Projeto B - Paralelo

Anderson Araujo de Oliveira, N USP: 11371311

1 Parte I - Discussão sobre o código

1.1 Leitura do Arquivo/Criação do Grafo/Comunicação

O código começa verificando se o arquivo foi aberto corretamente. Em seguida, ele lê o número de nós e arestas presentes no grafo, extraindo essas informações do arquivo.Por fim, realiza a comunicação dos valores de M (nós) e N (arestas) aos demais processos.

```
    FILE *arquivo = fopen(argv[1], "r");
    if (rank == 0) {
        if (arquivo == NULL)
        { // verificando se consegue abrir o arquivo
            fprintf(stderr, "Erro ao abrir o arquivo.\n");
            MPI_Finalize();
            return 2;
        }

        int read1 = fscanf(arquivo, " %d", &N);
        int read2 = fscanf(arquivo, " %d", &M);

        if (read1 != 1)
        { // verificando se os parametros de arestas e nos
            fprintf(stderr, "Erro na leitura da leitura da quantidade de n  .\n");
            MPI_Finalize();
            return 3;
        }
        if (read2 != 1)
        {
            fprintf(stderr, "Erro na leitura na leitura da quantidade de aresta.\n");
            MPI_Finalize();
            return 4;
        }
    }
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&M, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

1.1.1 Lista de adjacência

Nessa etapa o código lê as conexões entre os nós do grafo(as arestas), mas só o processo 0 faz isso. Para representar as arestas, ele utiliza uma lista de adjacência. Se fossemos utilizar uma matriz de adjacência, a complexidade seria $O(N^2)$, devido à verificação se um nó é vizinho, onde N é o número de nós. Com a lista de adjacência, a complexidade se torna $O(N)$, não tendo que fazer essa verificação.

```
    //Valor medio de vizinhos como tamanho das listas de adjacencia
    int k = (int)ceil(M/N);
    //criando o grafo usando lista de adjacencia
    Grafo *grafo = criarGrafo(N, k);

    if (rank == 0) {
        // adicionando as arestas
        for (int i = 0; i < M; i++)
        {
```

```
            if (fscanf(arquivo, "%d %d", &u, &v) != 2)
            {
                fprintf(stderr, "Erro ao ler aresta.\n");
                MPI_Finalize();
                return 1;
            }
            adicionarAresta(grafo, u - 1, v - 1); // Ajusta para  ndices  baseados em 0
        }


    }
```

1.1.2 Comunicação

Esta etapa do código efetua a distribuição completa das informações do grafo para todos os processos. Isso inclui o tamanho das listas de adjacência e os valores contidos em cada lista, garantindo que todos os processos possuam uma cópia da estrutura do grafo.

```
    //Broadcast do grafo
    MPI_Bcast(grafo->tamanho_lista, N, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(grafo->indice, N, MPI_INT, 0, MPI_COMM_WORLD);

    for (int i = 0; i < N; i++) {
        int tamanho_lista = grafo->tamanho_lista[i]; // Envia o tamanho da lista
   primeiro
        MPI_Bcast(&tamanho_lista, 1, MPI_INT, 0, MPI_COMM_WORLD);

        // Todos os processos alocam o tamanho correto da lista
        if (rank != 0) {
            grafo->lista[i] = (int *)malloc(tamanho_lista * sizeof(int));
        }

        MPI_Bcast(grafo->lista[i], tamanho_lista, MPI_INT, 0, MPI_COMM_WORLD);
    }
```

1.2 Calculo da eficiência

1.2.1 Calculo da eficiência

Esta etapa crucial no código, calcular a eficiência do grafo. Mas antes de calcular a eficiência a equação abaixo, precisamos determinar as distâncias entre todos os pares de nós. É denotada por d_{ij}, representa o menor número de arestas que devemos percorrer para ir do nó i ao nó j.

$$E = \frac{1}{N(N-1)} \sum_{i=0}^{N-1} \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{1}{d_{ij}} \tag{1}$$

O código utiliza o algoritmo de busca em largura (BFS) para calcular distância. A ideia é explorar o grafo camada por camada, partindo do nó de origem. Os nós vizinhos são armazenados em uma fila, garantindo a ordem de visitação. Durante a busca, as distâncias de cada nó em relação à origem são calculadas e armazenadas. O processo continua até que todos os nós acessíveis a partir da origem tenham sido visitados e suas distâncias calculadas.

A paralelização do código se dá pela divisão da tarefa de calcular a eficiência entre os processos disponíveis, onde cada processo atua de forma independente, calculando a eficiência de um subconjunto específico de nós de tamanho proporcional $\frac{N}{p}$, onde "p" é quantidade de processos.

