Lucas C.
Machado

Dijsktra OpenMP

Código Original

```c
int *dijkstra(struct Graph *graph, int source) {
    int nNodes = graph->nNodes;
    int *visited = (int *) malloc(sizeof(int) * nNodes);
    int *distances = (int *) malloc(sizeof(int) * nNodes);
    int k, v;

    for (v = 0; v < nNodes; v++) {
        distances[v] = INT_MAX;
        visited[v] = 0;
    }
    distances[source] = 0;
    visited[source] = 1;
    for (k = 0; k < graph->nEdges[source]; k++)
        distances[graph->edges[source][k]] = graph->w[source][k];

    for (v = 1; v < nNodes; v++) {
        int min = 0;
        int minValue = INT_MAX;
        for (k = 0; k < nNodes; k++)
            if (visited[k] == 0 && distances[k] < minValue) {
                minValue = distances[k];
                min = k;
            }

        visited[min] = 1;

        for (k = 0; k < graph->nEdges[min]; k++) {
            int dest = graph->edges[min][k];
            if (distances[dest] > distances[min] + graph->w[min][k])
                distances[dest] = distances[min] + graph->w[min][k];
        }
    }

    free(visited);

    return distances;
}
```

Código Otimizado

Código Otimizado

```
int *dijkstra(struct Graph *graph, int source) {
    int nNodes = graph->nNodes;
    int *visited = (int *) malloc(sizeof(int) * nNodes);
    int *distances = (int *) malloc(sizeof(int) * nNodes);
    int v, k;

    for (v = 0; v < nNodes; v++) {
        distances[v] = INT_MAX;
        visited[v] = 0;
    }
    distances[source] = 0;
    visited[source] = 1;

    for (k = 0; k < graph->nEdges[source]; k++)
        distances[graph->edges[source][k]] = graph->w[source][k];

    for (v = 1; v < nNodes; v++) {

        int min = -1;
        int minValue = INT_MAX;

        #pragma omp parallel
        {
            int localMin = -1;
            int localMinValue = INT_MAX;

            #pragma omp for nowait
            for (k = 0; k < nNodes; k++) {
                if (visited[k] == 0 && distances[k] < localMinValue) {
                    localMinValue = distances[k];
                    localMin = k;
                }
            }

            #pragma omp critical
            {
                if (localMinValue < minValue) {
                    minValue = localMinValue;
                    min = localMin;
                }
            }
        }
```

```
        if (min == -1) break;

        visited[min] = 1;

        #pragma omp parallel
        {       You, 7 seconds ago • Uncommitted changes
            #pragma omp single
            for (k = 0; k < graph->nEdges[min]; k++) {
                int dest = graph->edges[min][k];
                int newDist = distances[min] + graph->w[min][k];
                if (newDist < distances[dest]) {
                    #pragma omp task firstprivate(dest, newDist)
                    {
                        if (newDist < distances[dest])
                            distances[dest] = newDist;
                    }
                }
            }
        }
    }

    free(visited);
    return distances;
}
```

Código Otimizado

```
int *dijkstra(struct Graph *graph, int source) {
    int nNodes = graph->nNodes;
    int *visited = (int *) malloc(sizeof(int) * nNodes);
    int *distances = (int *) malloc(sizeof(int) * nNodes);
    int v, k;

    for (v = 0; v < nNodes; v++) {
        distances[v] = INT_MAX;
        visited[v] = 0;
    }
    distances[source] = 0;
    visited[source] = 1;

    for (k = 0; k < graph->nEdges[source]; k++)
        distances[graph->edges[source][k]] = graph->w[source][k];

    for (v = 1; v < nNodes; v++) {

        int min = -1;
        int minValue = INT_MAX;

        #pragma omp parallel
        {
            int localMin = -1; /* local variable for each thread*/
            int localMinValue = INT_MAX;

            #pragma omp for nowait
            for (k = 0; k < nNodes; k++) {
                if (visited[k] == 0 && distances[k] < localMinValue) {
                    localMinValue = distances[k];
                    localMin = k;
                }
            }
            #pragma omp critical
            {
                if (localMinValue < minValue) {
                    minValue = localMinValue;
                    min = localMin;
                }
            }
        }

        if (min == -1) break;

        visited[min] = 1;

        #pragma omp parallel
        {
            #pragma omp single
            for (k = 0; k < graph->nEdges[min]; k++) {
                int dest = graph->edges[min][k];
                int newDist = distances[min] + graph->w[min][k];
                if (newDist < distances[dest]) {
                    #pragma omp task firstprivate(dest, newDist)
                    {
                        if (newDist < distances[dest])
                            distances[dest] = newDist;
                    }
                }
            }
        }
    }

    free(visited);
    return distances;
}
```

Entendendo o Código

#pragma omp critical

Garante que somente uma thread por vez atualize os valores globais (minValue e min).

