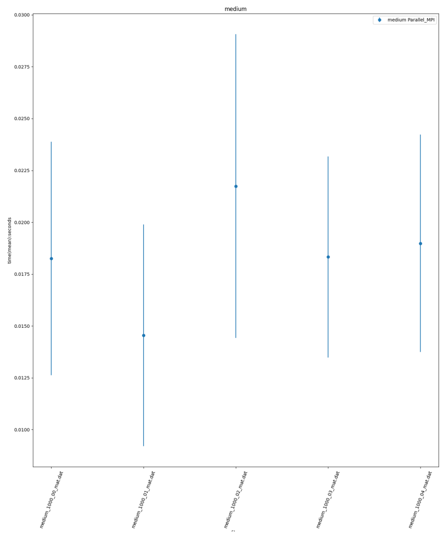

Figure 9: Parallel MPI

4.3 Large

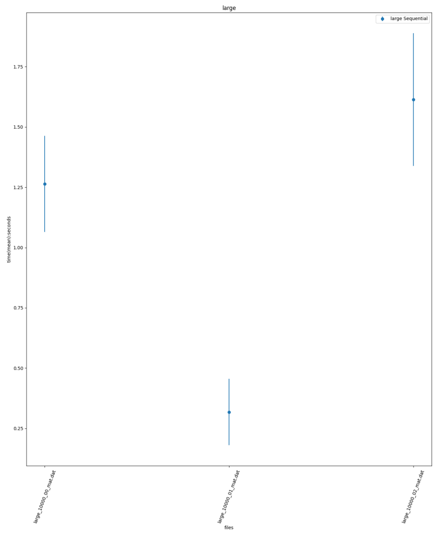

Figure 10: Sequential

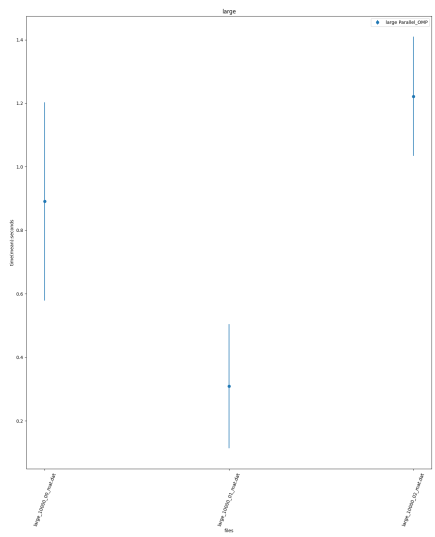

Figure 11: Parallel OpenMP

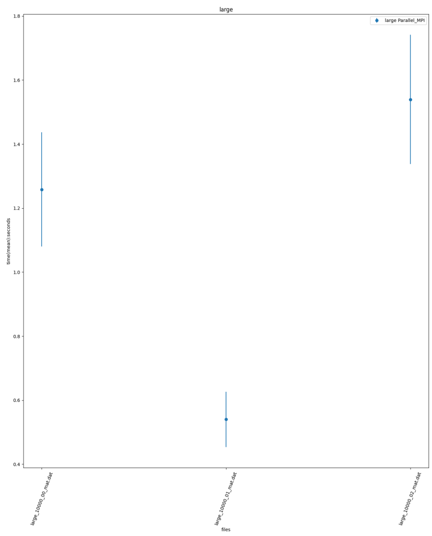

Figure 12: Parallel MPI

4.4 Huge

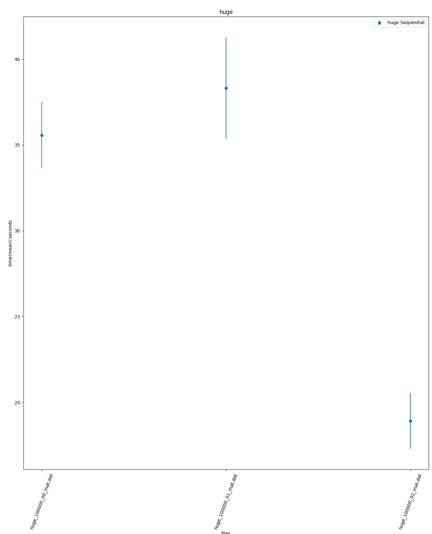

Figure 13: Sequential

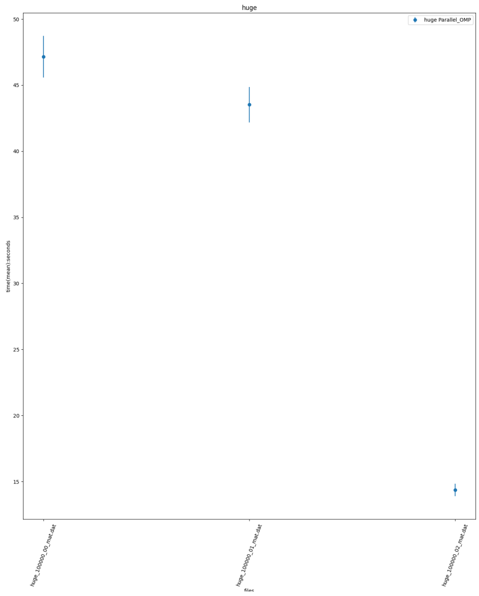

Figure 14: Parallel OpenMP

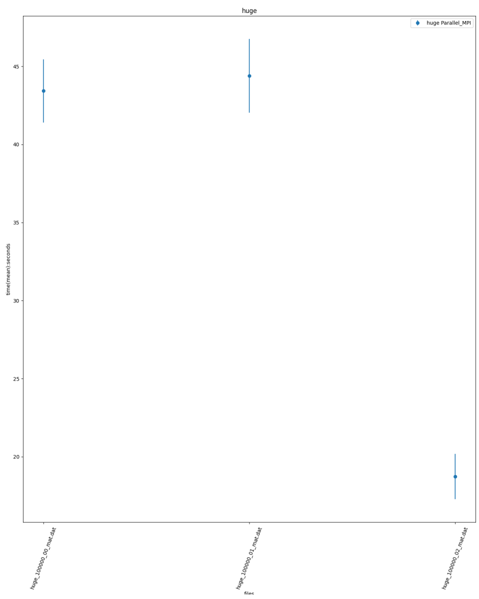

