Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»

**Выпускная квалификационная работа бакалавра**

**«Параллельные алгоритмы раскраски графов»**

**Выполнила**:

студентка группы 381803-2

Курникова А.А.

**Научный руководитель**:   
к. т. н., доцент кафедры МОСТ

Мееров И.Б.

Нижний Новгород

2022

**Оглавление**

[Введение 4](#_Toc105451805)

[1. Основные понятия и определения 6](#_Toc105451806)

[2. Постановка задачи и цель работы 8](#_Toc105451807)

[2.1. Математическая постановка задачи 8](#_Toc105451808)

[2.2. Цель и задачи работы 8](#_Toc105451809)

[3. Последовательный жадный алгоритм раскраски графа 10](#_Toc105451810)

[3.1. Общие сведения 10](#_Toc105451811)

[3.2. Схема последовательного жадного алгоритма 10](#_Toc105451812)

[4. Параллельные алгоритмы 13](#_Toc105451813)

[4.1. Общие сведения 13](#_Toc105451814)

[4.2. Схема стандартного алгоритма Джонса-Плассмана 14](#_Toc105451815)

[4.3. Модифицированный алгоритм Джонса-Плассмана 15](#_Toc105451816)

[4.4. Алгоритм Чаталюрека 15](#_Toc105451817)

[5. Переносимые параллельные реализации 17](#_Toc105451818)

[5.1. Общие сведения 17](#_Toc105451819)

[5.2. Переносимая параллельная реализация с использованием модели KOKKOS 19](#_Toc105451820)

[5.3. Переносимая параллельная реализация с использованием модели Data Parallel C++ 20](#_Toc105451821)

[5.4. Реализация алгоритма Чаталюрека для GPU 20](#_Toc105451822)

[6. Программная реализация 22](#_Toc105451823)

[6.1. Файловая структура программы 22](#_Toc105451824)

[6.2. Структуры данных 22](#_Toc105451825)

[6.3. Модульная структура программы 23](#_Toc105451826)

[7. Результаты экспериментов 29](#_Toc105451827)

[7.1. Методика проведения экспериментов 29](#_Toc105451828)

[7.2. Последовательный жадный алгоритм 31](#_Toc105451829)

[7.3. Параллельные алгоритмы 32](#_Toc105451830)

[7.4. Реализация с использованием KOKKOS и Data Parallel C++ 35](#_Toc105451831)

[7.5. Запуск реализаций алгоритма Чаталюрека на GPU 38](#_Toc105451832)

[Заключение 42](#_Toc105451833)

[Список литературы 44](#_Toc105451834)

[Приложение А. Фрагменты кода программы 46](#_Toc105451835)

# ****Введение****

На протяжении последних десятилетий активно развивается теория графов. Стремительно расширяется область приложений теории графов, увеличивается количество задач, которые могут быть решены посредством методов теории графов. Одной из вычислительно-трудоемких задач на графах является задача отыскания хроматического числа графа – минимального числа цветов, необходимых для раскраски вершин графа. Данная задача имеет большое значение для практического использования. Так, раскраска вершин графа (graph coloring) позволяет моделировать многие проблемы планирования, такие как распределение радиочастот [1], хранение химических веществ [2], составление расписаний [2], распределение регистров в микропроцессорах [3], политическая картография [3]. В многочисленных практических приложениях раскраски графов вершины в каждом цветовом классе обычно представляют отдельные субъекты, которые не конкурируют или не конфликтуют друг с другом [4]. Раскраска графов находит применение и в различных алгоритмических задачах, например, в задаче решения популярной головоломки Судоку.

Раскраска графов как алгоритмическая проблема начала изучаться с 1970-х годов: определение хроматического числа входит в число 21 NP-полных задач Карпа [4]. Примерно в то же время были разработаны разнообразные алгоритмы на базе поиска с возвратом и рекурсивного удаления и стягивания Зыкова [5]. Помимо классических типов проблем, рассматриваются задачи с наложением ограничений на сам граф, способ присвоения цветов или на сами цвета.

Задача нахождения раскраски в минимальное число цветов на практике решается эвристическими методами. Такие методы не всегда находят наилучшее возможное решение, но позволяют получить достаточный результат за приемлемое время. Среди них отметим жадные алгоритмы, алгоритмы стягивания, полиномиальные алгоритмы, точные алгоритмы и др. В высокопроизводительных приложениях используются параллельные алгоритмы раскраски графов для вычислительных систем с общей и распределенной памятью. Среди них отметим алгоритмы Боумана [6], Чаталюрека [7], Джонса-Плассмана [8] и другие.

Несмотря на наличие нескольких широко распространенных алгоритмов раскраски графа [9], поиск возможностей для ускорения вычислений при использовании современных вычислительных систем постоянно продолжается. В этом направлении и выполнена данная работа.

В работе рассматриваются четыре алгоритма раскраски графов: жадный алгоритм, алгоритм Джонса-Плассмана, модифицированный алгоритм Джонса-Плассмана и алгоритм Чаталюрека (Catalyurek). Для жадного алгоритма выполнено сравнение использования различных функций приоритета вершин. Параллельные алгоритмы Джонса-Плассмана и Чаталюрека реализованы с применением трех технологий распараллеливания (OpenMP, KOKKOS, DPC++), произведен анализ запусков на различных архитектурах и типах процессоров. Основная идея работы заключается в том, чтобы установить, какие подходы к распараллеливанию алгоритмов раскраски графа и какие средства распараллеливания наилучшим образом подходят для современных центральных и графических процессоров Intel.

# Основные понятия и определения

Раскраска графа – теоретико-графовая конструкция, частный случай разметки графа. Раскраска ставит элементам графа в соответствие метки с учётом определённых ограничений; эти метки принято называть цветами. Когда говорят о раскраске графов, почти всегда подразумевают под этим *раскраску вершин* – способ раскраски вершин графа, при котором любым двум смежным вершинам соответствуют разные цвета [4]. Терминология, в которой метки называются цветами, происходит от раскраски политических карт. Метки с названиями конкретных цветов используются, только когда число цветов мало, чаще же подразумевается, что метки являются целыми числами.

*Раскраска рёбер* присваивает цвет каждому ребру так, чтобы любые два смежных ребра имели разные цвета [10].

*Раскраска областей планарного графа* назначает цвет каждой области так, что каждые две области, имеющие общую границу, не могут иметь одинаковый цвет.

Раскраска с использованием цветов называется *-раскраской*.

Наименьшее число цветов, необходимое для раскраски графа, называется его *хроматическим числом* [4].

Подмножество вершин, выделенных одним цветом, называется *цветовым классом*, каждый такой класс формирует независимый набор. Таким образом, -раскраска – это то же самое, что и разделение вершин на  независимых наборов [4].

*Качество раскраски* будем определять по количеству цветов: чем меньше цветов было использовано для полной раскраски графа, тем лучше качество.

*Функция приоритета* – это функция, определяющая очередность обхода вершин графа для алгоритма.

*Степень насыщенности* – количество уже раскрашенных соседей вершины.

*Степень вершины* – число ее соседей.

*Жадный алгоритм* – алгоритм, заключающийся в принятии локально-оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

*Жадная раскраска* – раскраска вершин неориентированного графа, созданная жадным алгоритмом, который проходит вершины графа в некоторой предопределённой последовательности и назначает каждой вершине первый доступный цвет.

*Эвристический алгоритм* – это алгоритм решения задачи, включающий практический метод и не являющийся гарантированно точным или оптимальным, но достаточный для решения поставленной задачи.

*Кликой неориентированного графа* называется подмножество его вершин, любые две из которых соединены ребром [5].

*Независимый набор* – это набор таких вершин, что никакие две из них не являются соседями [10].

*Функтор* – это функциональный объект программы.

*Лямбда* – это сокращение функтора, автокомпилятор.

*Кэширование* – пересылка данных в быструю память для эффективного повторного использования [11].

*Центральный процессор (CPU)* – это электронный блок или интегральная схема, исполняющая машинные инструкции. Это главная часть аппаратного обеспечения компьютера или программируемого логического контроллера.

*Графический процессор (GPU)* – отдельное устройство персонального компьютера или сервера, выполняющее графический рендеринг, которое позднее стало массово применяться и в других устройствах в качестве универсальных вычислительных устройств с параллельной архитектурой [12].

# Постановка задачи и цель работы

## Математическая постановка задачи

Поставим задачу в математических терминах.

Пусть дан граф , – множество вершин данного графа, – множество его ребер. Требуется найти разбиение множества вершин на непересекающихся подмножеств , таким образом, чтобы внутри каждого подмножества не содержалось смежных вершин. Если теперь каждому из таких подмножеств поставить в соответствие определенный цвет, то вершины внутри этого подмножества можно раскрасить в один цвет, вершины другого подмножества – в другой цвет, и так далее до раскраски всех подмножеств.

## Цель и задачи работы

Целью работы является сравнительный анализ подходов к распараллеливанию вычислений при решении задачи о раскраске графа с использованием современных центральных и графических процессоров Intel.

Достижение данной цели требует решения следующих задач:

1. Изучить параллельные алгоритмы раскраски графа, выбрать наиболее перспективные для реализации, тестирования и анализа;
2. Изучить современные средства для параллельного программирования oneAPI и KOKKOS, позволяющие разработать переносимую программную реализацию алгоритмов раскраски, которая далее может использоваться и на центральных, и на графических процессорах;
3. Выполнить параллельную программную реализацию выбранных алгоритмов с использованием языков и библиотек C++/OpenMP, Data Parallel C++, KOKKOS;
4. Выбрать тестовые данные и провести сравнение производительности разработанных кодов программ на доступных CPU и GPU. Выявить основные архитектурные механизмы, которые влияют на производительность. Улучшить код в соответствии с результатами анализа;
5. Сформулировать основные выводы по результатам работы.

# Последовательный жадный алгоритм раскраски графа

## Общие сведения

Жадные алгоритмы для решения задач раскраски графа, в общем случае, не дают минимально возможное число цветов, однако они используются в математике в качестве техники доказательств других результатов, относящихся к раскраске, а также в компьютерных программах для получения раскраски с небольшим числом цветов [4]. Существуют графы, для которых с большой вероятностью случайно выбранная последовательность вершин приведёт к использованию числа цветов, существенно большему минимально необходимого. Таким образом, очень важно осторожно выбирать последовательность, в которой вершины проходятся жадным алгоритмом [9].

Вершины любого графа всегда можно упорядочить таким образом, что качественная раскраска может быть получена с использованием жадного алгоритма. Так, для качественной раскраски перенумеровываются в убывающем порядке вершины из наименьшего по размеру множества вершин одного цвета. Затем перенумеровывается второе по размеру множество, третье, и так далее. Если далее применять жадный алгоритм с полученным порядком вершин, итоговая раскраска будет достаточно качественной.

Определение того, существует ли в данном графе совершенное упорядочение вершин, является NP-полной задачей. По этой причине для раскраски графов с целью уменьшения числа цветов используются эвристические алгоритмы, хоть они и не дают оптимального числа цветов.

## Схема последовательного жадного алгоритма

В данной работе рассматривается реализация, в которой выбор последовательности окрашивания вершин осуществляется при помощи некоторой заранее известной функции, подставляющейся затем в функцию раскраски. Так, реализованы упорядочивание вершин по убыванию степени насыщенности (Saturation Degree Ordering, SDO), убыванию степени вершины (Largest Degree First, LDF) и в прямом порядке появления вершин в графе (First Fit, FF).

Последовательный жадный алгоритм заключается в следующем:

1. Пометить все вершины как “бесцветные”;
2. Последовательно для каждой вершины:
   1. выбрать вершину согласно функции приоритета;
   2. выбрать наименьший свободный цвет вершины;
   3. для всех вершин, последующих за данной по приоритету, повысить минимальный доступный цвет на единицу;
   4. убрать данную вершину из множества нераскрашенных.

Далее приведены примеры использования алгоритма с подстановкой различных функций приоритета (Рисунок 1, Рисунок 2, Рисунок 3):

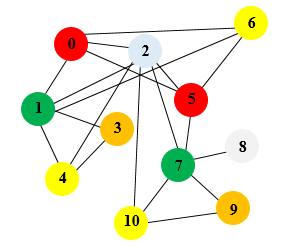


Рисунок . Пример последовательной раскраски LDF

Порядок раскраски вершин: 2, 1, 7, 0, 5, 4, 6, 10, 3, 9, 8

Порядок цветов: 1 – голубой, 2 – зеленый, 3 – красный, 4 – желтый, 5 – оранжевый, 6 – серый