```c
// Fun   o para calcular a efici ncia do grafo direcionado
double calcularEficiencia(Grafo *grafo, int rank, int size)
{
    double eficiencia = 0;
    int *distancia = (int *)calloc(grafo->N, sizeof(int));
    int inicio = (grafo->N / size) * rank;
    int fim = (grafo->N / size) * (rank + 1);
    if (rank == size - 1) {
        fim = grafo->N;
    }

    for (int origem = inicio; origem < fim; origem++)
    {                                                  // percorrer todos os nos
        buscaEmLargura(grafo, origem, distancia); // algoritmo de busca em largura
        for (int i = 0; i < grafo->N; i++)
        {
            if (i != origem && distancia[i] > 0)
            {
                eficiencia += 1.0 / distancia[i];
                distancia[i] = 0;
            }
        }
    }
    free(distancia);
    // eficiencia = eficiencia / (grafo->N * (grafo->N - 1)); // calculo de efici ncia
    return eficiencia;
}
```

A abaixo a função de busca em largura no grafo. A escolha desse algoritmo se deve à sua capacidade de explorar os nós permitindo registrar as distâncias entre eles. Essa informação sobre as distâncias é crucial para a etapa de calcular a eficiência do grafo.

```c
// Fun   o para realizar a busca em largura a partir de um v rtice de origem
void buscaEmLargura(Grafo *grafo, int origem, int *distancia) {
    int u,v;//criando variaveis para auxiliar
    int *visitado = (int *)calloc(grafo->N, sizeof(int));//criando vetor com os valores
   visitados

    Fila *q = criarFila(grafo->N);//criando fila com tamanho N

    visitado[origem] = 1;//colocar o primeiro no como visitado
    distancia[origem] = 0;//colocando distancia 0 no primeiro no
    enfileirar(q, origem);//adcionionando na fila

    while (!filaVazia(q)) {//parar while quando a fila tiver vazia

        u = desenfileirar(q);
        grafo->indice[u]=0;//come ar lista de vizinhos
        v = grafo -> lista[u][grafo->indice[u]];//inserir o primeiro vizinho

        while (v != -1) {//continuar at  todos vizinhos sejam visitados

            if (visitado[v]==0) {
                visitado[v] = 1;//colocar como vizitado
                distancia[v] = distancia[u] + 1;//calculo da distancia
                enfileirar(q, v);//adicionar vizinho
            }

            ++grafo->indice[u];// ir para o proximo vizinho
```

```
            v = grafo->lista[u][grafo->indice[u]];
        }
    }
    liberarFila(q);// liberar a fila e visitados
    free(visitado);
}
```

1.2.2 Etapa final

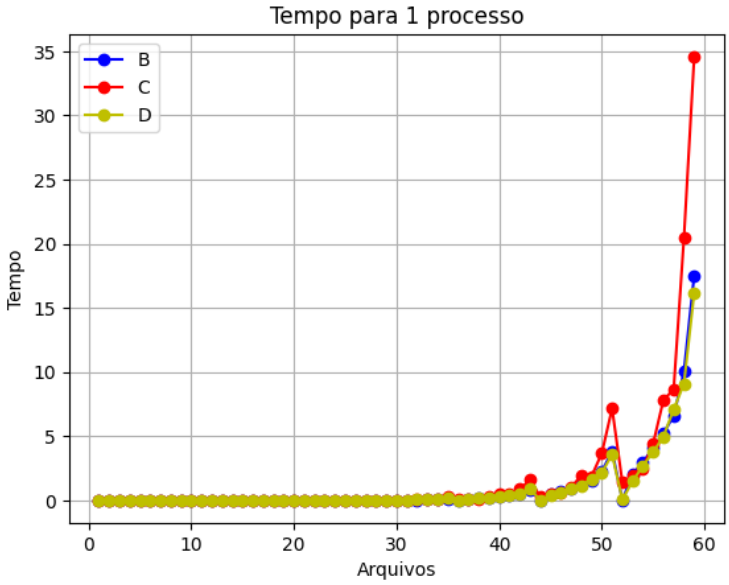
Após o cálculo individual das eficiências por cada processo, procede-se com a soma de todos os valores obtidos. Essa etapa de consolidação garante que a eficiência global do grafo seja determinada pela agregação dos resultados parciais de cada processo, além de imprimir o tempo e o resultado.

