Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра алгоритмических языков

Отчёт о выполнении задания практикума

«Исследование масштабируемости графовых алгоритмов для различных реализаций и различных компьютерных платформ»

Студенты 624 группы
А. К. Асирян
Е. В. Галкина
Вариант 26

Москва, 2017

Формулировка задания

Требуется исследовать масштабируемость алгоритма поиска в ширину, реализованного в библиотеке Parallel Boost Graph Library (PBGL). Для тестирования используются сгенерированные графы типов RMAT и SSCA2. Задание выполняется на суперкомпьютере IBM Blue Gene/P.

Масштабируемость определяется как зависимость производительности от размеров задачи и числа использованных процессоров (ядер). В качестве метрики для оценки производительности используется метрика TEPS (Traversed Edges Per Second) — число ребер графа, которое алгоритм обрабатывает за одну секунду.

В результате выполнения задания для каждого из двух типов графов (RMAT и SSCA2) требуется:

- построить график зависимости производительности от числа используемых для вычисления процессоров/нитей и размера графа;
- найти параметры, при которых получается максимальное значение TEPS.

Настройка библиотеки, опции компиляции и платформа тестирования

При выполнении задания использовалась библиотека `boost` версии 1.65.1. Для компиляции программы использовался компилятор `mpicxx` с версией `gcc` 4.1.2 с опцией `-O3`.

Вычисления проводились на системе IBM Blue Gene/P. Система состоит из двух стоек, включающих 8192 процессорных ядер (2 * 1024 четырехъядерных вычислительных узлов), с пиковой производительностью 27,9 ТФлопс.

Характеристики вычислительного узла:

- четыре микропроцессорных ядра;
- пиковая производительность: 13,6 ГФлопс;
- пропускная способность памяти: 13,6 ГБ/с;
- 2 ГБ общей памяти;
- 2 * 4 МБ кэш-памяти 2-го уровня;
- легковесное ядро, представляющее собой Linux-подобную операционную систему, поддерживающую значительное подмножество Linux-совместимых системных вызовов;

- асинхронные операции межпроцессорных обменов (выполняются параллельно с вычислениями);
- операции ввода-вывода перенаправляются I/O-картам через сеть коллективных операций;
- MPI-операции типа «точка-точка» осуществляются через сеть трехмерного тора.
  - вычислительный узел имеет двунаправленные связи с шестью соседями;
  - пропускная способность каждого соединения — 425 МБ/с (5,1 ГБ/с для всех 12 каналов);
  - латентность (ближайший сосед):
    * 32-байтный пакет — 0,1 мкс;
    * 256-байтный пакет — 0,8 мкс.

Генерация тестовых данных

При выполнении задания использовался параллельный генератор RMAT/SSCA2-графов. Его компиляция производилась командой `g++ generator.cpp -O3 -o generator -fopenmp`.

Генератор использует следующие флаги:

- `-s N` — определяет масштаб графа, задает общее число вершин в генерируемом графе, равное 2^N. По умолчанию N установлено равным 15, в результате чего генерируется граф с 32 тыс. вершин;
- `-e M` — определяет среднюю степень связанности каждой вершины, определяет суммарное число ребер в графе, которое будет равно $M * 2^N$. По умолчанию M равно 32;
- `-[un]directed` — определяет, генерируется ориентированный или неориентированный граф. В случае ориентированного графа все дуги произвольны, в случае неориентированного дуги имеют вид (src_id, dst_id, weight), причем для каждой дуги src_id $\leq$ dst_id;
- `-[un]weighted` — определяет наличие весов для каждой дуги в графе. В случае взвешенного графа (weighted), каждая дуга определяется тройкой чисел (src_id, dst_id, weight), в случае невзвешенного — парой (src_id, dst_id);

- `-t T` — число openMP потоков, используемых для генерации графа. По умолчанию данное значение устанавливается равным результату вызова функции `omp_get_max_threads()`;

- `-file NAME` — задаёт имя выходного файла, куда будет сохранен сгенерированный граф. По умолчанию выходной файл называется `graph_N.bin`, где N - масштаб графа;

- `-type TYPE` —задаёт тип генерируемого графа. Значение `TYPE` может быть быть равно `RMAT` или `SSCA2`.