#pragma omp for nowait

Divide o loop for entre threads, sem que elas esperem no final.

Local Min

Guarda o nó com a menor distância em cada thread, começa com -1 para não tratada como o índice de um nó válido antes mesmo que o loop avalie os nós

```
#pragma omp parallel
{
    int localMin = -1;
    int localMinValue = INT_MAX;

    #pragma omp for nowait
    for (k = 0; k < nNodes; k++) {
        if (visited[k] == 0 && distances[k] < localMinValue) {
            localMinValue = distances[k];
            localMin = k;
        }
    }

    #pragma omp critical
    {
        if (localMinValue < minValue) {
            minValue = localMinValue;
            min = localMin;
        }
    }
}
```

Entendendo o Código

#pragma omp parallel

Habilita o ambiente de paralelismo

#pragma omp single

Garante que apenas uma thread crie as tarefas

#pragma omp task

Garante que apenas uma thread Cria uma tarefa para calcular e atualizar a distância para cada vizinho. Essas tarefas podem ser executadas em paralelo por threads diferentes. as tarefas

#pragma omp single

As variáveis **dest** e **newDist** são passadas como cópias para cada tarefa, evitando conflitos entre threads

```
    #pragma omp parallel
    {
        #pragma omp single
        for (k = 0; k < graph->nEdges[min]; k++) {
            int dest = graph->edges[min][k];
            int newDist = distances[min] + graph->w[min][k];
            if (newDist < distances[dest]) {
                #pragma omp task firstprivate(dest, newDist)
                {
                    if (newDist < distances[dest])
                        distances[dest] = newDist;
                }
            }
        }
    }
}
```

Script Utilizado para testes

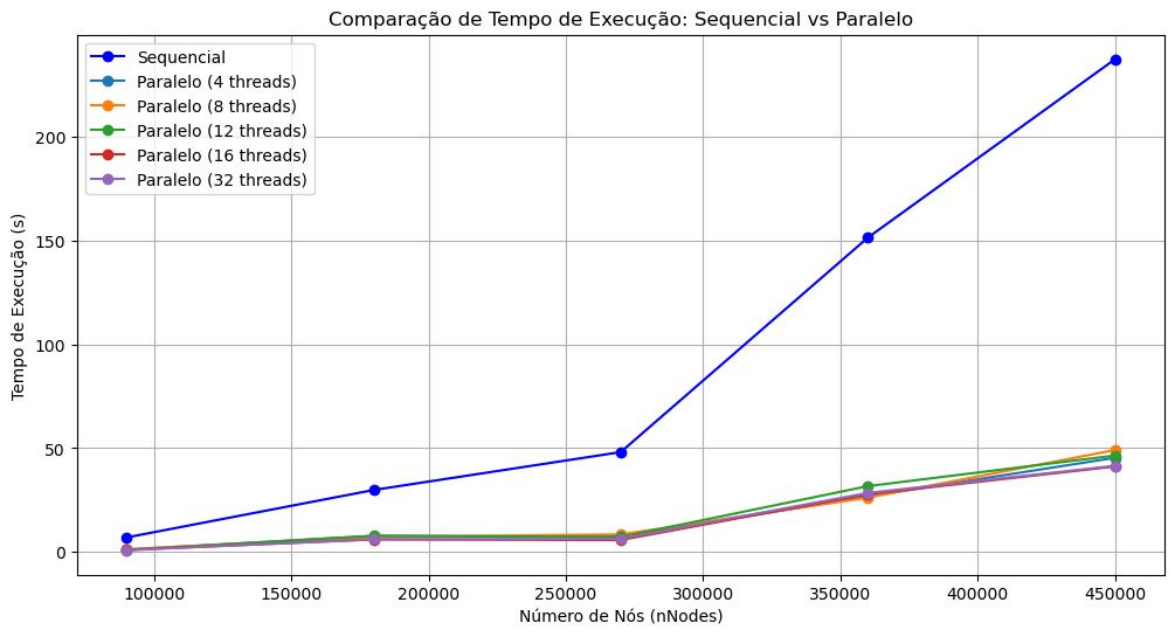
```
for j in {1..5}; do
    OMP_NUM_THREADS=$((j*4 ))
    echo  ">>NUM_THREADS= $OMP_NUM_THREADS" >> "resultados.txt"
    for i in {1..5}; do
        n=$((90000 * i ))
        m=$((i * 200  ))
        r=$(((i % 5) + 1))

        ./dijkstra "$n" "$m" "$r" >> "resultados.txt"
    done
done
```