Figure 15: Parallel MPI

4.5 All size files

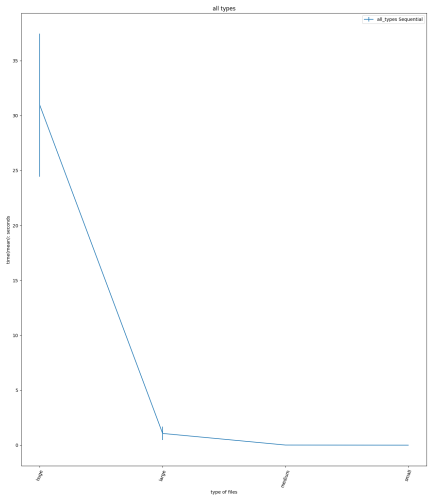

Figure 16: Mean of each type file - Sequential

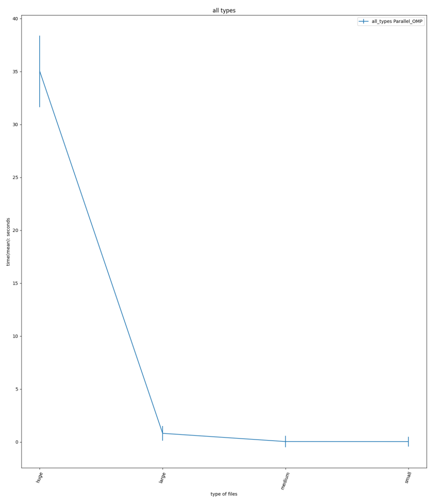

Figure 17: Mean of each type file - Parallel OpenMP

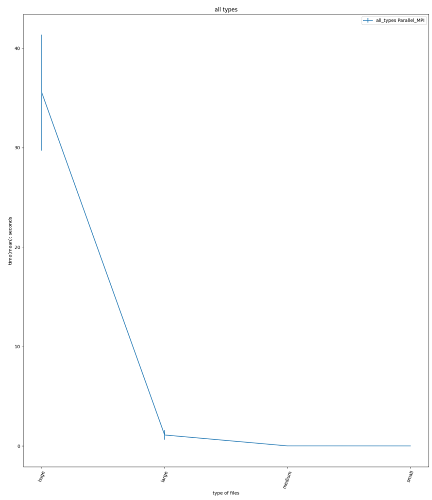

Figure 18: Mean of each type file - Parallel MPI

4.6 Comparisons

In view of the images presented above, an image was made with the implementations together for better analysis.