```
    // calculo da efici ncia
    double eficiencia_local = calcularEficiencia(grafo, rank, size);
    double eficiencia;

    MPI_Reduce(&eficiencia_local, &eficiencia, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD
    );

    if (rank == 0) {
        // terminando a leitura do tempo
        clock_gettime(CLOCK_MONOTONIC, &fim);

        tempo_decorrido = (fim.tv_sec - inicio.tv_sec) +
                          (fim.tv_nsec - inicio.tv_nsec) / 1e9;
        eficiencia = eficiencia / (grafo->N * (grafo->N - 1));
        // imprindo resultado no terminal
        //printf("Efici ncia do grafo: %6f\n", eficiencia);
        printf("%6f %6f\n",eficiencia,tempo_decorrido+tempo_decorrido2);
        //printf("Tempo decorrido: %6f segundos\n", tempo_decorrido+tempo_decorrido2);
        liberarGrafo(grafo);
    }

    MPI_Finalize();

    return 0;
}
```

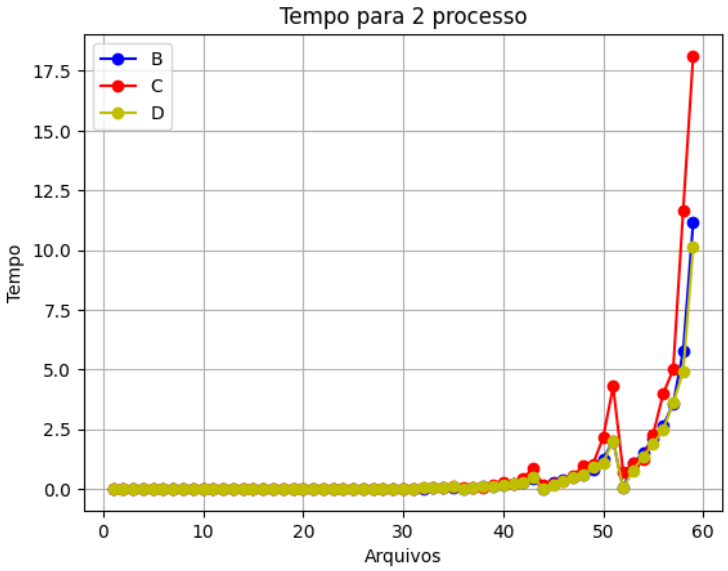
2 Parte II - Comparações

2.1 Desempenho do código

As figuras a seguir apresentam uma análise comparativa do desempenho do código ao processar diferentes tipos de grafos, variando também a quantidade de processos utilizados.

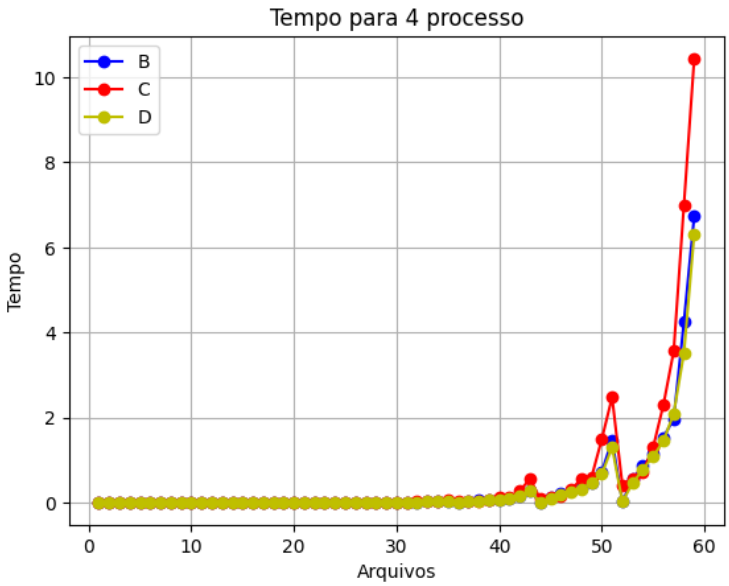


(a) Grafo B

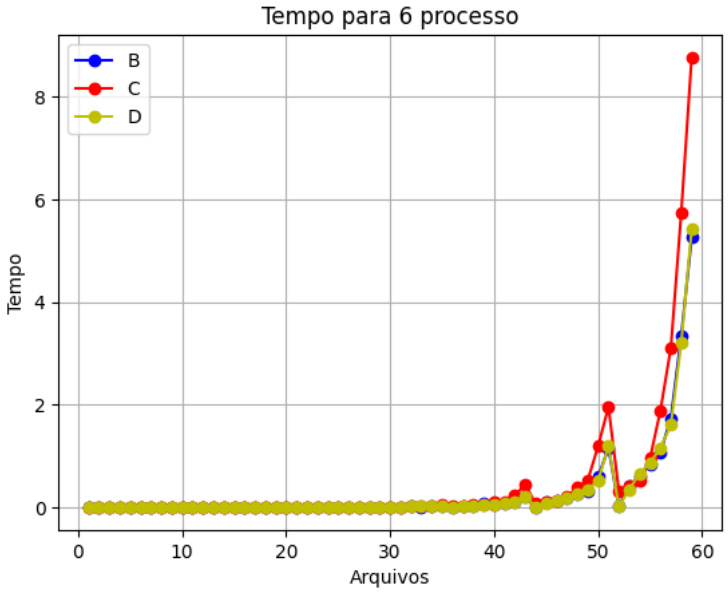


(b) Grafo C

Figura 1: Desempenho do código paralelo para diferentes grafos e processos



(a) Grafo B



(b) Grafo C

Figura 2: Desempenho do código paralelo para diferentes grafos e processos

2.1.1 Complexidade do código

