МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ РАСКРАСКИ ГРАФОВ

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Ионова Кирилла Игоревича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н. _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования. Задача по раскраске графов является достаточно распространенной, у нее существует ряд применений.

Так, ее решением приходится заниматься при составлении расписаний. Имеется некоторое количество типов работ и временных промежутков. Требуется распределить работу во времени таком образом, чтобы в один момент времени выполнялись только те работы, у которых нет общих ресурсов.

Разработчики компиляторов учитывают алгоритмы по раскраске графа при распределении регистров. Как правило, компилируемая программа располагает достаточно большим числом переменных, тогда как у процессора ограниченное и малое количество регистров. Требуется хранить в регистрах процессора такие переменные, к которым происходит больше всего обращений.

Производители процессоров решают сходные задачи, стараясь добиться параллелизма на уровне команд на аппаратном уровне. Программа состоит из потока инструкций, призванного выполняться параллельно. Процессору предстоит определить, результаты каких инструкций не зависят друг от друга и могут выполняться параллельно.

Раскраска графов применяется при передаче скрытых сообщений, например, когда создателю сообщения требуется закодировать некоторую секретную информацию. Оригинальное сообщение представляется в виде графа, в него закладывается секретная информация, на выходе получается граф, защищенный цифровым знаком. О наличии скрытого послания в сообщении можно догадаться, сравнив хроматическое число у графов, построенных на данных исходного и полученного сообщения.

Все вышеперечисленное стало причиной появления работы по анализу алгоритмов по раскраске графа.

Цель исследования. Анализ алгоритмов по определению хроматического числа графа, сравнение быстродействия их параллельных вариантов в многоядерной системе.

Задачи работы:

- Ознакомление с теорией графов;
- Изучение псевдокода алгоритмов по определению хроматического числа;
- Анализ времени работы алгоритмов;

— Умение генерировать классы небольших графов (от 1 до 10 вершин);
— Кодирование алгоритмов с использованием возможностей Java.

Научная новизна. Параллельные алгоритмы, как правило, сравниваются на кластерах с большим количеством ядер. В данной работе исследуется быстродействие алгоритмов на домашних процессорах, возможности которых значительно выросли за последнее десятилетие. Также в данной работе разбираются проблемы параллелизма и предлагаются их решения.

Практическая значимость работы заключается в формировании картины касательно возможностей современных процессоров и их быстродействия на примере параллельных алгоритмов по правильной раскраске графов.

Основное содержание исследования. Выпускная квалификационная работа состоит из введения, одной теоретической и одной практической главы, заключения, списка использованных источников и приложений.

1 Исследование параллельных алгоритмов раскраски графа

Первая глава содержит теоретические данные по графам, в ней приводится псевдокод алгоритмов, анализируется их работа на классах небольших графов, рассматриваются проблемы параллелизации алгоритмов.

Графом представляет собой совокупность двух множеств: вершин и ребер. Вершины называются смежными, если они соединены ребрами. Раскраска графа — это процесс присвоения всем вершинам графа некоторого числа или символа, характеризующего цвет. [1]

Правильная раскраска предполагает, что ни одна из пар смежных вершин не будет окрашена в один цвет. Алгоритм по поиску хроматического числа ищет минимальное число цветов, при котором возможна правильная раскраска графа. [2]

Алгоритмы по раскраске графов начали изучаться в XIX веке. Исследователи преследовали цель узнать, возможно ли окрасить любую сферическую карту в четыре цвета. Доказательство проблемы четырех красок стало возможно благодаря вычислительным машинам. [3]

Были попытки определить хроматическое число графа за счет формул. Так известно, что хроматическое число графа не будет превышать степени максимальной вершины, к которой прибавили число 1.

$$\overline{d} = \frac{1}{n}\sum_{i=0}^{n} d_i; \quad n = 0, 1...N \tag{1}$$

$$\tilde{d} = d_{n/2} \tag{2}$$

Также хроматическое число можно посчитать как среднее арифметическое всех половин степеней вершин в графе (уравнение 2). Оценку можно улучшить, оставив в выборке только те вершины, чья степень не меньше 2. С этими условиями уравнение 1 дает ответ, который зачастую оказывается близок к правильному.

Наивный алгоритм раскраски графа не предполагает никаких эвристик.

```
1 параллельно для n вершин в множестве всех вершин V {
2       S = {множество цветов соседей текущей вершины}
3       покрасить текущую вершину в минимальный цвет, который не содержится в S
```

```
}
```

Итерируясь по всем вершинам графа, алгоритм окрашивает текущую вершину в минимальный цвет, который не был присвоен соседям 1. Это повторяется, пока есть непосещенные вершины. [4] В параллельной версии алгоритма каждый поток работает со своими вершинами.

Независимое множество — такое множество вершин на графе, что ни одна пара вершин в нем не смежна. Как правило, различные алгоритмы по поиску хроматического числа пытаются разбить граф на независимые множества и окрасить их параллельно.

Алгоритмы различаются лишь способом распределения вершин по независимым множествам и выбором окраски. Все рассматриваемые графы простые (не содержат петель и кратных ребер) и хранятся в централизованной системе, т.е. каждому потоку процессора доступна информация о всем графе.

2-distance алгоритм формирует на графе независимые множества вершин, причем в одно множество войдут только те вершины, которые находятся на расстоянии 2 друг от друга [5]. Каждое независимое множество будет окрашено в уникальный цвет.

```
функция dfs(v, D) { // v номер текущей вершины, D независимое множество вершин,
    изначально пустое
        пометить v как посещенную
        добавить v в D
        для всех w в множестве непосещенных вершин W {
                если расстояние между текущей вершиной w и v равно 2, то {
                        добавить текущую вершину w в D
                        запустить dfs (w, D)
                }
        }
        вернуть D
}
```

В 2-distance алгоритме потокам подается на вход стартовая и конечная вершины, из которых нужно построить независимое множество путем поиска в глубину — рекурсивного обхода графа, который предполагает продвижение из начальной вершины вглубь 1, покуда это еще является возможным. [6]

```
для всех i из множества вершин графа {
        для всех j из множества вершин графа {
                если вершины i и j смежны, то
```

```
                    d[i][j] = вес ребра между ними,
                    иначе d[i][j] = некоторое максимальное число
        }
}
для всех k из множества вершин графа {
        для всех i из множества вершин графа {
                для всех j из множества вершин графа {
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j])
                }
        }
}
```

Независимое множество строится по описанному способу, что вершины находятся на кратчайшем расстоянии 2 друг от друга. Поэтому до запуска основного алгоритма на графе необходимо посчитать матрицу кратчайших путей 1. Это можно сделать алгоритмом Флойда-Уоршелла. [7]

Кратчайшим путем между двумя вершинами является последовательность из минимального количества ребер. Матрица кратчайших путей содержит таковые пути для всех пар вершин.

```
L = {} // ассоциативный массив, где по целочисленным ключам содержатся независимые
    множества дистанции 2
параллельно для вершин v в множестве всех вершин V {
        color = 0
        если v не была посещена, то {
                D = {}
                L[color++] = dfs (v, D)
        }
}
для всех key, set из множества L {
        покрасить все вершины в set цветом key
}
```

В конечном итоге с учетом описанных функция алгоритм 2-distance выглядит так, как описано в листинге 1.

Алгоритм Джонса-Плассманна предполагает присвоение всем вершинам графа некоторого случайного уникального веса. [8] На текущем шаге обрабатываемая вершина исследует еще не покрашенных соседей и, если у данной вершины максимальный вес, окрашивается в минимально возможный цвет.

```
инициализировать w случайной перестановкой неповторяющихся весов
U = V
пока U содержит хотя бы 1 элемент {
        параллельно для вершин v в множестве всех вершин U {
                I = множество всех вершин в U, которые имеют наибольший вес среди
                    соседей
                параллельно для вершин s в множестве I {
                        S = {множество цветов соседей текущей вершины}
                        покрасить текущую вершину в минимальный цвет, который
                            несодержится в S
                }
        }
        вычесть из множества U множество I
}
```

Процесс продолжается, пока есть неокрашенные вершины 1. Потоки получают на вход номер начальной и конечной вершины, которые надо окрасить, и итерируются по заданному промежутку.

Largest-Degree-First алгоритм предполагает формирование независимого множества на основании весов и степеней вершин. Каждой вершине устанавливается в соответствие некий случайный вес. [9] Также определяются степени.

```
функция LDF(w[]) {
        D = множество вершин, отсортированное в порядке убывания степеней вершин
        пока U содержит хотя бы 1 элемент {
                параллельно для вершин v в множестве D {
                        I = {}
                        добавить вершину v в множество I, если
                                у нее наибольшая степень среди соседей,
                                либо у нее максимальная степень и наибольший вес среди
                                    соседей с такой же степенью
                        параллельно для вершин w в множестве I {
                                S = {множество цветов соседей текущей вершины}
                                покрасить текущую вершину в минимальный цвет,
                                    который не содержится в S
                        }
                }
                вычесть из множества D множество I
        }
}
инициализировать w[] случайной перестановкой неповторяющихся весов
```

```
LDF(w)
```

Потокам на вход подаются интервалы номеров вершин, которые надо обработать. На текущем шаге обрабатываемая вершина красится в цвет, если у нее максимальная степень среди соседей, либо же есть соседи с такой же степенью, но у текущей вершины наибольший вес 1.

Smallest-Degree-Last алгоритм более комплексно подходит к распределению весов между вершинами. Вершины упорядочиваются по возрастанию степеней, текущий вес устанавливается минимальным. На очередном шаге всем вершинам минимальной степени назначается текущий вес, после чего рассмотренные вершины удаляются из графа, степени оставшихся пересчитываются, а вес увеличивается. [10]

```
функция SDL {
        инициализировать w[] нулями
        k = 1
        i = 1
        U = V
        пока U содержит хотя бы 1 элемент {
                пока существуют вершины v в U со степенью не менее k выполнять
                    параллельно {
                        S = {множество вершин v со степенью не менее k}
                        для всех вершин v в S {
                                w(v) = i  // назначить i весом вершины v
                        }
                        вычесть из множества U множество S
                        i++
                }
                k++;
        }
        вернуть w
}
LDF(SDL())
```

Процесс продолжается, пока есть непосещенные вершины. Далее вершины сортируются по убыванию своих изначальных степеней, также для каждой из них посчитан вес. После этого можно запускать LDF или алгоритм Джонса-Плассманна, порядок окраски будет определяться весом вершин, посчитанным на прошлом шаге 1.

```
errorVertices  = {}  // список ошибочных вершин
```

```
параллельно для n вершин в графе {
        для всех соседей текущей вершины {
                если цвет соседа совпадает с цветом текущей вершины, добавить вершину
                минимального номера из данной пары в список ошибочных вершин
        }
}
для всех вершин в errorVertices {
        S = {множество цветов соседей текущей вершины}
        покрасить текущую вершину в минимальный цвет, который не содержится в S
}
```

У рассмотренных параллельных алгоритмов могут возникнуть ошибки ввиду того, что две соседние вершины могут быть окрашены двумя потоками в один момент времени. Поэтому после работы алгоритмов необходимо еще раз пройти по графу, определить все смежные вершины, окрашенные в один цвет, и добавить их в список на перекраску 1.

Также при разработке алгоритмов были учтены проблемы синхронизации, все использованные структуры данных поддерживают неблокирующее чтение и сегментированную запись.

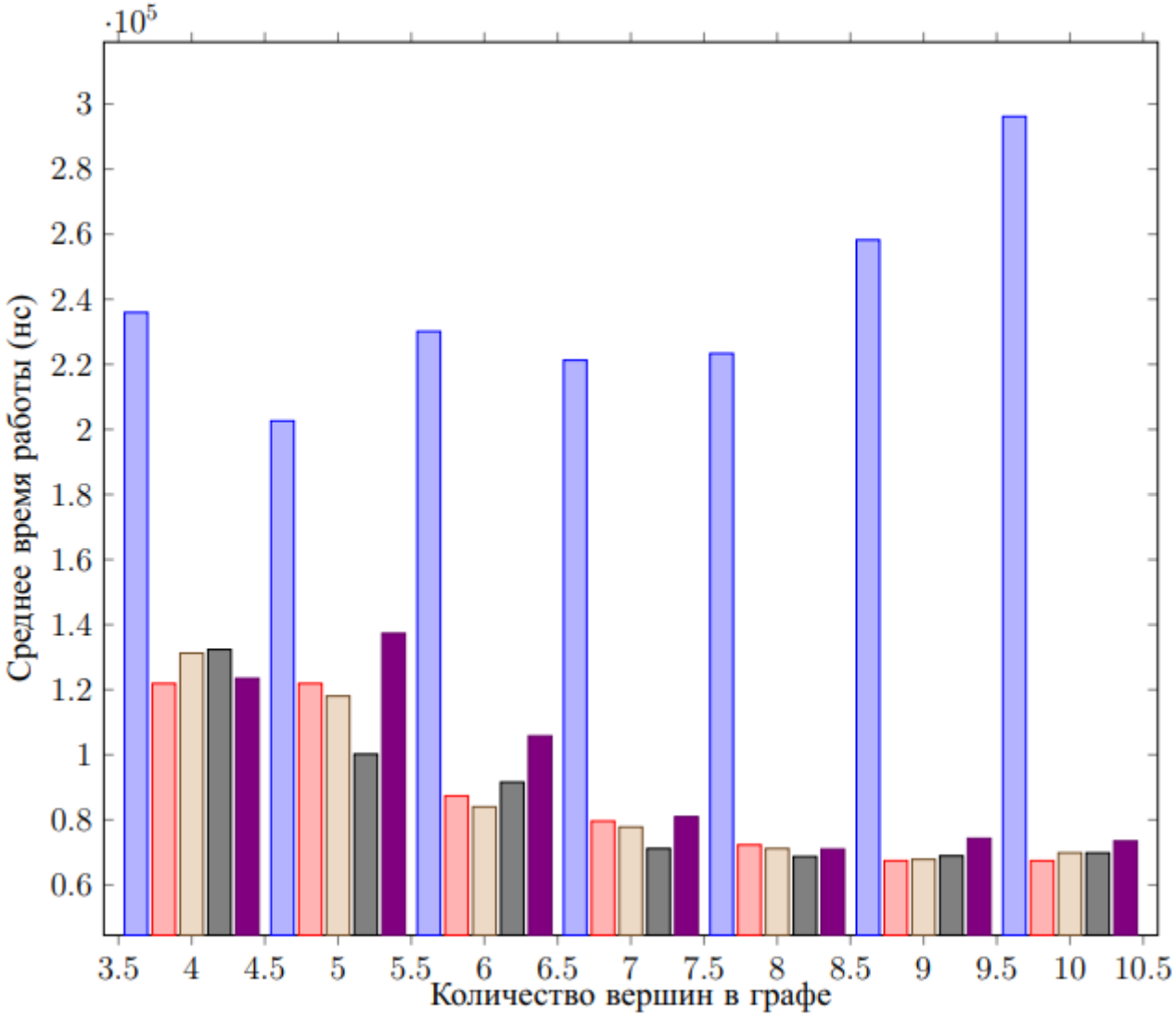

Рисунок 1 – Наивный алгоритм раскраски вершин

Было измерено среднее время работы рассмотренных 5 алгоритмов на классах $4 - 10$ вершинных графов (рисунок 1). Наивный алгоритм оказался значительно медленнее остальных участников теста, SDL также оказался неоптимальным. Алгоритмы 2-distance, Джонса-Плассманна и LDF работают оптимально, поэтому при выборе параллельного алгоритма можно пользоваться любым из них.

Также сравнению подверглись все алгоритмы на четырех различных процессорах Intel. Алгоритмы показали свою масштабируемость в зависимости от числа имеющихся потоков.

2 Программная реализация параллельных алгоритмов по раскраске графа

Вторая глава описывает процесс генерации графов для программы. В ней рассматривается потокобезопасная реализация алгоритмов на Java.

Список смежности — это способ представления графа, в котором для каждой вершины перечисляются номера смежных с ней вершин.

Для генерации графов используется программа $geng$ из пакета $nauty\ and\ traces$ [11], которой на вход подается число вершин в классе графа. Все сущности представлены в сжатом формате [12], который переводится в список смежности при помощи программы $showg$ из того же пакета.

Реализованная программа представлена в формате jar-архива. В архив вшиты настройки по умолчанию, однако поддерживаются внешние параметры конфигурации, которые должны быть описаны в файле $application.properties$. Также можно переопределить детали логгирования в $logj2.xml$ [13].

Программа сканирует входную директорию на предмет файлов с графами, при этом рассматриваются только графы с числом вершин, попадающим в заданный интервал в конфигурационном файле.

Графы обрабатываются на диапазоне алгоритмов, начальный и финальный номер которых заданы в конфигурационном файле. Установлено следующее соответствие:

- 1 — наивный параллельный алгоритм;
- 2 — 2-distance алгоритм;
- 3 — алгоритм Джонса-Плассманна;
- 4 — LDF-алгоритм;
- 5 — SDL-алгоритм.