Для выполнения задания тестовые данные были сгенерированы следующим образом:

```
generator -s N -e 32 -undirected -unweighted -t 4 \
    -file {RMAT,SSCA2}_N.graph -type {RMAT,SSCA2}
```

где `N` — масштаб графа. Тогда число рёбер графа вычисляется следующим образом: $32 * 2^N = 2^{N+5}$.

Число вершин графа было выбрано от 2^{12} до 2^{19} для графов типа RMAT и до 2^{20} для SSCA2. Генерация графов бóльшего размера не производится, поскольку возникают проблемы при их обработке на суперкомпьютере. Задача снимается из очереди, поскольку работает дольше разрешённого времени на 4, 8, 16 и 32 процессорах. Ограничение по времени для этого числа процессоров — 1 час. При запуске на 1 и 2 процессорах программа аварийно завершается с ошибкой при выделении памяти:

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  St9bad_alloc
  <Nov 25 12:04:25.585224> BE_MPI (ERROR):
    The error message in the job record is as follows:
  <Nov 25 12:04:25.585309> BE_MPI (ERROR):
    "killed with signal 6"
```

Таким образом, генерация графов бóльшего размера нецелесообразна, поскольку время работы может быть подсчитано только на 64 и 128 процессорах.

Пример использования библиотечной функции алгоритма поиска в ширину

Для тестирования была реализована следующая программа, использующая функцию `breadth_first_search` из библиотеки PBGL:

```
#include <boost/graph/use_mpi.hpp>
#include <boost/graph/distributed/mpi_process_group.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/distributed/adjacency_list.hpp>
#include <boost/graph/distributed/graphviz.hpp>
#include <fstream>
#include <string>
#include <iostream>

using namespace boost;

using graph::distributed::mpi_process_group;

typedef adjacency_list<vecS, distributedS<mpi_process_group, vecS>,
    undirectedS, property<vertex_color_t, default_color_type >>Graph;

int main(int argc, char* argv[]) {
    mpi::environment env(argc,argv);

    int vertices_pow = atoi(argv[1]);
    int vertices_count = pow(2.0, vertices_pow);
    long long edges_count = 0;
    char *file_name = argv[2];

    Graph g(vertices_count);

    if (process_id(process_group(g)) == 0) {
        std::fstream file(file_name, std::ios::in | std::ios::binary);

        vertices_count = 0;
        edges_count = 0;
        file.read((char*)(&vertices_count), sizeof(int));
        file.read((char*)(&edges_count), sizeof(long long));

        for(int i = 0; i < edges_count; i++) {
            int src_id = 0, dst_id = 0;

            file.read((char*)(&src_id), sizeof(int));
            file.read((char*)(&dst_id), sizeof(int));

            add_edge(vertex(src_id, g), vertex(dst_id, g), g);
        }

        file.close();
    }

    graph_traits<Graph>::vertex_descriptor start = vertex(0, g);
    property_map<Graph, vertex_color_t>::type colorMap
        = get(vertex_color, g);
```

```
    property_map<Graph, vertex_index_t>::type index
        = get(vertex_index, g);

    double t1 = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD);
    breadth_first_search(g, start, color_map(colorMap));
    MPI_Barrier(MPI_COMM_WORLD);
    double t2 = MPI_Wtime();

    if (process_id(process_group(g)) == 0) {
      std::cout << edges_count / ((t2 -t1) * 1e6)
                << " MTEPS\n" << t2 - t1 << " seconds\n";
    }
    MPI_Barrier(MPI_COMM_WORLD);

    Graph::vertex_iterator it, itEnd;
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int start_index
        = process_id(process_group(g)) *
            (int) ceil(vertices_count * 1.0 / size);

    for (tie(it, itEnd) = vertices(g); it != itEnd; it++) {
        if (colorMap[*it] == 4) {
            std::cout << start_index + index[*it] << " ";
        }
    }

  return 0;
}
```

Компиляция производилась следующим образом:

```
mpicxx -O3 -Iboost/ task1_par.cpp -o task1_par -Lboost/stage/lib \
    -lboost_graph_parallel -lboost_mpi -lboost_serialization -lboost_system
```

Проверка корректности работы программы

Для проверки корректности работы параллельной программы её результаты сравнивались с результатами работы последовательного алгоритма на тех же входных данных. Ниже приведён код последовательной программы:

```
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graphviz.hpp>
#include <fstream>
#include <string>
```