Relembrando a complexidade do algoritmo sequencial, consideramos que para cada um dos N nós, serão calculados as distâncias pelo BFS, ou seja, N vezes algo proporcional a quantidade de arestas M, a complexidade fica $O(NM)$.

No código paralelo à complexidade vai ser $O(\frac{NM}{p})$, essa redução na complexidade se deve à divisão da tarefa de calcular a eficiência dos nós entre os "p" processos disponíveis. Cada processo fica responsável por calcular

a eficiência de um subconjunto de nós, diminuindo a carga de trabalho individual e, consequentemente, o tempo total de processamento.

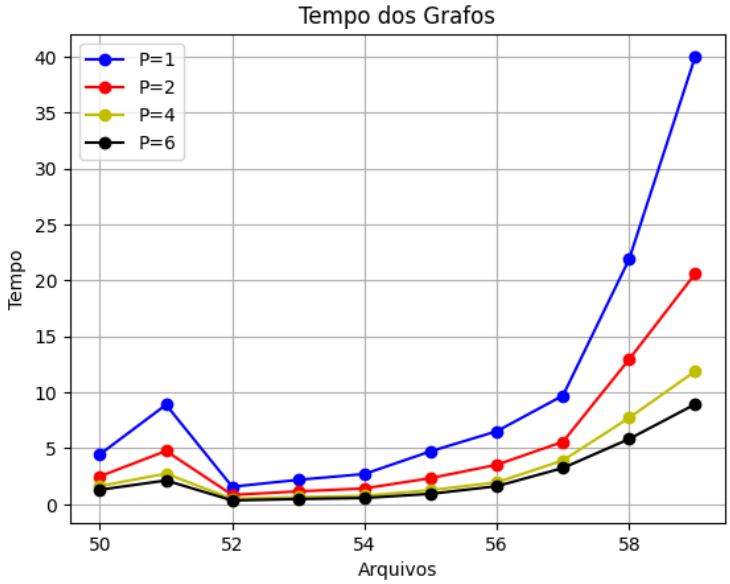
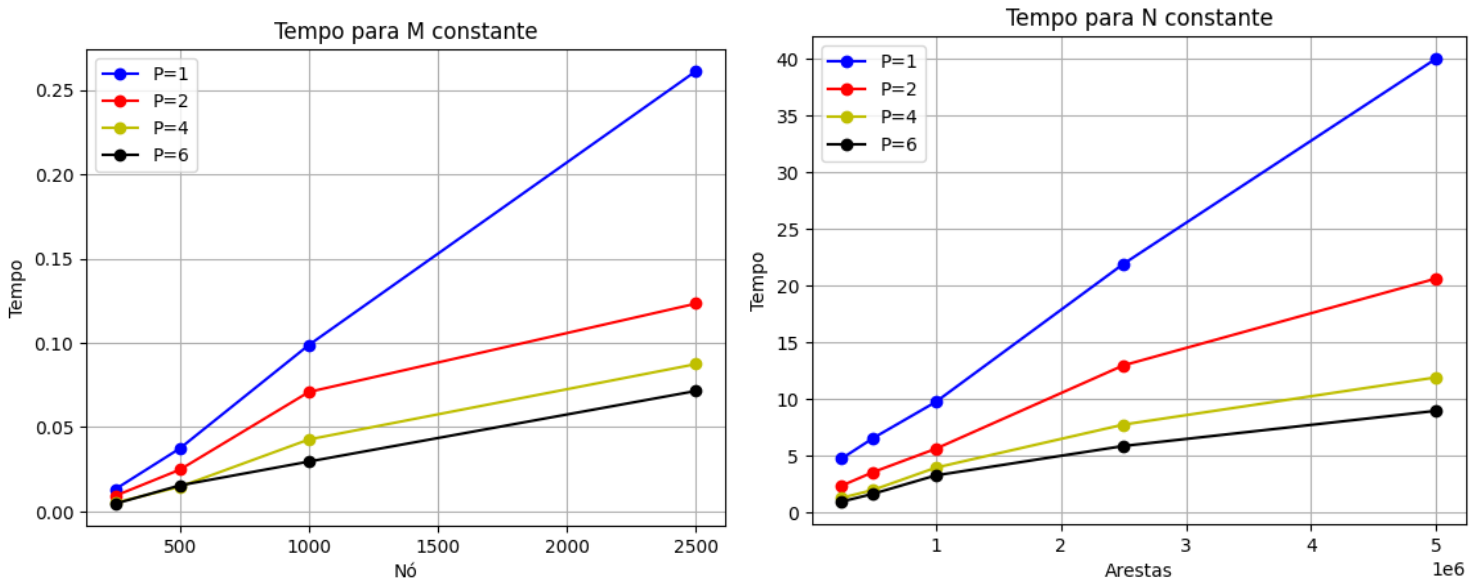


Figura 3: Grafos com maior quantidade de arestas e nós

Os gráficos abaixo ilustram o comportamento do código em cenários onde o número de nós (M) e o número de arestas (N) são mantidos constantes.



(a) M Constantes, M=25000

(b) N Constantes, N=10000

Figura 4

2.1.2 Eficiência do código

A figura apresenta uma análise do desempenho do código ao processar o grafo definido no arquivo "C_59.net"(as outras figuras foi utilizado o mesmo arquivo), avaliando o impacto do número de processos na eficiência da execução.

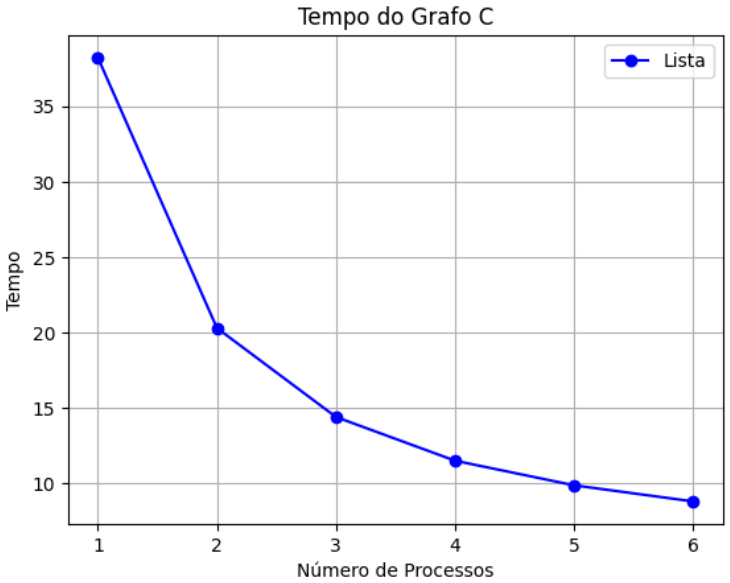


Figura 5: Desempenho do código do arquivo "C_59.net" em função da quantidade de processos

As imagens a seguir ilustram a eficiência e o Speed Up alcançados pelo código.

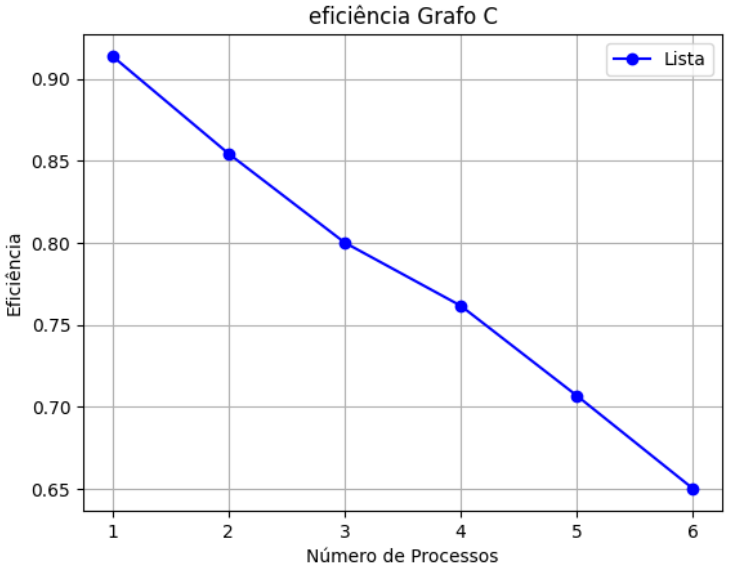


Figura 6: Eficiência do código

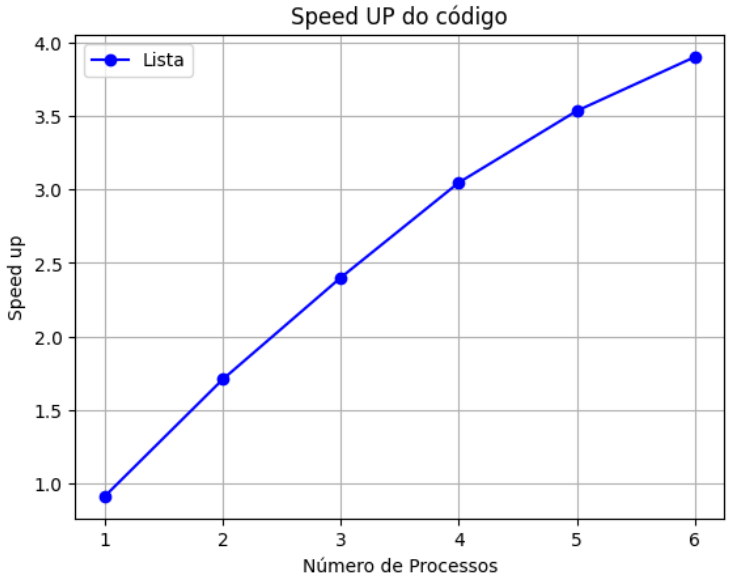


Figura 7: Speed Up do código