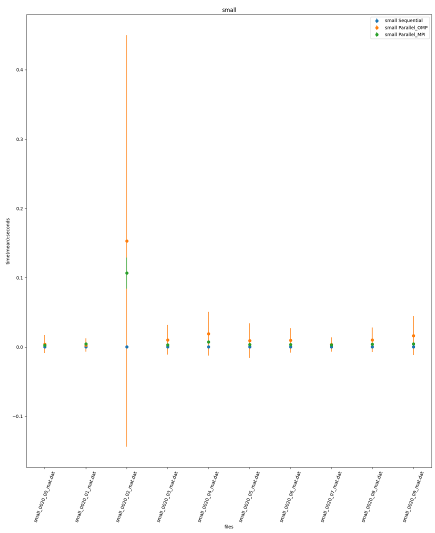

Figure 19: Small

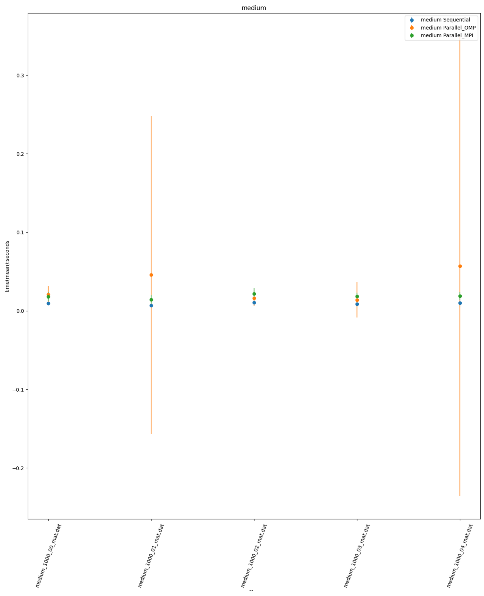

Figure 20: Medium

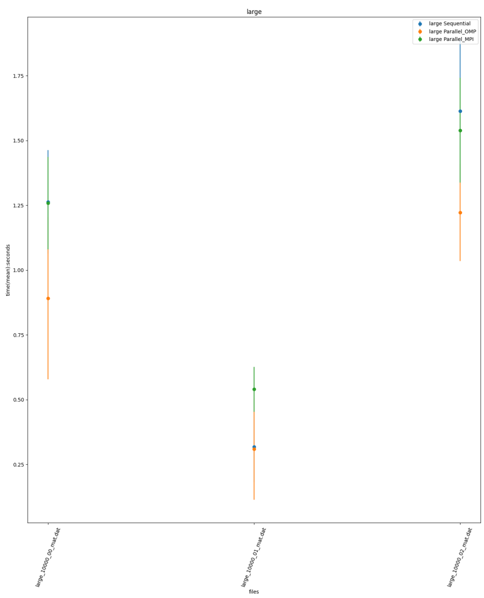

Figure 21: Large

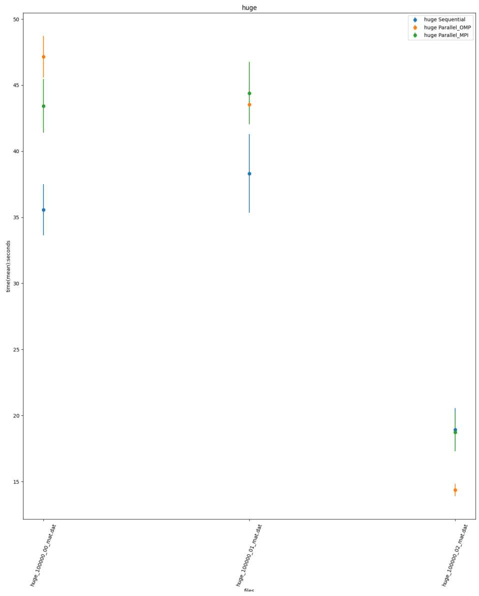

Figure 22: Huge

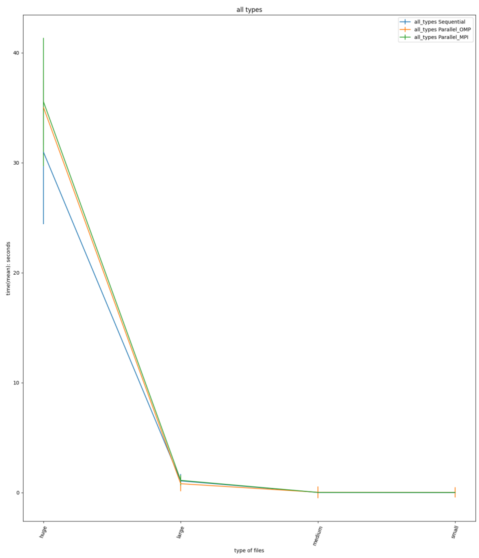

Figure 23: All Types

5 Conclusions

In short, among the three implementations made in this report, the routine that had the best performance was the sequential code. However, it would be expected that the parallel codes would have better performance due to the partition of work between the processors. This result was obtained because the machine that was used contained only two processors, so the passing of messages needed during the parallelization procedures took a large portion of time, leading to an increase in time that exceeded the work divided into the processors.

6 Reference

[1] Links with the images of Graph and Adjancecy List

[2] da Silva, Éverton Luís Mendes. Codes from this project

[3] da Silva, Éverton Luís Mendes. Codes and images from this project in Google Drive