МИНОБРНАУКИ РОССИИ

**Федеральное государственное автономное образовательное учреждение высшего образования**

**«Санкт-Петербургский политехнический университет Петра Великого»**

**Институт компьютерных наук и технологий**

**Кафедра «Распределенные вычисления и компьютерные сети»**

**КУРСОВАЯ РАБОТА**

**Параллельный алгоритм обхода графа в ширину**

по дисциплине «Параллельное программирование»

Выполнил

студент гр. 43507 <*подпись*> В.Б. Борисов

Руководитель

ст. преподаватель <*подпись*> Д.А. Тимофеев

Санкт-Петербург

2017

**Оглавление**

[**Оглавление** 2](#_Toc481415386)

[**Введение** 3](#_Toc481415387)

[**Основная часть** 4](#_Toc481415388)

[**Постановка задачи** 4](#_Toc481415389)

[**Ход работы** 4](#_Toc481415390)

[**Тестирование и результаты** 4](#_Toc481415391)

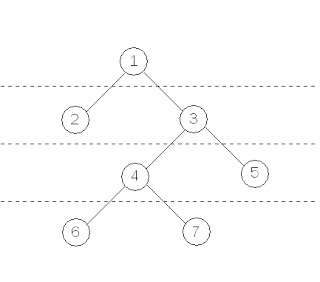
[**Реализация последовательного параллельного выполнения BFS на языке C++** 4](#_Toc481415392)

[**Заключение** 5](#_Toc481415393)

[**Список литературы** 5](#_Toc481415394)

# **Введение**

Поиск в ширину или сокращенно «BFS» - это метод, задача которого является обход всего дерева, начиная с узла-источника, путем последовательного просмотра всех узлов на заданном уровне перед переходом на следующий уровень.



Данный метод помогает решать классические задачи, такие как поиск кратчайшего пути. Для задач с малым количеством вершин «BFS» работает за малое количество времени и не требует лишних строчек кода, но что, если в поставленной задаче у нас, например, 10000 вершин, которые надо обойти за кратчайшее время. Зачастую для решения задач подобного рода последовательное выполнение алгоритма занимает намного больше времени, чем хотелось. Возникает потребность ускорить работу алгоритма и получить тот же результат, что и при обычной работе. Помочь в этом может параллельное программирование.

В современном мире компьютеры обладают многоядерными процессорами, а это значит, что мы можем организовать работу алгоритма так чтобы поставленную задачу, решал не один процесс, а несколько. В теории это должно повысить производительность и скорость работы. Например, если 1 процесс решает задачу за 60 минут, то 2 процесса решат за 30 минут, а 3 процесса за 20 минут и так далее.

# **Основная часть**

# **Постановка задачи**

В рамках данной курсовой работы необходимо:

* выбрать и изучить один из опубликованных параллельных алгоритмов для решения прикладных задач;
* реализовать этот алгоритм на любом языке программирования;
* провести экспериментальную оценку эффективности построенной реализации;
* описать в отчете постановку задачи, алгоритм, архитектуру программной реализации и принятые при ее создании решения, эксперимент и его результаты, включая их анализ.

# **Ход работы**

Выполнение данной курсовой работы и реализация алгоритма была выполнена в среде «Visual Studio» на языке C++.

Метод «BFS» был реализован при помощи очереди, которая сначала содержит только s-узел (начальная вершина), и множество посещенных вершин X, которое изначально тоже содержит только S. На каждом шаге алгоритм вынимает из начала очереди вершину, рассматривает все исходящие из нее ребра и добавляет все связанные с ней не посещённые вершины в X и в конец очереди. Если очередь пуста, то алгоритм завершает работу.

Чтобы распараллелить метод поиска в глубину, была использована технология Openmp, предназначенная для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

# **Тестирование и результаты**

Обход графа в ширину был написан так и последовательно, так и параллельно. В последовательном алгоритме нет ничего особенного, в нем всю работу делал один процесс. Параллельный код выполнялся на 4-х ядерном процессоре с частотой 2.5 GHz. И последовательный, и параллельный алгоритмы обходили один и тот же самый граф. Результатом обхода было количество ребер бинарного графа и время обхода. Каждый метод работал 10 раз (т.е. 10 тестов) независимо, после 10 тестов высчитывалось среднее время. Результаты тестирования обоих методов представлены в таблице ниже.

|  |  |  |
| --- | --- | --- |
| n-узлов | SerialBFS | ParallelBFS |
| 1000 | 0.0248 | 0.0184 |
| 5000 | 0.1144 | 0.081 |
| 10000 | 0.2187 | 0.1556 |
| 20000 | 0.4602 | 0.3139 |
| 50000 | 1.1274 | 0.7902 |
| 100000 | 2.2561 | 1.59 |

Исходя из полученной таблицы, мы видим, что параллельный метод обхода графа в ширину работает приблизительно в 1.4 раза быстрее, чем последовательный.

# **Реализация последовательного и параллельного выполнения BFS на языке C++**

* Последовательное выполнение

void Graph::BFS(int s)

{

cout << "Начинаем поиск в ширину с вершины ";

cout << s << " : " << endl;

cout << "Введите количество тестов: ";

cin >> coltest;

cout << "------------------------------------------------------------------------" << endl;

cout << "Работа последовательного BFS" << endl;

clock\_t start\_time = clock();

for (int counter = 0; counter < coltest; counter++){

int \*visited = new int[V];

list<int> queue;

queue.clear();

//cout << queue.max\_size();

for (int i = 0; i < V; i++)

visited[i] = 0;

queue.push\_back(s);

visited[s] = 1;

while (!queue.empty())

{

int u = queue.front();

//cout << u << " ";

queue.pop\_front();

for (list<int>::iterator i = adj[u].begin(); i != adj[u].end(); i++)

{

int vv = \*i;

if (visited[vv] == 0)

{

visited[vv] = 1;

queue.push\_back(vv);

if (vv > 0)

{

edges = edges + 1;

}

}

}

}

delete[] visited;

clock\_t end\_time = clock();

cout << endl;

result[counter] = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

sum += result[counter];

cout << endl << "Время затраченное на поиск в ширину: " << (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC << endl;

cout << "Количество ребер: " << edges << endl;

start\_time = clock();

edges = 0;

}

cout << endl << "------------------------------------------------------------------------" << endl;

cout << "Среднее арифметическое: " << sum / coltest << endl;

cout << endl << "------------------------------------------------------------------------" << endl;

sum = 0;

}

* Параллельное выполнение

void Graph::BFSparallelOpenMP(int s)

{

cout << "Начинаем поиск в ширину с вершины ";

cout << s << " : " << endl;

cout << "Введите количество тестов: ";

cin >> coltest;

cout << "------------------------------------------------------------------------" << endl;

cout << "Работа параллельного BFS" << endl;

for (int counter = 0; counter < coltest; counter++){

clock\_t start\_time = clock();

int \*visited = new int[V];

list <int> Q;

Q.clear();

for (int i = 0; i < V; i++)

visited[i] = 0;

Q.push\_back(s);

visited[s] = 1;

list<int>::iterator j;

while (!Q.empty())

{

int u = Q.front();

Q.pop\_front();

//cout << u << " ";

int size = adj[u].size();

j = adj[u].begin();

int vv;

#pragma omp parallel for

for (int k = 0; k < size; k++)

{

{

vv = \*j;

++j;

}

if (visited[vv] == 0)

{

visited[vv] = 1;

Q.push\_back(vv);

if (vv > 0)

{

edges = edges + 1;

}

}

}

}

clock\_t end\_time = clock();

result[counter] = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

sum += result[counter];

cout << endl;

cout << endl << "Время затраченное на поиск в ширину: " << (float)(end\_time - start\_time) / CLOCKS\_PER\_SEC << endl;

cout << "Количество ребер: " << edges << endl;

edges = 0;}

cout << endl << "------------------------------------------------------------------------" << endl;

cout << "Среднее арифметическое: " << sum / coltest << endl;

sum = 0;}

При помощи директивы #pragma omp parallel for мы сообщаем, что хотим распараллелить блок for. Так как у нас 4-х ядерная система и мы явно не указываем количество потоков, то делить блок for будет 4 потока, каждому потоку будет отдана своя часть цикла. Например, если у нас 8000 элементов, то в первом потоке переменная k принимает значения от 0 до 1999, во втором от 2000 до 3999, в третьем от 4000 до 5999, в четвертом от 6000 до 7999. Теоретически мы получаем ускорение в 4 раза. На практике ускорение будет меньше из-за необходимости создать потоки и дождаться их завершения. В конце параллельного региона выполняется барьерная синхронизация. Иначе говоря, достигнув конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.

# **Заключение**

В результате выполнения курсовой работы был реализован параллельный и последовательный алгоритм обхода в ширину. Были проведены тесты обоих методов независимо друг от друга. Была приведена таблица результатов, в которой отчетливо был виден прирост производительности. Параллельный алгоритм BFS примерно в 1.4 раза быстрей обходит граф, чем последовательный.

# **Список литературы**

[1] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri, BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems.

[2] Rudolf Berrendorf and Matthias Makulla, Level-Synchronous Parallel Breadth-First Search Algorithms For Multicore and Multiprocessor Systems.