|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра параллельных вычислительных технологий | | |
| Практическое задание № 10 | | |
| по дисциплине «Методы построения и анализа алгоритмов» | | |
| **Алгоритм Флойда** | | |
|  | | |
|  |  | АРИФУЛОВ ТИГРАН |
| Группа ПМ-01 | ДЫЧКО АРСЕНИЙ |
|  | ЖИЖЧЕНКО ЛЕОНИД |
|  | САМСОНОВ СЕМЕН |
|  | ЯКОВЛЕВА ЕЛЕНА |
|  |  |
| Преподаватель | домников пётр александрович |
|  |  |
| Новосибирск, 2021 | | |

# Задание:

Алгоритм Флойда. Поиск расстояний кратчайших путей между всеми парами вершин в взвешенном графе с распараллеливанием. Граф задается в файле в следующем формате. Первое число – количество вершин в графе. Второе число – количество ребер в графе. Далее перечислены ребра графа, заданные тройками чисел: первое число – начальная вершина ребра, второе число – конечная вершина ребра, третье число – вес ребра (натуральное число). На выходе файл с перечислением расстояний между всеми парами различных и имеющих между собой путь вершин.

# Текст программы:

#include <iostream>

#include <fstream>

#include <vector>

#include <omp.h>

#include <chrono>

#pragma region Trash)))

#define per(name, beg, en) for (int name = beg; name <= en; ++name)

#define rep(name, beg, en) for (int name = beg; name >= en; --name)

#define pb push\_back

#define mp make\_pair

#define sz(x) (int)(x).size()

#define F first

#define S second

#define FF first.first

#define FS first.second

#define SF second.first

#define SS second.second

#pragma endregion

using namespace std;

class Timer

{

private:

// Псевдонимы типов используются для удобного доступа к вложенным типам

using clock\_t = std::chrono::high\_resolution\_clock;

using second\_t = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<clock\_t> m\_beg;

double last\_elapsed;

public:

Timer() : m\_beg(clock\_t::now()), last\_elapsed(0)

{

}

void reset()

{

m\_beg = clock\_t::now();

last\_elapsed = 0.0;

}

double elapsed()

{

last\_elapsed = std::chrono::duration\_cast<second\_t>(clock\_t::now() - m\_beg).count();

return last\_elapsed;

}

inline double getLastElapsed()

{

return last\_elapsed;

}

};

using pii = pair<int, int>;

using GraphList = vector<vector<pii>>; // Граф списком смежности

using GraphMatrix = vector<vector<int>>; // Граф матрицей смежности

const int inf = INT\_MAX;

const pii pinf = mp(inf, inf);

/// <summary>

/// Находит ребро с конечной вершиной [v] в списке смежности [vec] от начальной вершины

/// </summary>

/// <param name="vec"> - список смежности для начальной вершины</param>

/// <param name="v"> - конечная вершина</param>

/// <returns></returns>

inline pii finder(const vector < pii >& vec, int v)

{

pii ans = pinf;

ans.F = v;

for (const auto& x : vec)

if (x.F == v && x.S < ans.S)

{

ans = x;

}

return ans;

}

inline void matrixPrinter(const GraphMatrix& graph, ostream& out)

{

size\_t size = graph.size();

per(u, 1, size - 1)

{

per(v, 1, size - 1)

out << ((graph[u][v] == inf) ? 0 : graph[u][v]) << " \t";

out << endl;

}

}

inline void listPrinter(const GraphMatrix& graph, ostream& out)

{

size\_t size = graph.size();

per(u, 1, size - 1)

{

per(v, 1, size - 1)

{

if (u != v && graph[u][v] != inf)

{

out << u << " \t" << v << " \t" << graph[u][v] << endl;

}

}

}

}

/// <summary>

/// Алгоритм Флойда от матрицы смежности

/// </summary>

/// <param name="sD"> Начальная матрица смежности, она же результат работы</param>

inline void \_floyd(GraphMatrix& sD, int numThreads = omp\_get\_max\_threads())

{

size\_t size = sD.size();

#pragma omp parallel for num\_threads(numThreads) schedule(dynamic, 200)

for (int i = 1; i < size; i++)