Script de Testes

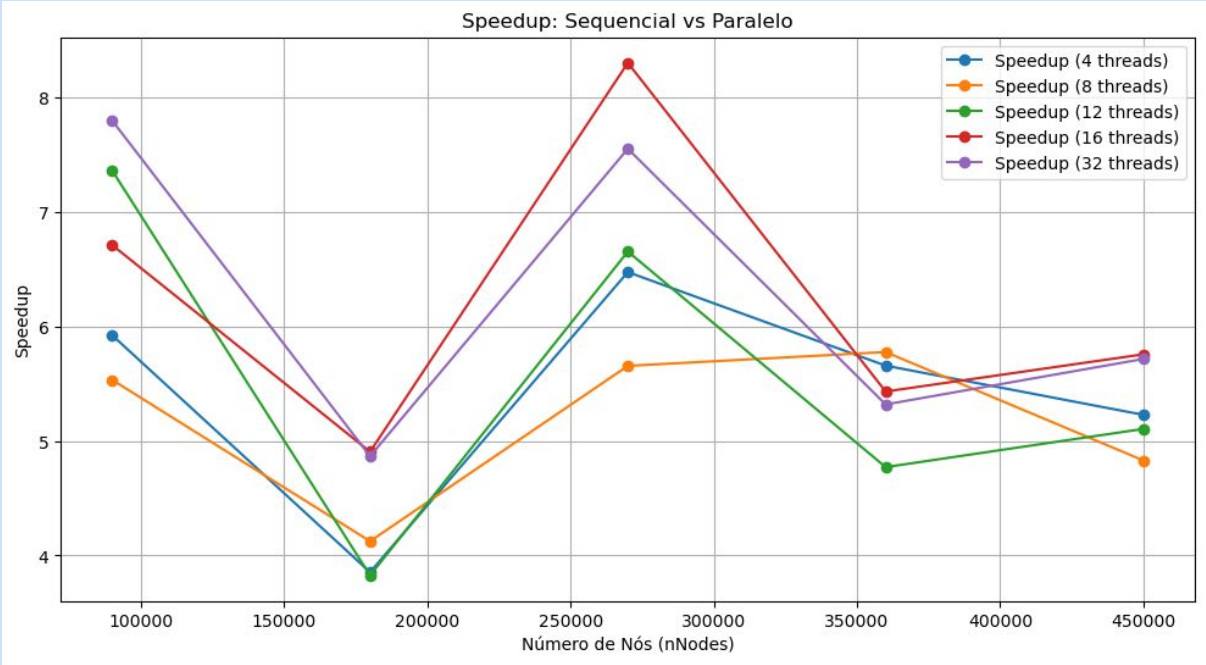
```
for j in {1..5}; do
    OMP_NUM_THREADS=$((j*4 ))
    echo  ">>NUM_THREADS= $OMP_NUM_THREADS" >> "resultados.txt"
    for i in {1..5}; do
        n=$((90000 * i ))
        m=$((i * 200  ))
        r=$(((i % 5) + 1))

        ./dijkstra "$n" "$m" "$r" >> "resultados.txt"
    done
done
```

Resultados

Resultados


Comparação de Tempo de Execução: Sequencial vs Paralelo
Sequencial
Paralelo (4 threads)
Paralelo (8 threads)
Paralelo (12 threads)
Paralelo (16 threads)
Paralelo (32 threads)
Tempo de Execução (s)
0
50
100
150
200
100000
150000
200000
250000
300000
350000
400000
450000
Número de Nós (nNodes)