Результатом работы программы являются выходные файлы в директориях $details$, $short$ и $summary$, находящихся в папке $results$. В $details$ содержится максимально подробная информация, в $short$ время работы алгоритмов и только оно, в $summary$ метрики по среднему и общему времени работы.

Выходные файлы в своем имени содержат номер алгоритма и число вершин графа. Если конфигурационный файл содержит какие-то ошибки или не были найдены заданные описания графов, будет выброшено исключение.

На пользователя возлагается ответственность по генерированию графов с помощью вышеуказанных инструментов, программа не проверяет входные

файлы на корректность. Если какой-то из графов содержит ошибку в описании, программа завершит свою работу, и данные о всех прошлых успешно обработанных графах будут записаны в выходные директории.

В конкретный момент времени программа хранит в памяти и обрабатывает только один граф. На нем прогоняются все алгоритмы заданное число раз. Время работы алгоритма определяется как среднее арифметическое от общего времени работы всех запусков.

Входной точку в программу служит класс $JavaLauncher$, в нем создается экземпляр класса $ApplicationProperties$, где парсятся все параметры конфигурации. Приоритет отдается внешнему конфигурационному файлу, если его нет, считываются опции из внутреннего.

Далее управление главным потоком передается классу $AlgorithmController$. Тут считываются входные данные из заданного диапазона и прогоняются на конкретных алгоритмах, после чего записываются в выходные файлы, чье имя является комбинацией первых двух параметров.

```
Graph 12005168, chromatic number = 10, computation time: 795900 nanoseconds
0: RED
1: GREEN
2: BLUE
3: WHITE
4: BLACK
5: YELLOW
6: BROWN
7: ORANGE
8: PURPLE
9: PINK
```

Рисунок 2 – Формат выходного файла

Графы считываются в формате списка смежности. В выходном файле для каждой вершины будет записан ее цвет, при этом раскраска будет являться правильной. Также для каждого графа записываются его хроматическое число и время работы алгоритма на данном графе (рисунок 2).

После создания всех сущностей и выбора запускаемого алгоритма $Algorithm\ Controller$ передает управление абстрактному классу $AbstractAlgorithm$. Здесь вычисляется число используемых в вычислении потоков (для параллельного алгоритма максимальное число потоков процессора, для последовательного 1).

Также вычисляется «чанк» – количество вершин, которое будет передано на обработку каждому потоку. Каждый алгоритм запускается на графе

заданное число раз, после чего записывается среднее арифметическое времени работы алгоритма.

$AbstractAlgorithm$ содержит метод для коррекции ошибок, вызываемый из наследников класса. Методу передается на вход множество вершин, которые нужно перекрасить. Цвет выбирается минимальный доступный — с учетом раскрасок соседей.

От $AbstractAlgorithm$ наследуются пять классов, в каждом из которых реализован свой алгоритм раскраски. Они переопределяют или используют уже готовую логику классов $Colorer$ и $Recolorer$, в которых описаны сущности, необходимые для раскраски и перекраски вершин.

$Colorer$ и $Recolorer$ — это задачи, которые будут переданы фиксированному пулу исполнителей, определенному в каждом алгоритме. Это высокоуровневое средство организации потоков, которому достаточно передать на вход задачи и ожидать результата.

Фиксированный пул исполнителей самостоятельно контролирует цикл работы потоков, в основе лежит идея переиспользования, т.е. вместо инициализации новых потоков свежие задачи передаются старым потокам, завершившим свою работу.

В реализации программы фиксированному пулу передаются задачи по окраске и перекраске графа. Далее, в зависимости от реализации алгоритма, исполнители либо самостоятельно записывает значение в граф и ничего не возвращает (тип $Runnable$), либо от него ожидается какое-то значение в будущем ($Future$-данные [14] для типа $Callable$).

В первом алгоритме на вход каждому исполнителю подается диапазон из начальной и конечной вершин (длиной в «чанк»). Это позволяет гарантировать, что каждый поток получит равное число вершин на обработку, кроме может быть последнего.

Каждый исполнитель красит текущую вершину в минимальный цвет, по окончанию обработки всего графа производится многопоточный анализ на ошибочно покрашенные вершины.

В 2-distance алгоритме на графе считается матрица кратчайших расстояний, после чего формируются независимые множества вершин расстояния 2 и окрашиваются в уникальные цвета.

