Отчёт по практическому заданию по курсу «Суперкомпьютерное моделирование и технологии программирования»

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

Аят Оспанов

617 группа, ММП, ВМК МГУ, Москва

21 ноября 2017 г.

Содержание

1 Формулировка задания

В данном задании будет исследована масштабируемость графовых алгоритмов на примере **алгоритма PageRank** из библиотеки **Parallel Boost Graph Library**. Масштабируемость определяется как зависимость производительности от размеров задачи и числа использованных процессоров (ядер).

В данном задании предполагается, что графовые алгоритмы оперируют с графами в их простейшем определении: каждый граф – это совокупность заданного числа вершин и ребер. Графы бывают взвешенные (каждому ребру соответствует некоторое значение) и невзвешенные (каждому ребру соответствует только пара вершин, которые оно соединяет). В случае взвешенного графа каждое ребро определяется как тройка чисел (вершина-начало ребра, вершина-конец ребра, а также вес ребра).

В качестве основной метрики для оценки производительности целевого компьютера на реализациях графовых алгоритмов используется TEPS (Traversed Edges Per Second) – число ребер графа, которое алгоритм обходит (обрабатывает) за одну секунду. Для удобства, часто также используются метрики GTEPS и MTEPS с приставками из системы СИ.

Таким образом, производительность компьютера на реализации графового алгоритма можно вычислить по следующей формуле:

$$\text{Производительность(TEPS)} = \frac{\text{число ребер графа}}{\text{время выполнения реализации алгоритма в секундах}} \quad (1)$$

Данная метрика позволяет сравнивать производительность целевого компьютера на реализации графового алгоритма для графов различных размеров, а также для различного числа используемых при вычислениях процессоров. Обычно при увеличении размера графа производительность реализации может заметно снижаться по причине того, что данные перестают попадать в различные уровни кэш-памяти. Данная зависимость крайне интересна, так как графы реального мира имеют всё больший и больший размер, поэтому эффективность реализаций даже для очень больших размеров графов крайне важна. Кроме того, ситуация может меняться и при увеличении числа используемых процессоров.

2 Платформа тестирования

В качестве платформы для выполнения задания применялся вычислительный комплекс IBM Blue Gene/P.

2.1 IBM Blue Gene/P

IBM Blue Gene/P – массивно-параллельная вычислительная система, которая состоит из двух стоек, включающих 8192 процессорных ядер (2 x 1024 четырехъядерных вычислительных узлов), с пиковой производительностью 27,9 терафлопс (27,8528 триллионов операций с плавающей точкой в секунду).

Характеристики системы:

- две стойки с вычислительными узлами и узлами ввода-вывода
- 1024 четырехъядерных вычислительных узла в каждой из стоек
- 16 узлов ввода-вывода в стойке (в текущей конфигурации активны 8, т.е. одна I/O-карта на 128 вычислительных узлов)
- выделенные коммуникационные сети для межпроцессорных обменов и глобальных операций
- программирование с использованием MPI, OpenMP/pthreads, POSIX I/O
- высокая энергоэффективность: 372 MFlops/W (см. список Green500)
- система воздушного охлаждения

Стойка (rack, cabinet) состоит из двух midplane'ов. В midplane входит 16 node-карт (compute node card), на каждой из которых установлено 32 вычислительных узла (compute card). Midplane, 8 x 8 x 8 = 512 вычислительных узлов, – минимальный раздел, на котором становится доступна топология трехмерного тора; для разделов меньших размеров используется топология трехмерной решетки. Node-карта может содержать до двух узлов ввода-вывода (I/O card). Вычислительный узел включает в себя четырехъядерный процессор, 2 ГБ общей памяти и сетевые интерфейсы.

Микропроцессорное ядро:

- модель: PowerPC 450
- рабочая частота: 850 MHz
- адресация: 32-битная
- кэш инструкций 1-го уровня (L1 instruction): 32 KB
- кэш данных 1-го уровня (L1 data): 32 KB
- кэш предвыборки (L2 prefetch): 14 потоков предварительной выборки (stream prefetching): 14 x 256 байтов
- два блока 64-битной арифметики с плавающей точкой (Floating Point Unit, FPU), каждый из которых может выдавать за один такт результат совмещенной операции умножения-сложения (Fused Multiply-Add, FMA)
- пиковая производительность: 2 FPU x 2 FMA x 850 MHz = 3,4 GFlop/sec per core

Вычислительный узел:

- четыре микропроцессорных ядра PowerPC 450 (4-way SMP)

- пиковая производительность: 4 cores x 3,4 GFlop/sec per core = 13,6 GFlop/sec
- пропускная способность памяти: 13,6 GB/sec
- 2 ГБ общей памяти
- 2 x 4 МБ кэш-памяти 2-го уровня (в документации по BG/P носит название L3)
- легковесное ядро (compute node kernel, CNK), 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

3 Настройка среды

Для выполнения задания требуется библиотека **boost**. Для выполнения данного задания использовалась библиотека **boost** версии 1.47.

3.1 Сборка библиотеки boost

1. Скачивание библиотеки на платформу
   ```
   wget --no-check-certificate -c
   https://sourceforge.net/projects/boost/files/
   →boost/1.47.0/boost_1_47_0.tar.gz/download
   ```
2. Распаковка
   ```
   tar -xf boost147_0.tar.gz
   ```
3. Настройка установщика
   ```
   ./bootstrap.sh --prefix="../boost_install" --with-toolset=gcc
   --with-libraries=mpi,filesystem,system,serialization,graph,graph_parallel
   ```
4. Добавление строк в файл `<boost_root>/tools/build/v2/user-config.jam`:
   ```
   using gcc : 4.1.2 : mpicxx : ;
   using mpi : /bgsys/drivers/ppcfloor/comm/bin/mpicxx ;
   ```
5. Сборка и установка:
   ```
   ./b2 --prefix="../boost_install" --layout=versioned
   toolset=gcc variant=release install
   ```

3.2 Компиляция и запуск программы

Компиляция самой программы и вспомогательных программ (будет описано позже) проводилась следующим makefile-ом.

```
HOST_CC=mpicxx

FLAGS=-O3
INCLUDES=-I$(HOME)/libs/boost/include
LIBS=-L$(HOME)/libs/boost/lib -lboost_system -lboost_mpi -lboost_serialization -lboost_graph_parallel

SOURCE=code.cpp
OUT=run

all:
	$(HOST_CC) $(FLAGS) $(INCLUDES) -o $(OUT) $(SOURCE) $(LIBS)

st:
	g++ $(FLAGS) $(INCLUDES) -o run_st code_st.cpp $(LIBS)

generator:
	xlc++_r -qsmp=omp -O3 generator.cpp -o generator

checker:
	g++ check.cpp -o check
```

Запуск производился следующим bash файлом.

```
#!/bin/bash
set -e

source params.sh

OUT_PATH=../out
TYPE=${TYPES[$1]}

echo ${TYPE}

for P in $(seq ${P_min} ${P_step} ${P_max})
do
    FILE_NAME=${DATA_PATH}graph_${TYPE}_${P}.bin
    for N in ${Ns[@]}
    do
        OUT_FILE=${P}_${N}_${TYPE}
        $MPIRUN -n ${N} --stdout ${OUT_PATH}/${OUT_FILE}.out --stderr ${OUT_PATH}/${OUT_FILE}.err -w 00:10:00
        ↪  ${PROG} -- ${FILE_NAME} ${P}
        echo -e ""
    done
done
```

Где `params.sh`

```
#!/bin/bash

# Graph sizes
P_min=14
P_max=18
P_step=1

# Number of MPI processes
Ns=(1 2 4 8 16 32 64 128)
```

```
# Types of graphs
TYPES=("RMAT" "SSCA2")

DATA_PATH=../data/

PROG=./run
MPIRUN=mpisubmit.bg
```

4 Генерация данных

Для тестирования были сгенерированы ориентированные невзвешанные графы типа RMAT и SSCA2 с числом вершин $2^{14} - 2^{18}$. Выбор таких размеров был сделан с учетом того, что оперативная память, доступная одному узлу ограничивается 2ГБ, что затрудняет чтение больших файлов, а само чтение производилось из корневого процесса. Генерация файлов производилась с помощью приведенного ниже bash файла, который запускает предварительно скомпилированный с помощью makefile-а (см. раздел 3.2) генератор:

```
#!/bin/bash
set -e

source params.sh

make generator

for TYPE in ${TYPES[@]}
do
    for P in $(seq ${P_min} ${P_step} ${P_max})
    do
        FILE_NAME=${DATA_PATH}graph_${TYPE}_${P}.bin
        echo Generating ${P} ${TYPE}
        ./generator -s ${P} -directed -unweighted -file ${FILE_NAME} -type ${TYPE}
    done
done
```

Средняя степень связанности каждой вершины, определяющее суммарное число ребер в графе, был оставлен по умолчанию и равна 32. Т.к. мы тестируем алгоритм PageRank, такой выбор был сделан с целью имитации графа сети интернет, который является достаточно разреженным.

5 Пример использования библиотечной функции алгоритма PageRank

Для исследования алгоритма был написан следующий код, использующий библиотеку boost. Данный код содержит не только запуск алгоритма, но также сбор данных со всех процессов для дальнейшей записи результатов в файл. Данные результаты также пригодятся для проверки корректности результатов работы параллельной версии с последовательной.

```cpp
#include <iostream>

#include <boost/graph/use_mpi.hpp>
#include <boost/graph/distributed/mpi_process_group.hpp>
#include <boost/graph/distributed/page_rank.hpp>
#include <boost/algorithm/string.hpp>

#include "graph.cpp"

using namespace boost;
using boost::graph::distributed::mpi_process_group;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cout << "Параметры: <входной файл> <масштаб графа>" << std::endl;
        return 1;
    }

    // Инициализация MPI среды
    mpi::environment env(argc,argv);
    boost::mpi::communicator world;

    // Получение имени входного файла с графом и числа вершин входного графа
    std::string file_name = argv[1];
    unsigned int vertices_count = 1U << strtol(argv[2], NULL, 10);
    long long edges_count = 0;

    // Создаем граф с заданным числом вершин
    Graph g(vertices_count);

    // Читаем граф на корневом процессе
    if (process_id(process_group(g)) == 0) {
        read_graph(file_name, g, vertices_count, edges_count);
    }

    // Считаем PageRank с замером времени выполнения
    std::vector<double> ranks(num_vertices(g));

    double t1 = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD);
    graph::page_rank(g, make_iterator_property_map(
            ranks.begin(), get(vertex_index, g)));
    MPI_Barrier(MPI_COMM_WORLD);
    double t2 = MPI_Wtime();

    // Печатаем производительность с корневого процесса
    // и собираем со всех процессов результаты и сохраняем
    if (process_id(process_group(g)) == 0) {
        std::cout << "Performance: " << 1.0 * edges_count / ((t2 - t1) * 1e3)
                  << " KTEPS" << std::endl;
        std::cout << "Time: " << t2 - t1 << " seconds" << std::endl;

        std::vector<std::vector<double> > all_ranks;

        mpi::gather(world, ranks, all_ranks, 0);

        std::vector<std::string> strs;
        split(strs, file_name, is_any_of("/"));
        std::string fname = "../res/" + strs[strs.size()-1] + "_" +
                lexical_cast<std::string>(process_group(g).size) + ".mp_res";
        std::ofstream out_file;
```

```
        int size = 0;
        for (int i = 0; i < all_ranks.size(); ++i) {
            size += all_ranks[i].size();
        }
        out_file.open(fname.c_str());
        out_file << size << std::endl;
        for (int i = 0; i < all_ranks.size(); ++i) {
            for (int j = 0; j < all_ranks[i].size(); ++j) {
                out_file << all_ranks[i][j] << " ";
            }
        }
    } else {
        mpi::gather(world, ranks, 0);
    }

    return 0;
}
```

6 Проверка корректности результатов

Для проверки корректности результатов был также написан код, использующий последовательную версию алгоритма. Компиляция также производилась с помощью makefile-а: make st (см. раздел 3.2). Запуск проводился с помощью bash файла, приведенного в Приложении (см. Листнг 3). Листинг последовательной версии приведен ниже:

```
#include <iostream>
#include <fstream>

#include <boost/graph/page_rank.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/algorithm/string.hpp>

using namespace boost;

typedef adjacency_list<vecS, vecS, directedS> Graph;

void read_graph(const std::string &file_name, Graph &g, unsigned int &vertices_count, long long &edges_count) {
    std::fstream file(file_name.c_str(), std::fstream::in | std::fstream::binary);

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

    // add edges from file
    for(long long i = 0; i < edges_count; i++) {
        unsigned int src_id = 0, dst_id = 0;
        float weight = 0;

        // read i-th edge data
        file.read((char*)(&src_id), sizeof(int));
        file.read((char*)(&dst_id), sizeof(int));

        //print edge data
        add_edge(vertex(src_id, g), vertex(dst_id, g), g);
    }

    file.close();
}
```

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cout << "Параметры: <входной файл> <масштаб графа>" << std::endl;
        return 1;
    }

    // получение имени входного файла с графом, числа вершин входного графа, а так же количество итераций
    ↪  pagerank
    std::string file_name = argv[1];
    unsigned int vertices_count = 1U << strtol(argv[2], NULL, 10);
    long long edges_count = 0;

    // Создаем граф с заданным числом вершин
    Graph g(vertices_count);

    // читаем граф на корневом процессе
    read_graph(file_name, g, vertices_count, edges_count);

    // считаем PageRank с замером времени выполнения
    std::vector<double> ranks(num_vertices(g));

    graph::page_rank(g, make_iterator_property_map(
            ranks.begin(), get(vertex_index, g)));

    std::vector<std::string> strs;
    split(strs, file_name, is_any_of("/"));
    std::string fname = "../res/" + strs[strs.size()-1] + ".st_res";
    std::ofstream out_file;

    out_file.open(fname.c_str());
    out_file << ranks.size() << std::endl;
    for (int i = 0; i < ranks.size(); ++i) {
        out_file << ranks[i] << " ";
    }

    return 0;
}
```

Также был написан код проверки результатов (см. Листинг 1) и bash файлы для запуска кода проверки (см. Листинг 2). Нужно добавить, что все версии прошли проверку.

7 Исследование масштабируемости

По результатам выполнения программы были построены трехмерные графики производительности для двух типов матриц (Рис. 1 и Рис. 2):

По оси X – число используемых программой MPI-процессов: 1, 2, 4, 8, 16, 32, 64, 128

По оси Y – степень числа вершин графа, от 14 до 18.

По оси Z – производительность программы в KTEPS (посчитана по формуле 1)

8 Выводы и результаты

Приведенные графики масштабируемости демонстрируют интересные свойства реализации алгоритма. Как видно, на обоих графиках алгоритм показывает достаточно хорошие результаты при одном процессе. Но при двух, резко падает производительность. Хотя

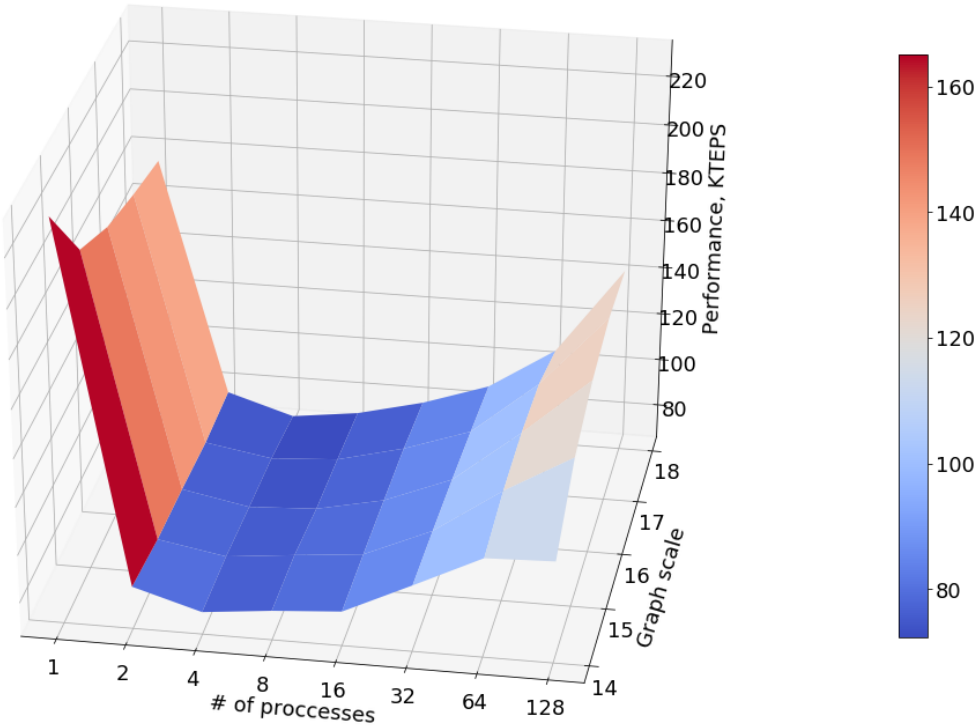

Рис. 1: График масштабируемости алгоритма boost::pagerank для графов типа RMAT

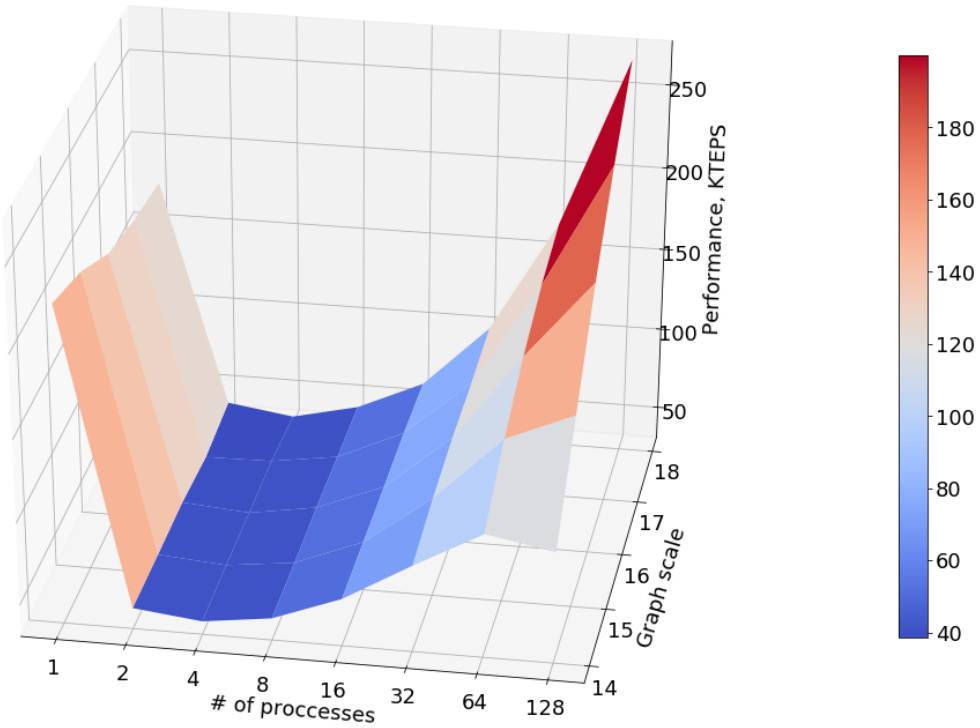

Рис. 2: График масштабируемости алгоритма boost::pagerank для графов типа SSCA2

начиная с четырех процессов производительность экспоненциально возрастает. Данное поведение можно объяснить следющим образом: т.к. данные помещаются в память одного процесса и нет обмена между процессами, то при одном процессе производительность достаточно хорошая. Начиная с двух процессов начинается обмен данными между процессами, что требует очень много времени. Т.к. при меньших количествах процессов, данных на один процесс приходится много, то увеличивается количество обмениваемых данных. Поэтому, при росте количества процессов, увеличивается производительность. Также можно заметить рост и по размерам графа при количестве процессов больших двух. Объяснить это можно тем, что каждый процесс, при малых количествах данных, не использовался на всю мощь. Это подтверждается и тем, что для одного процесса производительность уменьшается по мере увеличения размера графа.

Также при сравнении двух графиков можно видеть, что на SSCA2 графе производительность больше, чем на RMAT графе. Это связано со структурами графов. SSCA2 графы имеют независимые компоненты связности, что хорошо подходят для распараллеливания, т.к. обмен между процессами будет меньше, чем в случае с RMAT.

Для обоих графов по данным графикам нельзя определить глобальные максимумы (доступные аппаратные возможности не позволили это протестировать), т.к. видно, что график растет с ростом количества процессов и степенью графов. Таким образом можно сделать вывод, что алгоритм хорошо распараллеливается. Для данных графиков, максимум для RMAT достигается при 1 процессе и степени вершин 14 (это ни в коем случае нельзя это называть глобальным максимумом), и для SSCA2 – при 128 процессах и степени вершин графа 18.

9 Приложение