{

for (int u = 1; u < size; u++)

{

for (int v = 1; v < size; v++)

{

if (sD[u][i] < inf && sD[i][v] < inf)

sD[u][v] = min(sD[u][v], sD[u][i] + sD[i][v]);

}

}

}

}

void floydAlgMatrix(const GraphMatrix& graph, ostream& out, int numThreads = omp\_get\_max\_threads())

{

//GraphMatrix sD = graph;

GraphMatrix sD;

int size = static\_cast<int>(sz(graph));

sD.resize(size);

#pragma omp parallel for num\_threads(numThreads)

per(u, 1, size - 1)

{

sD[u].resize(size);

per(v, 1, size - 1)

if (graph[u][v] != 0)

sD[u][v] = graph[u][v];

else

sD[u][v] = inf;

}

\_floyd(sD, numThreads);

listPrinter(sD, out);

}

void floydAlgList(const GraphList& graph, ostream& out, int numThreads = omp\_get\_max\_threads())

{

GraphMatrix sD;

int size = static\_cast<int>(sz(graph));

sD.resize(size);

#pragma omp parallel for num\_threads(numThreads)

per(i, 1, size - 1)

{

sD[i].pb(0); // Заполняем нулевые элементы каждого ряда

per(j, 1, size - 1)

{

sD[i].pb(finder(graph[i], j).S);

}

}

\_floyd(sD, numThreads);

listPrinter(sD, out);

}

// Ввод данных и вызов методов

void solve(int num\_threads)

{

ifstream in("input.txt");

if (!in.is\_open()) throw exception("File input.txt wasn't open!");

int n, m;

in >> n >> m;

GraphMatrix matrixGraph; // граф матрицей смежности

GraphList listGraph; // граф списком смежности

listGraph.resize(n + 1);

matrixGraph.resize(n + 1);

per(i, 1, n)

matrixGraph[i].resize(n + 1, 0);

per(i, 1, m)

{

int u, v, w;

in >> u >> v >> w;

matrixGraph[u][v] = matrixGraph[v][u] = (matrixGraph[u][v] == 0 ? w : min(w, matrixGraph[u][v]));

listGraph[u].pb(mp(v, w));

listGraph[v].pb(mp(u, w));

}

in.close();

ofstream outMatrix("output.txt");

Timer timer;

floydAlgMatrix(matrixGraph, outMatrix, num\_threads);

timer.elapsed();

cout << "Алгоритм по матрице завершил работу за " << timer.getLastElapsed() << " секунд" << endl;

outMatrix.close();

ofstream outList("output.txt");

timer.reset();

floydAlgList(listGraph, outList, num\_threads);

timer.elapsed();

cout << "Алгоритм по листу завершил работу за " << timer.getLastElapsed() << " секунд" << endl;

outList.close();

}

int main()

{

setlocale(LC\_ALL, "ru");

vector<int> numThreadsList = { 1 };

for (int i = 2; i <= omp\_get\_max\_threads(); i += 2) numThreadsList.push\_back(i);

for (auto& numThreads : numThreadsList)

{

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "Работа алгоритмов для [" << numThreads << "] потоков: " << endl << endl;

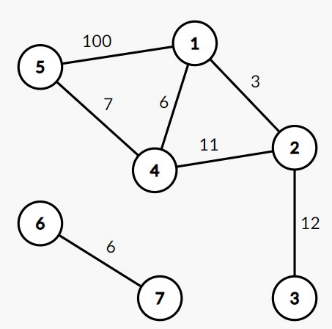
solve(numThreads);

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

}

}

# Результат работы программы:

1. Тест №1
   1. Входные данные:

7 7  
1 2 3  
1 4 6  
1 5 100  
2 3 12  
2 4 11  
4 5 7  
6 7 6

* 1. Выходные данные:

1 2 3

1 3 15

1 4 6

1 5 13

2 1 3

2 3 12

2 4 9

2 5 16

3 1 15

3 2 12

3 4 21

3 5 28

4 1 6

4 2 9

4 3 21

4 5 7

5 1 13

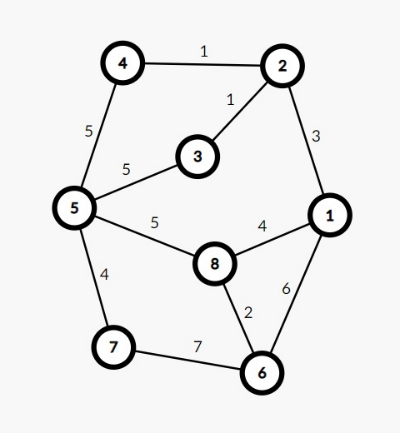
5 2 16

5 3 28

5 4 7

6 7 6

7 6 6

1. Тест №2
   1. Входные данные:

8 11  
4 2 1  
1 2 3  
7 5 4  
2 3 1  
8 6 2  
8 5 5  
5 4 5  
7 6 7  
3 5 5  
1 6 6  
8 1 4

* 1. Выходные данные:

1 2 3

1 3 4

1 4 4

1 5 9

1 6 6

1 7 13

1 8 4

2 1 3

2 3 1

2 4 1

2 5 6

2 6 9

2 7 10

2 8 7

3 1 4

3 2 1

3 4 2

3 5 5

3 6 10

3 7 9

3 8 8

4 1 4

4 2 1

4 3 2

4 5 5

4 6 10

4 7 9

4 8 8

5 1 9

5 2 6

5 3 5

5 4 5

5 6 7

5 7 4

5 8 5

6 1 6

6 2 9

6 3 10

6 4 10

6 5 7

6 7 7

6 8 2

7 1 13

7 2 10

7 3 9

7 4 9

7 5 4

7 6 7

7 8 9

8 1 4

8 2 7

8 3 8

8 4 8

8 5 5

8 6 2

8 7 9

# Тестирование параллельной версии

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Версия  алгоритма | Число вершин | Число рёбер | Время при числе потоков, секунды | | | | |
| 1 | 2 | 4 | 6 | 12 |
| По матрице | 8 | 11 | 0.0001963 | 0.0006798 | 0.0012466 | 0.0020218 | 0.0043522 |
| По листу | 0.0001696 | 0.0003446 | 0.0002084 | 0.0003712 | 0.0002408 |
| По матрице | 1000 | 917 | 0.799949 | 0.558214 | 0.341504 | 0.220841 | 0.290436 |
| По листу | 0.864348 | 0.458169 | 0.330309 | 0.226918 | 0.247627 |
| По матрице | 10000 | 9035 | 711.502 | 365.34 | 197.866 | 152.643 | 128.653 |
| По листу | 713.153 | 358.69 | 195.93 | 153.435 | 128.667 |

Тестирование проводилось на 6 ядерном процессоре с 12 потоками. В итоге по полученным данным стало ясно, что наибольшее ускорение программы достигается на 12 потоках, то есть на максимальном его числе. Причём на малых размерах использование параллельной версии лишь замедляет работу программы, тогда как на размерах, например, от 1000 вершин, появляется ощутимый прирост. Наиболее эффективный прирост скорости наблюдается при переходе к 2 потокам и к 4 потокам, дальше ускорение не настолько ощутимое, но всё ещё значимое.

# Сравнение с алгоритмом Дейкстры

Для сравнения с алгоритмом Дейкстры на аналогичных тестах, приведу таблицу тестирования алгоритма:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Число вершин | Число рёбер | Время при числе потоков, секунды | | | | | |
| 1 | 2 | 4 | 6 | 10 | 12 |
| 8 | 11 | 0.000125 | 0.000682 | 0.00112 | 0.001859 | 0.00275 | 0.004365 |
| 1000 | 917 | 0.007581 | 0.005004 | 0.005262 | 0.008784 | 0.007254 | 0.007904 |
| 10000 | 9035 | 0.298968 | 0.159570 | 0.105718 | 0.100554 | 0.076601 | 0.077286 |

При сравнении с алгоритмом Флойда на малых данных, разница во времени работы практически отсутствует. Но при переходе на большие данные, алгоритм Флойда начинает значительно уступать по скорости работы алгоритму Дейкстры.