В алгоритме Джонса-Плассманна вершинам изначально присваиваются

случайные веса (перестановка их индексов). Далее начинается многопоточная обработка графа.

Текущая вершина красится в минимальный доступный цвет, если среди неокрашенных смежных вершин у нее самый большой номер. Обработка продолжается, пока есть неокрашенные вершины.

LDF-алгоритм в текущий момент окрашивает вершину в наименьший доступный цвет, если: у нее либо наибольшая степень среди всех смежных неокрашенных вершин, либо наибольший случайный вес среди всех вершин той же степени.

SDL-алгоритм распределяет веса по вершинам таким образом, что вершины наименьшей степени получат минимальный вес. Далее запускается основной алгоритм, шаги которого совпадают с описанными в LDF.

ЗАКЛЮЧЕНИЕ

Рассмотрен ряд параллельных алгоритмов, занимающихся поиском минимальной вершинной раскраски произвольных графов. В результате этого:

- Проведено знакомство с теорией графов;
- Изучен псевдокод алгоритмов по определению хроматического числа;
- Проанализировано время работы алгоритмов;
- Сгенерированы полные классы графов с небольшим числом вершин;
- Рассмотренные алгоритмы реализованы на высокоуровневом языке программирования Java.

Был рассмотрен тот факт, что скорость работы алгоритмов зависит от числа потоков процессора, которые доступны программе. Поддерживаются вычисления на графах с произвольным числом вершин. Реализованный код пригоден для использования в централизованных системах, возможна интеграция в него новых видов алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Harary, F.* Graph Theory / F. Harary. — Boston: Addison-Wesley, 1969.

2 *Molloy, M. S.* Graph colouring and the probabilistic method / M. S. Molloy, B. Reed. — Berlin: Springer, 2002.

3 *Appel, K.* Every planar map is four colorable. part i: Discharging / K. Appel, W. Haken // *Illinois J. Math.* — 1977. — Vol. 21, no. 3. — Pp. 429–490.

4 *Normann, P.* Parallel graph coloring: Parallel graph coloring on multi-core cpus / P. Normann // *Uppsala University*. — 2014. — Vol. 1. — Pp. 1–45.

5 *Bozdag, D.* A parallel distance-2 graph colouring algorithm for distributed memory computers / D. Bozdag, U. Catalyurek, A. H. Gebremedhin, F. Manne, E. G. Boman, F. Ozguner // *High Performance Computing and Communications*. — 2005. — Vol. 1. — Pp. 796–806.

6 *Even, S.* Graph algorithms / S. Even. — 2011. — Vol. 1. — Pp. 46–48.

7 *Floyd, R. W.* Algorithm 97: Shortest path / R. W. Floyd // *Communications of the ACM*. — 1962. — Vol. 5, no. 6. — P. 345.

8 *Jones, M. T.* A parallel graph coloring heuristic / M. T. Jones, P. E. Plassmann // *SIAM Journal of Scientic Computing 14*. — 1993. — Vol. 1. — P. 654.

9 *Welsh, D. J. A.* An upper bound for the chromatic number of agraph and its application to timetabling problems / D. J. A. Welsh, M. B. Powell // *The Computer Journal*. — 1967. — Vol. 10. — Pp. 85–86.

10 *Matula, D. W.* Graph coloring algorithms / D. W. Matula, G. Marble, J. D. Isaacson // *Computers and Operations Research*. — 1972. — Vol. 13. — Pp. 27–32.

11 *McKay, B. D.* Practical graph isomorphism, {II} / B. D. McKay, A. Piperno // *Journal of Symbolic Computation*. — 2014. — Vol. 60, no. 0. — Pp. 94 – 112.

12 Description of graph6, sparse6 and digraph6 encodings [Электронный ресурс]. — URL: http://users.cecs.anu.edu.au/~bdm/data/formats.txt (Дата обращения 29.05.2020). Загл. с экр. Яз. англ.

13 Log4j – Configuring Log4j 2 - Apache Logging Services [Электронный ресурс]. — URL: https://logging.apache.org/log4j/2.x/manual/configuration.html (Дата обращения 29.05.2020). Загл. с экр. Яз. англ.

14 Future (Java Platform SE 7) - Oracle Docs [Электронный ресурс]. — URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html (Дата обращения 29.05.2020). Загл. с экр. Яз. англ.