```cpp
#include <fstream>
#include <iostream>
#include <vector>
#include <cmath>


int main(int argc, char *argv[]) {
    int size1, size2;

    double EPS = 0.001;
    if (argc < 3) {
        std::cout << "Usage: check <input file 1> <input file 2>" << std::endl;
        return 1;
    }
    if (argc == 4)
        EPS = atof(argv[3]);

    std::ifstream file1, file2;
    file1.open(argv[1]);
    file2.open(argv[2]);

    file1 >> size1;
    file2 >> size2;
    if (size1 != size2) {
        std::cout << "Wrong sizes!\n";
        return 1;
    }

    double tmp1, tmp2;
    for (int i = 0; i < size1; ++i) {
        file1 >> tmp1;
        file2 >> tmp2;
        if (std::fabs(tmp1 - tmp2) > EPS) {
            std::cout << "Wrong values: " << i << " " << tmp1 << " " << tmp2 << std::endl;
            return 1;
        }
    }
    file1.close();
    file2.close();

    return 0;
}
```

Листинг 1: Код проверки результатов

```
#!/bin/bash
set -e

source params.sh

#make

TYPE=${TYPES[$1]}

echo ${TYPE}

for P in $(seq ${P_min} ${P_step} ${P_max})
do
    FILE_NAME=${DATA_PATH}graph_${TYPE}_${P}.bin
    for N in ${Ns[@]}
    do
        ./check ${FILE_NAME}_${P}.mp_res ${FILE_NAME}.st_res
    done
done

echo "Pass"
```

Листинг 2: Код запуска программы проверки

```
#!/bin/bash
set -e

source params.sh

TYPE=${TYPES[$1]}

echo ${TYPE}

for P in $(seq ${P_min} ${P_step} ${P_max})
do
    FILE_NAME=${DATA_PATH}graph_${TYPE}_${P}.bin
    echo "Running" ${P}
    LD_LIBRARY_PATH=${HOME}/libs/boost/lib:$LD_LIBRARY_PATH ./run_st ${FILE_NAME} ${P}
    echo "Done"
done
```

Листинг 3: Код запуска последовательной программы