КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Звіт
до лабораторної роботи
з дисципліни "Розподілене та паралельне програмування"
на тему
"Імплементація алгоритму Форда-Беллмана
з використанням послідовних та паралельних обчислень"

Виконала студентка 3 курсу
факультету комп'ютерних наук та кібернетики
спеціальності "Інформатика"
групи ТТП-32
Алєксєєнко Анна Костянтинівна

КИЇВ-2022

Постановка задачі

Імплементувати послідовну та паралельну програму, що виконує алгоритм Форда-Беллмана. Для паралельних обчислень використати MPI та OpenMP. Порівняти час виконання для кожного випадку та порівняти залежність часу виконання від кількості використаних процесорів.

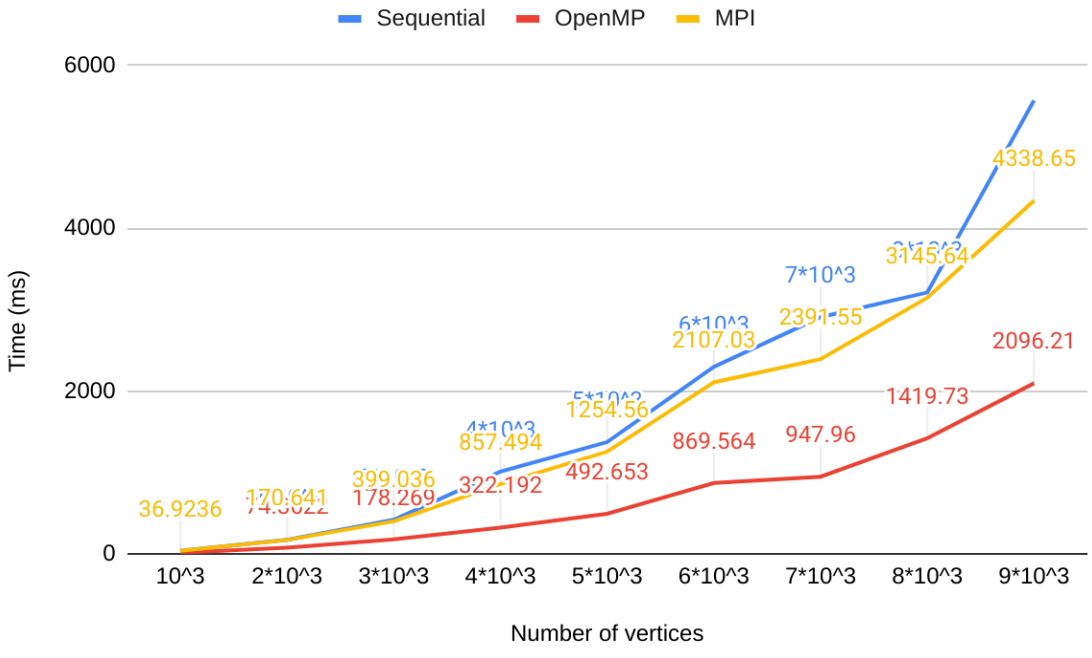
Аналіз алгоритму

Вхідними даними алгоритму є матриця суміжності графу, що містить ваги для кожного ребра. Алгоритм виконується у декілька фаз - спочатку N - 1 разів виконуємо релаксацію ребер (N - кількість вершин у графі, релаксація ребер - спроба мінімізувати відстань до заданої вершини від початкової). Згодом виконується перевірка на наявність циклу від’ємної ваги. Асимптотична складність алгоритму - O(N^3).

Результати виконання

1) Порівняння часу роботи алгоритму в залежності від кількості вершин графа. Заміри виконувалися на 6 процесорах.

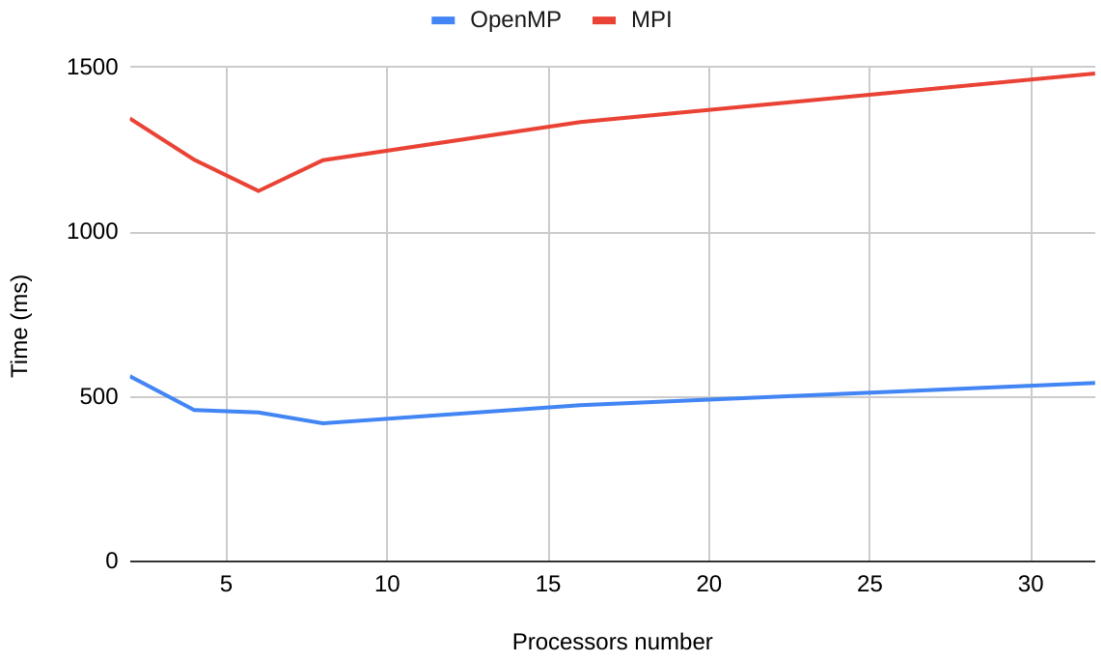
N	Послідовний (ms)	OpenMP (ms)	MPI (ms)
10^3	37.5735	14.8417	36.9236
$2 * 10^3$	172.471	74.3622	170.641
$3 * 10^3$	418.424	178.269	399.036
$4 * 10^3$	1009.24	322.192	857.494
$5 * 10^3$	1373.69	492.653	1254.56
$6 * 10^3$	2297.08	869.564	2107.03
$7 * 10^3$	2907.81	947.960	2391.55
$8 * 10^3$	3211.68	1419.73	3145.64
$9 * 10^3$	5570.27	2096.21	4338.65



Графік 1. Залежність часу виконання від кількості вершин

2) Порівняння часу роботи алгоритму в залежності від кількості процесорів P. Заміри виконувалися для N = 5000.

P	OpenMP	MPI
2	562.374	1345.42
4	459.756	1219.87
6	452.324	1124.99
8	419.785	1218.49
16	474.52	1334.1
32	541.804	1481.72



Графік 2. Залежність часу виконання від кількості процесорів

Висновок

Час виконання зростає за кубічною залежністю при збільшенні кількості вершин у графі. При цьому використання технологій OpenMP та MPI значно зменшує час виконання алгоритму.

Як видно з графіку 1, OpenMP дає більше пришвидшення, ніж MPI, адже алгоритм працює з великим об’ємом даних та потребує копіювання при використанні MPI. Концепція OpenMP shared memory дозволяє позбутися зайвого копіювання та ефективно розпаралелити виконання алгоритму.

Відповідно до графіку 2, обидва методи паралелизації дають мінімальний час при запуску з 8 процесами. Кількість процесів більша за 8 не дає прискорення через особливості машини, що використовувалася для запуску.

Отже, можна зробити висновок, що для паралелизації алгоритма Форда-Беллмана краще підходить OpenMP, адже не потребує копіювання даних та ефективно розподіляє задачі між процесорами.

Додаток 1 - код MPI імплементації

```
#include <string>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cstring>

#include <mpi.h>
#include "utils.hpp"

using namespace std;

#define INF 1000000

int N; //number of vertices
int p;
int *matrix; // the adjacency matrix
int *dist;
bool hasNegativeCycle = false;

int calculateCoordinate(int x, int y, int n) {
   return x * n + y;
}

int readMatrix(const string &filename) {
   std::ifstream ifStream(filename, std::ifstream::in);
   ifStream >> N;
   matrix = (int *) malloc(N * N * sizeof(int));
   for (int i = 0; i < N; i++)
       for (int j = 0; j < N; j++) {
           ifStream >> matrix[calculateCoordinate(i, j, N)];
       }
   return 0;
}

void initDistances(int *copyDist, int &copyN) {
   for (int i = 0; i < copyN; i++) {
       copyDist[i] = INF;
   }

   copyDist[0] = 0;
}

int outputResult(const string &file) {
   std::ofstream outStream(file, std::ofstream::out);
   if (!hasNegativeCycle) {
```

```
        for (int i = 0; i < N; i++) {
            if (dist[i] > INF)
                dist[i] = INF;
            outStream << dist[i] << '\n';
        }
        outStream.flush();
    } else {
        outStream << "Negative cycle" << endl;
    }
    outStream.close();
    return 0;
}

void relax(int start, int end, int copyN, int *copyMatrix, int *copyDist,
bool &rangeChanged) {
    for (int u = start; u < end; u++) {
        for (int v = 0; v < copyN; v++) {
            int weight = copyMatrix[calculateCoordinate(u, v, copyN)];
            if (weight < INF) {
                if (copyDist[u] + weight < copyDist[v]) {
                    copyDist[v] = copyDist[u] + weight;
                    rangeChanged = true;
                }
            }
        }
    }
}

/**
* Bellman-Ford algorithm. Find the shortest path from vertex 0 to other
vertices.
*/
void performBellmanFord(int rank, MPI_Comm comm) {
    int copyN, start, end;
    int *copyMatrix, *copyDist;
    if (rank == 0) {
        copyN = N;
    }

    // broadcast a message to all other processes
    MPI_Bcast(&copyN, 1, MPI_INT, 0, comm);

    // find local task range
    int range = copyN / p;
    start = range * rank;
    end = range * (rank + 1);
    if (rank == p - 1) {
```

```
        end = copyN;
    }

    copyMatrix = (int *) malloc(copyN * copyN * sizeof(int));
    copyDist = (int *) malloc(copyN * sizeof(int));

    if (rank == 0) {
        memcpy(copyMatrix, matrix, sizeof(int) * copyN * copyN);
    }
    MPI_Bcast(copyMatrix, copyN * copyN, MPI_INT, 0, comm);

    initDistances(copyDist, N);

    MPI_Barrier(comm);

    bool rangeChanged;
    int iterNum = 0;
    for (int iter = 0; iter < copyN - 1; iter++) {
        rangeChanged = false;
        iterNum++;
        relax(start, end, copyN, copyMatrix, copyDist, rangeChanged);

        MPI_Allreduce(MPI_IN_PLACE, &rangeChanged, 1,
                      MPI_CXX_BOOL, MPI_LOR, comm);
        if (!rangeChanged)
            break;
        MPI_Allreduce(MPI_IN_PLACE, copyDist, copyN,
                      MPI_INT, MPI_MIN, comm);
    }

    if (iterNum == copyN - 1) {
        rangeChanged = false;
        for (int u = start; u < end; u++) {
            for (int v = 0; v < copyN; v++) {
                int weight = copyMatrix[calculateCoordinate(u, v, copyN)];
                if (weight < INF) {
                    if (copyDist[u] + weight < copyDist[v]) {
                        copyDist[v] = copyDist[u] + weight;
                        rangeChanged = true;
                        break;
                    }
                }
            }
        }
        MPI_Allreduce(&rangeChanged, &hasNegativeCycle, 1, MPI_CXX_BOOL,
MPI_LOR, comm);
    }
```

```
    //step 6: retrieve results back
    if (rank == 0)
        memcpy(dist, copyDist, copyN * sizeof(int));

    //step 7: remember to free memory
    free(copyMatrix);
    free(copyDist);

}

int main(int argc, char **argv) {
    const string inputFile = "../input.txt", outputFile = "../output.txt";
    if (argc <= 1) {
        cerr << "Please, specify number of processes";
        exit(1);
    } else p = stoi(argv[1]);

    //MPI initialization
    MPI_Init(&argc, &argv);
    MPI_Comm comm;

    int rank; // thread number
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &rank);

    //only rank 0 process do the I/O
    if (rank == 0) {
        if (argc <= 2) readMatrix(inputFile);
        else generateMatrix(stoi(argv[2]), matrix);
        N = stoi(argv[2]);
        dist = (int *) malloc(sizeof(int) * N);
    }

    //time counter
    double start, end;
    MPI_Barrier(comm);
    start = MPI_Wtime();

    performBellmanFord(rank, comm);
    MPI_Barrier(comm);

    end = MPI_Wtime();

    if (rank == 0) {
        std::cout << "Time: " << (end - start) * 1000 << " ms" << endl;
```

```
        outputResult(outputFile);
        free(dist);
        free(matrix);
    }
    MPI_Finalize();
    return 0;
}
```

Додаток 2 - код OpenMP імплементації

```
/*
* This is a openmp version of bellman_ford algorithm
* Compile: g++ -std=c++11 -o openmp_bellman_ford openmp_bellman_ford.cpp
* Run: ./openmp_bellman_ford <input file> <number of threads>, you will find the
output file 'output.txt'
* */

#include <string>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <chrono>

#include <omp.h>
#include "utils.hpp"

using namespace std;
using namespace chrono;

#define INF 1000000

int N; //number of vertices
int p = 10; // number of processes
int *matrix; // the adjacency matrix
int *dist;
bool hasNegativeCycle = false;

int calculateCoordinate(int x, int y, int n) {
   return x * n + y;
}

int readMatrix(const string &filename) {
   std::ifstream ifStream(filename, std::ifstream::in);
   ifStream >> N;
   matrix = (int *) malloc(N * N * sizeof(int));
   for (int i = 0; i < N; i++)
       for (int j = 0; j < N; j++) {
           ifStream >> matrix[calculateCoordinate(i, j, N)];
       }
   return 0;
}

int outputResult(const string &file) {
   std::ofstream outStream(file, std::ofstream::out);
   if (!hasNegativeCycle) {
       for (int i = 0; i < N; i++) {
           if (dist[i] > INF)
               dist[i] = INF;
           outStream << dist[i] << '\n';
       }
```

```
        outStream.flush();
    } else {
        outStream << "Negative cycle" << endl;
    }
    outStream.close();
    return 0;
}

void initScope(int *start, int *end, int range) {
#pragma omp parallel for
    for (int i = 0; i < p; i++) {
        start[i] = range * i;
        end[i] = range * (i + 1);
        if (i == p - 1) {
            end[i] = N;
        }
    }
}

void initDistances() {
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        dist[i] = INF;
    }
    dist[0] = 0;
}

void relaxDistances(int *start, int *end, bool *rangeChanged) {
    int threadNum = omp_get_thread_num();
    for (int u = 0; u < N; u++) {
        for (int v = start[threadNum]; v < end[threadNum]; v++) {
            int weight = matrix[calculateCoordinate(u, v, N)];
            if (weight < INF) {
                int newDistance = dist[u] + weight;
                if (newDistance < dist[v]) {
                    rangeChanged[threadNum] = true;
                    dist[v] = newDistance;
                }
            }
        }
    }
}

/**
* Bellman-Ford algorithm. Find the shortest path from vertex 0 to other vertices.
*/
void performBellmanFord() {
    int start[p], end[p];
    int range = N / p, iterNum = 0;

    omp_set_num_threads(p);
    initScope(start, end, range);
```

```
    initDistances();

    bool distanceChanged;
    bool rangeChanged[p];

#pragma omp parallel
    {
        int threadNum = omp_get_thread_num();
        for (int iteration = 0; iteration < N - 1; iteration++) {
            rangeChanged[threadNum] = false;
            relaxDistances(start, end, rangeChanged);
#pragma omp barrier
#pragma omp single
            {
                iterNum++;
                distanceChanged = false;
                for (int t = 0; t < p; t++) {
                    distanceChanged |= rangeChanged[t];
                }
            }
            if (!distanceChanged) {
                break;
            }
        }
    }

    // check negative cycles
    if (iterNum == N - 1) {
        distanceChanged = false;
        for (int u = 0; u < N; u++) {

#pragma omp parallel for reduction(|:distanceChanged)
            for (int v = 0; v < N; v++) {
                int weight = matrix[u * N + v];
                if (weight < INF) {
                    if (dist[u] + weight < dist[v]) {
                        // if we can relax one more step, then we find a negative
cycle
                        distanceChanged = true;
                    }
                }
            }
        }
        hasNegativeCycle = distanceChanged;
    }
}

/**
*
* @param argc
* @param argv 1 - number of processes, 2 - number of vertices
* (if specified, generates random graph)
```

```
* @return
*/
int main(int argc, char **argv) {
   const string inputFile = "../input.txt", outputFile = "../output.txt";
   if (argc <= 1) {
       cerr << "Please, specify number of processes";
       exit(1);
   } else p = stoi(argv[1]);

   if (argc <= 2) readMatrix(inputFile);
   else generateMatrix(stoi(argv[2]), matrix);

   N = stoi(argv[2]);

   dist = (int *) malloc(sizeof(int) * N);

   auto start = steady_clock::now();

   //bellman-ford algorithm
   performBellmanFord();

   auto end = steady_clock::now();
   long duration = duration_cast<nanoseconds>(end - start).count();

   cout << "Time: " << (double) duration / 1000000 << " ms" << endl;
   outputResult(outputFile);
   free(dist);
   free(matrix);
   return 0;
}
```

Додаток 3 - код послідовної імплементації

```
#include <string>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <chrono>

#include "utils.hpp"

using namespace std;
using namespace chrono;

#define INF 1000000

int N; //number of vertices
int *matrix; // the adjacency matrix
int *dist;
bool hasNegativeCycle = false;

int calculateCoordinate(int x, int y, int n) {
   return x * n + y;
}

int readMatrix(const string &filename) {
   std::ifstream ifStream(filename, std::ifstream::in);
   ifStream >> N;
   matrix = (int *) malloc(N * N * sizeof(int));
   for (int i = 0; i < N; i++)
       for (int j = 0; j < N; j++) {
           ifStream >> matrix[calculateCoordinate(i, j, N)];
       }
   return 0;
}

int outputResult(const string &file) {
   std::ofstream outStream(file, std::ofstream::out);
   if (!hasNegativeCycle) {
       for (int i = 0; i < N; i++) {
           if (dist[i] > INF)
               dist[i] = INF;
           outStream << dist[i] << '\n';
       }
       outStream.flush();
   } else {
       outStream << "Negative cycle" << endl;
   }
   outStream.close();
   return 0;
}

void initDistances() {
```

```
    for (int i = 0; i < N; i++) {
        dist[i] = INF;
    }
    dist[0] = 0;
}

void relaxDistances(bool& distanceChanged) {
    for (int u = 0; u < N; u++) {
        for (int v = 0; v < N; v++) {
            int weight = matrix[calculateCoordinate(u, v, N)];
            if (weight < INF) {
                int newDistance = dist[u] + weight;
                if (newDistance < dist[v]) {
                    distanceChanged = true;
                    dist[v] = newDistance;
                }
            }
        }
    }
}

void checkNegativeCycle() {
    for (int u = 0; u < N; u++) {
        for (int v = 0; v < N; v++) {
            int weight = matrix[calculateCoordinate(u, v, N)];
            if (weight < INF) {
                if (dist[u] + weight < dist[v]) {
                    // if we can relax one more step, then we find a negative
cycle
                    hasNegativeCycle = true;
                    return;
                }
            }
        }
    }
}

/**
* Bellman-Ford algorithm. Find the shortest path from vertex 0 to other vertices.
*/
void performBellmanFord() {
    initDistances();

    bool distanceChanged;
    for (int i = 0; i < N - 1; i++) {// n - 1 iteration
        distanceChanged = false;
        relaxDistances(distanceChanged);
        if (!distanceChanged) {
            return;
        }
    }
```

```
    //do one more iteration to check negative cycles
    checkNegativeCycle();
}

int main(int argc, char **argv) {
    const string inputFile = "../input.txt", outputFile = "../output.txt";

    if (argc <= 1) readMatrix(inputFile);
    else generateMatrix(stoi(argv[1]), matrix);

    N = stoi(argv[1]);

    dist = (int *) malloc(sizeof(int) * N);

    auto start = steady_clock::now();

    //bellman-ford algorithm
    performBellmanFord();

    auto end = steady_clock::now();
    long duration = duration_cast<nanoseconds>(end - start).count();

    cout << "Time: " << (double) duration / 1000000 << " ms" << endl;
    outputResult(outputFile);
    free(dist);
    free(matrix);
    return 0;
}
```

Додаток 4 - посилання на GitHub

https://github.com/annalieks/bellman-ford