Resultados


Speedup: Sequencial vs Paralelo
Speedup (4 threads)
Speedup (8 threads)
Speedup (12 threads)
Speedup (16 threads)
Speedup (32 threads)
Speedup
Número de Nós (nNodes)
4
5
6
7
8
100000
150000
200000
250000
300000
350000
400000
450000

Resultados

```
 ================== [Sequencial] NOVA EXECUCAO ==================

  nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 7.017111     :: (mean/nodes): 1073741827.09}
  nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 29.873572   :: (mean/nodes): 4.82}
  nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 48.124940  :: (mean/nodes): 1073741826.44}
  nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 151.385813     :: (mean/nodes): 4.67}
  nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 237.283394     :: (mean/nodes): 4.67}
```

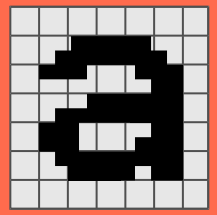
```
================== [Paralelo] NOVA EXECUCAO ==================
>>NUM_THREADS= 4
{ nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 1.184668     :: (mean/nodes): 1073741827.09}
 { nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 7.751466   :: (mean/nodes): 4.82}
 { nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 7.431504  :: (mean/nodes): 1073741826.44}
 { nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 26.752888     :: (mean/nodes): 4.67}
 { nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 45.381921     :: (mean/nodes): 4.67}
```

Resultados

```
 >>NUM_THREADS= 8
{ nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 1.268016     :: (mean/nodes): 1073741827.09}
 { nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 7.243439   :: (mean/nodes): 4.82}
 { nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 8.507861  :: (mean/nodes): 1073741826.44}
 { nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 26.202981     :: (mean/nodes): 4.67}
 { nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 49.127207     :: (mean/nodes): 4.67}
 >>NUM_THREADS= 12
{ nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 0.953222     :: (mean/nodes): 1073741827.09}
 { nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 7.807954   :: (mean/nodes): 4.82}
 { nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 7.230560  :: (mean/nodes): 1073741826.44}
 { nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 31.725227     :: (mean/nodes): 4.67}
 { nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 46.473750     :: (mean/nodes): 4.67}
 >>NUM_THREADS= 16
{ nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 1.045523     :: (mean/nodes): 1073741827.09}
 { nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 6.089072   :: (mean/nodes): 4.82}
 { nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 5.795452  :: (mean/nodes): 1073741826.44}
 { nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 27.867237     :: (mean/nodes): 4.67}
 { nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 41.211395     :: (mean/nodes): 4.67}
 >>NUM_THREADS= 32
{ nNodes: 90000 :: nEdges: 18000000  :: seed: 2  ::  Tempo de execução: 0.899762     :: (mean/nodes): 1073741827.09}
 { nNodes: 180000 :: nEdges: 72000000  :: seed: 3  ::  Tempo de execução: 6.137646   :: (mean/nodes): 4.82}
 { nNodes: 270000 :: nEdges: 162000000  :: seed: 4  ::  Tempo de execução: 6.371135  :: (mean/nodes): 1073741826.44}
 { nNodes: 360000 :: nEdges: 288000000  :: seed: 5  ::  Tempo de execução: 28.458996     :: (mean/nodes): 4.67}
 { nNodes: 450000 :: nEdges: 450000000  :: seed: 1  ::  Tempo de execução: 41.504310     :: (mean/nodes): 4.67}
```

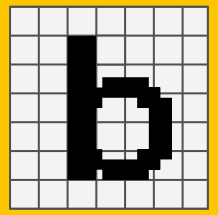
Obrigado!

Foobar



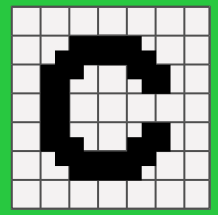
Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit



Lorem ipsum

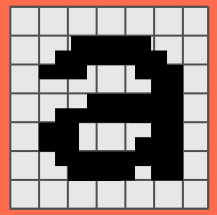
Lorem ipsum dolor sit amet, consectetur adipiscing elit



Lorem ipsum

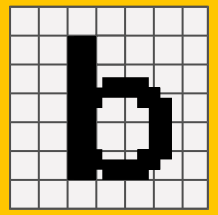
Lorem ipsum dolor sit amet, consectetur adipiscing elit

Foobar



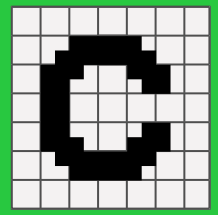
Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit



Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit



Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit