

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
**Кафедра компьютерных систем и программных технологий**

**Отчет о лабораторной работе**

**Курс:** Параллельные вычисления

**Тема:** Создание многопоточных программ на языке C++ с  
использованием Pthreads и OpenMP

Выполнила студентка группы 13541/3

\_\_\_\_\_ Мельникова Д.Н.

Преподаватель

\_\_\_\_\_ Стручков И.В.

# Содержание

1 Постановка задачи.....	3
1.1 Индивидуальное задание.....	3
1.2 Программа работы.....	3
2 Сведения о системе.....	4
3 Структура проекта.....	4
4 Алгоритм решения.....	5
5 Распараллеливание алгоритма.....	6
OpenMP.....	6
Pthreads.....	7
6 Тестирование.....	7
Выводы.....	9

# 1 Постановка задачи

## 1.1 Индивидуальное задание

Вариант 5, OpenMP.

Поиск кратчайшего пути в ориентированном графе (алгоритм Беллмана-Форда).

## 1.2 Программа работы

1. Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
3. Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
4. Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
5. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
6. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
7. Сделать общие выводы по результатам проделанной работы:
  - Различия между способами проектирования последовательной и параллельной реализаций алгоритма.
  - Возможные способы выделения параллельно выполняющихся частей, возможные правила синхронизации потоков
  - Сравнение времени выполнения последовательной и параллельной программ.
  - Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной

## 2 Сведения о системе

Linux Mint-MATE 4.10.0-38-generic #42~16.04.1-Ubuntu SMP T x86\_64 GNU/Linux

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz
Stepping:              3
CPU MHz:               800.024
CPU max MHz:           3900.0000
CPU min MHz:           800.0000
BogoMIPS:              6624.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):    0-3
```

## 3 Структура проекта

```
.
├── bin
│   ├── openmpBF
│   ├── pthreadBF
│   └── strictBF
├── build.sh
├── graph.txt
├── mkGraph
├── mkGraph.hs
├── README.md
├── run.sh
└── src
    ├── main.cpp
    └── pt_main.cpp
```

Build.sh собирает файл генерации графа и бинарные файлы openmpBF, strictBF и pthreadBF:

```
ghc mkGraph.hs
```

```
./mkGraph $1
```

```
mkdir bin
```

```

g++ -std=c++11 -o bin/strictBF src/main.cpp
g++ -std=c++11 -fopenmp -o bin/openmpBF src/main.cpp
g++ -std=c++11 -pthread -o bin/pthreadBF src/pt_main.cpp
Run.sh запускает все три программы на сгенерированном графе:
echo "Simple running: "
./bin/strictBF graph.txt
echo "======"
echo "OpenMP running: "
./bin/openmpBF graph.txt
echo "======"
echo "pthreads running: "
./bin/pthreadBF graph.txt
echo "======"

```

## 4 Алгоритм решения

Реализованный алгоритм Беллмана-Форда выглядит следующим образом:

```

for (int i = 0; i < v_cnt - 1; i++)
{
    for (int j = 0; j < e_cnt; j++)
        if (d[ gr[j].start ] < imax)
        {
            if (d[ gr[j].end ] > d[ gr[j].start ] + gr[j].weight)
            {
                d[ gr[j].end ] = d[ gr[j].start ] + gr[j].weight;
                p[ gr[j].end ] = gr[j].start;
            }
        }
}

```

, где `gr` – вектор, содержащий граф в виде списка ребер, `d` – вектор, содержащий путь от стартовой вершины до всех остальных, `p` – это вектор для восстановления минимального пути.

Граф считывается из текстового файла вида:

```
v_cnt e_cnt start_v end_v  
[st_v e_v weight] * e_cnt,
```

где `v_cnt` – количество вершин, `e_cnt` – количество ребер, `start_v` и `end_v` – начало и конец искомого пути, далее следуют `e_cnt` описания ребер вида: начальная вершина – конечная вершина – вес ребра.

Такой граф генерируется программой `mkGraph`, реализованной на языке `Haskell`. Для ее сборки потребуется компилятор `ghc`. Она принимает один аргумент – `random seed` для числа вершин. Далее случайно выбирается количество ребер из промежутка  $[v\_cnt, v\_cnt^2]$ , также случайно выбираются начальная и конечная вершины пути и генерируется необходимое число ребер.

## 5 Распараллеливание алгоритма

### OpenMP

Как видно выше, алгоритм состоит из двух вложенных циклов, каждый из которых может быть распараллелен. Значение каждой итерации независимо от предыдущих вычислений.

Тогда, достаточно проставить директиву `#pragma omp parallel for` над циклом для распараллеливания:

```
for (int i = 0; i < v_cnt - 1; i++)  
{  
    #pragma omp parallel for  
    for (int j = 0; j < e_cnt; j++)  
        if (d[ gr[j].start ] < imax)  
        {  
            if (d[ gr[j].end ] > d[ gr[j].start ] + gr[j].weight)  
            {  
                d[ gr[j].end ] = d[ gr[j].start ] + gr[j].weight;  
                p[ gr[j].end ] = gr[j].start;  
            }  
        }  
}
```

## Pthreads

Распараллеливание посредством posix threads реализовано в файле src/pt\_main.cpp.

В данном случае, функция, выполняющаяся для каждого потока выносится отдельно (thread), в нее передается указатель типа void на аргументы, который в последствии приводится к нужному типу.

В начале функции bf выполнение разделяется на потоки по числу нод. В каждый поток передается метайнформация, разделяемая между потоками – упомянутые выше массивы gr, d и p. Это делается для того, чтобы не переполнять стек – для тестирования на больших графах.

## 6 Тестирование

Для подсчета статистики был написан скрипт test.sh:

```
for (( i=1; i<=50; i++))
do
    ./bin/strictBF graph.txt >> strict.txt
    ./bin/openmpBF graph.txt >> openmp.txt
    ./bin/pthreadBF graph.txt >> pthread.txt
    echo "Number $i"
done
```

, из данных которого составляется csv-файл (в представленном эксперименте количество нод – 724, количество ребер – 471158):

strict	openMP	pthreads
4576454567	1331799243	1202449632
4540244688	1282379805	1208763897
4758902073	1323430873	1202700361
4567873866	1487811279	1217789723
4555314594	1373529157	1200823256
4561594567	1329215508	1211354668
4574228537	1467435027	1312254890
4954066211	1368134889	1218227055
4675620352	1335111468	1207981701
4682392889	1387321044	1211711179
4552928855	1347174098	1213824028
4582020374	1325112120	1193810956
4500901950	1239172783	1195918439
4486198355	1258160397	1192378716
4446759903	1235519027	1196029117
4561609663	1259838694	1191838854
4528772411	1280324518	1200645941
4475746933	1268828592	1198571262
4473219699	1297480366	1197011315
4469451674	1283939939	1189830376
4496813786	1245921014	1206798494
4552441034	1322954757	1196554167
4546966585	1292564279	1194749184
4518531637	1280177862	1184134853
4504410606	1298994573	1186980682
4475005526	1218604363	1200432083
4456327871	1245672588	1188711951
4486547301	1232375756	1180930596
4464461470	1300050383	1196684683
4467572884	1293470950	1184144462
4469511218	1265015127	1202513267
4515831956	1320827282	1203068684
4515834191	1288414202	1187769915
4540850014	1398756271	1206875957
4647410551	1406472311	1209239902
4492449858	1279802408	1197844013
4431973550	1203482933	1177990585
4419415082	1208802806	1185742267
4458415968	1241850340	1190132076
4492836990	1212199997	1184520188
4424769802	1242458629	1192000959
4454666089	1208845951	1184954909
4547670080	1272721262	1190895866
4599883306	1211675870	1188174328
4462937941	1272491232	1187044784
4471666856	1247390321	1184318645
4499736277	1492301327	1194416082
4502073038	1443030498	1218160978
4649380356	1355421620	1198593231
4661227741	1454157810	1223645138



	Strict	OpenMP	Pthreads
Среднее значение	4504866598	1292424617	1193700781
Дисперсия	94586321.7	75263931.4	19486868.1
Доверительный интервал (0.05)	26217509.0029086	20861714.076565	5401385.00490997

## Выводы

OpenMP быстрее последовательного алгоритма в 3.486 раз, pthread – в 3.773, как показали эксперименты на 4-х ядерной машине.

Как показывает статистика, наиболее быстрым решением являются POSIX threads: у них минимальные средние временные затраты на подсчет и дисперсия.