```
#include <iostream>

using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS,
    property<vertex_color_t, default_color_type > >Graph;

int main(int argc, char* argv[]) {
    int vertices_pow = atoi(argv[1]);
    int vertices_count = pow(2.0, vertices_pow);
    long long edges_count = 0;
    char *file_name = argv[2];

    Graph g(vertices_count);

    std::fstream file(file_name, std::ios::in | std::ios::binary);

    vertices_count = 0;
    edges_count = 0;
    file.read((char*)(&vertices_count), sizeof(int));
    file.read((char*)(&edges_count), sizeof(long long));

    for(int i = 0; i < edges_count; i++) {
        int src_id = 0, dst_id = 0;

        file.read((char*)(&src_id), sizeof(int));
        file.read((char*)(&dst_id), sizeof(int));

        add_edge(vertex(src_id, g), vertex(dst_id, g), g);
    }
    file.close();

    graph_traits<Graph>::vertex_descriptor start = vertex(0, g);
    property_map<Graph, vertex_color_t>::type colorMap
        = get(vertex_color, g);
    property_map<Graph, vertex_index_t>::type index
        = get(vertex_index, g);

    breadth_first_search(g, start, color_map(colorMap));

    Graph::vertex_iterator it, itEnd;

    for (tie(it, itEnd) = vertices(g); it != itEnd; it++) {
        if (colorMap[*it] == 4) {
            std::cout << index[*it] << " ";
        }
    }

    return 0;
```

```
}
```

Для компиляции программы с последовательным алгоритмом использовалась команда:

```
mpicxx -O3 -Iboost/ task1_seq.cpp -o task1_seq -Lboost/stage/lib \
    -lboost_graph -lboost_serialization -lboost_system
```

Результаты работы последовательной и параллельной программ сохраняются в файлы. Для проверки совпадения результатов используется скрипт на Python:

```
#!/usr/bin/python
import glob
from collections import defaultdict
from os import path

from joblib import Parallel
from joblib import delayed


def get_stats(file):
    mteps = time = results = None
    with open(path.join('results', '%s.txt' % file), mode='r',
                encoding='utf8') as file:
        for line in file:
            if 'MTEPS' in line:
                mteps = float(line.split()[0])
            elif 'seconds' in line:
                time = float(line.split()[0])
            else:
                results = [int(x) for x in line.split()]
    return [x for x in [mteps, time, results]]

if __name__ == '__main__':
    files = [path.splitext(path.basename(file))[0] for file in \
        glob.glob('results/*.txt')]
    stats = defaultdict(lambda: defaultdict(lambda: defaultdict(lambda: \
        defaultdict(list))))
    files_stats = Parallel(n_jobs=-1, verbose=-10) \
        (delayed(get_stats)(file) for file in files)
    for file, file_stats in zip(files, files_stats):
        info = file.split('_')
        if len(info) > 2:
            type, pow, nproc = info[:3]
            nproc = int(nproc)
        else:
            type, pow = info
```

```
            nproc = 1
        pow = int(pow)

        if len(info) > 2:
            stats[type][pow][nproc]['mteps'].append(file_stats[0])
            stats[type][pow][nproc]['time'].append(file_stats[1])
            stats[type][pow][nproc]['result'].append(file_stats[2])
        else:
            stats[type][pow][nproc]['result'].append(file_stats[0])

type_info = defaultdict(dict)
for type, type_results in sorted(stats.items()):
    type_info[type]['pows'] = sorted(type_results)
    type_info[type]['nprocs'] =
        sorted(list(type_results.values())[0])[1:]

with open('stats.tsv', mode='w', encoding='utf8') as file:
    for type, type_results in sorted(stats.items()):
        check = True
        file.write('%s\n' % type)
        for x in ['mteps', 'time']:
            file.write('%s\n' % x.upper())
            file.write('\t'.join(['']
                + [str(x) for x in type_info[type]['nprocs']]))
            file.write('\n')
            for pow in type_info[type]['pows']:
                file.write('\t'.join([str(pow)]
                  + [('%0.6f' % (sum(y) / len(y))).replace('.', ',')
                      for y in [type_results[pow][nproc][x]
                          for nproc in type_info[type]['nprocs']]]))
                file.write('\n')

                if check:
                    pow_results = type_results[pow]
                    for nproc in type_info[type]['nprocs']:
                        nproc_results = pow_results[nproc]
                        if nproc != 1:
                            assert len(nproc_results['result']) == 3
                            assert len(nproc_results[x]) == 3

                            correct = set(pow_results[1]['result'])
                            assert len(correct) \
                                == len(pow_results[1]['result'])
                            for result in nproc_results['results']:
                                answer = set(result)
                                assert len(answer) == len(result)
                                assert answer == correct
            check = False
```

Исследование масштабируемости

Для оценки производительности компьютера использовалась метрика TEPS (Traversed Edges Per Second) — число ребер графа, которое алгоритм обрабатывает за одну секунду. Её также можно посчитать как отношение общего числа рёбер в графе ко времени работы алгоритма (в секундах) на графе данного размера.

Как было сказано выше, общее число рёбер графа равно $32 * 2^N$, где N — масштаб графа, изменяющийся от 12 до 19.

Таким образом, значение TEPS может быть подсчитано по следующей формуле: $TEPS = \frac{32*2^N}{T}$, где T — время выполнения в секундах. Замер времени осуществлялся с помощью функции `MPI_Wtime` с предварительной синхронизацией с помощью `MPI_Barrier`.

По результатам тестирования были построены графики, приведённые на рисунках 1 и 2 для графов типа RMAT и SSCA2 соответственно:

- ось X — число используемых программой MPI-процессов (2, 4, 8, 16, 32, 64, 128);
- ось Y — степень числа вершин графа, от 12 до 19 или 20 в зависимости от типа графа;
- ось Z — производительность программы в MTEPS.

Выводы и результаты

Максимальная производительность для графов обоих типов достигается на 128 процессорах и на графах с масштабом 19. Использовать бóльшее число процессоров не представляется возможным, поскольку программа завершается аварийно с ошибкой при выделении памяти: `MPI_Alloc_mem: Unable to allocate memory for MPI_Alloc_mem`.

На графике для RMAT (рис. 1) заметно снижение производительности для масштаба 12 на 64-х и 128-ми процессорах. Это может быть связано с высокой степенью накладных расходов при пересылке данных между процессорами.

Значительный рост производительности при увеличении числа процессов для графов типа SSCA2 наблюдается при масштабе больше 15-ти и растёт с увеличением масштаба вплоть до 19-ти (рис. 2). Однако при переходе значения степени от 19 к 20 наблюдается снижение производительности при фиксированном числе процессоров. Такое поведение можно объяснить тем, что величина MTEPS представляет собой отношение числа рёбер графа, которое

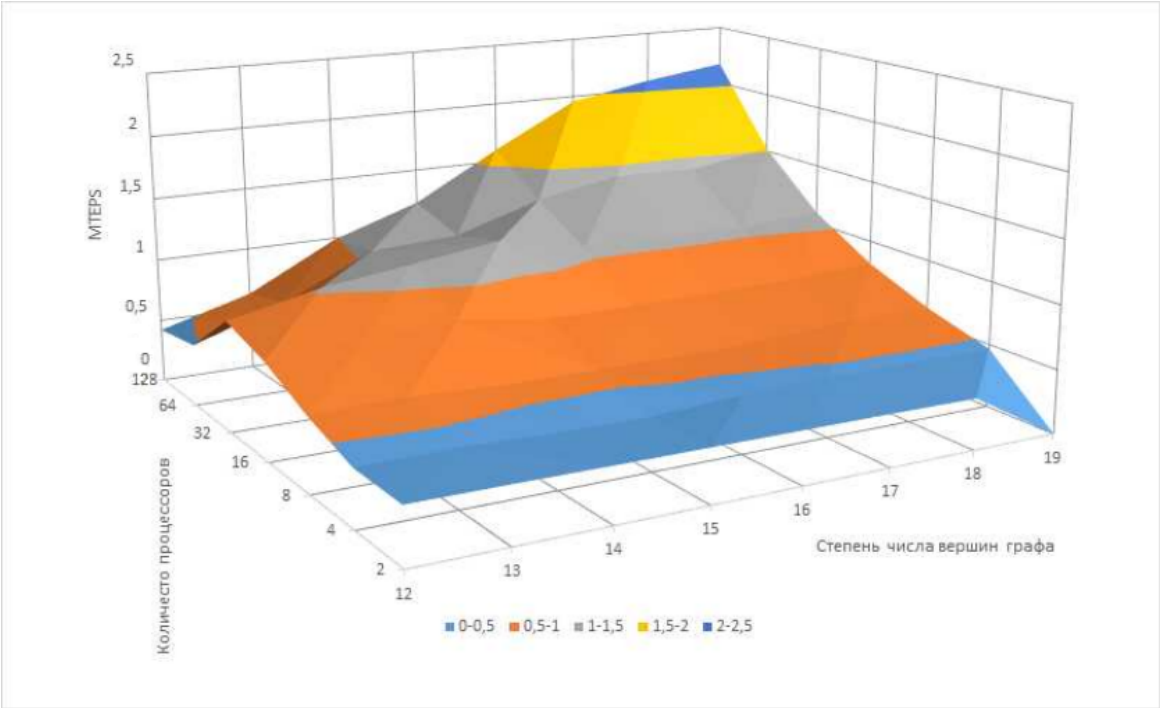

Рис. 1: График масштабируемости для графов типа RMAT

растёт экспоненциально в зависимости от степени вершин, ко времени работы программы. В данной ситуации увеличение числа рёбер очень велико, но значительно уменьшить время работы не представляется возможным при имеющихся ограничениях на число процессоров. При этом для фиксированной степени числа вершин графа, равной 20-ти, производительность растёт с ростом числа процессоров.

Достигнутая максимальная производительность для графов типа RMAT равна 2.17 MTEPS, для графов типа SSCA2 — 5.25 MTEPS. Такое низкое значение может быть связано с ограничениями на количество используемых процессоров. Алгоритм поиска в ширину обходит граф по слоям, а число параллельных операций на каждом слое равно сумме количества смежных вершин для каждой вершины слоя. Это число различно для каждого слоя и может быть довольно низким, поскольку количество вершин в каждом слое зависит от структуры графа.

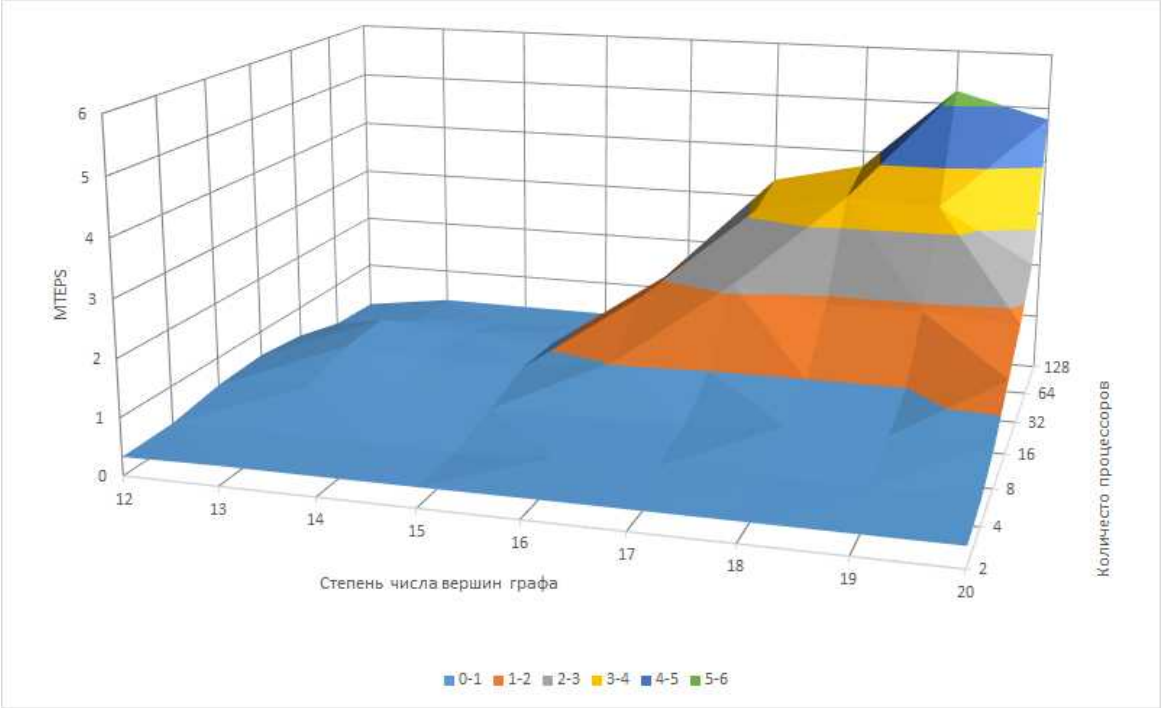

Рис. 2: График масштабируемости для графов типа SSCA2