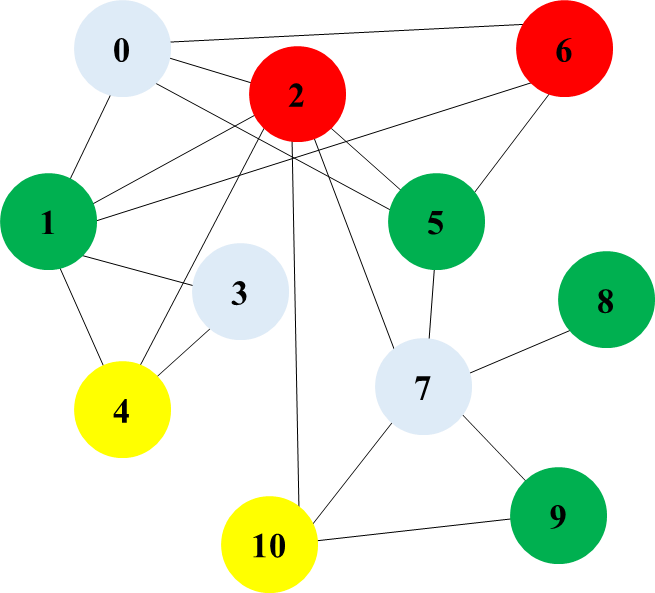


Рисунок . Пример последовательной раскраски FF

Порядок раскраски вершин: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Порядок цветов: 1 – голубой, 2 – зеленый, 3 – красный, 4 – желтый

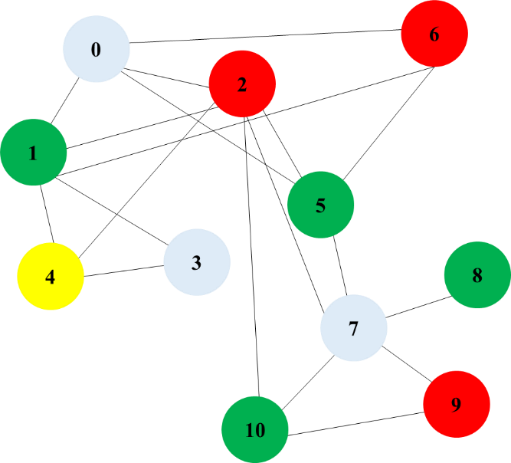


Рисунок . Пример последовательной раскраски SDO

Порядок раскраски вершин: 0, 1, 2, 5, 6, 4, 3, 7, 10, 9, 8

Порядок цветов: 1 – голубой, 2 – зеленый, 3 – красный, 4 – желтый

# Параллельные алгоритмы

## Общие сведения

При разработке параллельных алгоритмов также часто встречается задача раскраски. Рассмотрим практический пример такой задачи. Предположим, что вершины графа — это вычислительные устройства, которые могут коммутировать между собой, если они соединены каналом – ребром. Интересующая нас задача состоит в том, чтобы каждое вычислительное устройство выбрало для себя цвет так, чтобы соседние устройства выбрали отличные цвета от данного.

Для графов с достаточно большой максимальной степенью вершин наиболее используемые из параллельных алгоритмов раскраски работают быстрее, чем детерминированные алгоритмы. Наиболее быстрые параллельные алгоритмы используют технику множественных попыток.

В ненаправленных графах параллельные алгоритмы не способны отыскать достаточно качественную раскраску вершин – нужна дополнительная информация. Обычно делается предположение, что первоначально каждая вершина имеет уникальный идентификатор, например, из множества натуральных чисел, иными словами, предполагается, что нам сразу дана некоторая раскраска. Задача состоит в том, чтобы уменьшить количество цветов. Чем больше цветов используются, тем меньше коммуникаций потребуется для решения.

Описанный выше жадный алгоритм является наиболее предпочтительным из последовательных алгоритмов раскраски графа, но при его распараллеливании возникает ряд проблем. Так, простая версия распределённого жадного алгоритма требует итераций синхронизации в худшем случае, то есть, вероятно, придется проходить через все вершины. Поэтому для реализации были выбраны другие алгоритмы.

Существует две стратегии раскраски вершин, зависящих от требований к получаемой раскраске. Если требуется раскрасить граф в как можно меньшее число цветов, то в алгоритме обычно выбирается наименьший доступный цвет, то есть наименьший цвет, который еще не используется соседней вершиной. Если же целью является сбалансированная раскраска вершин, то выбирается наименее используемый доступный цвет, чтобы сбалансировать количество вершин каждого цвета.

## Схема стандартного алгоритма Джонса-Плассмана

В работе рассматривается алгоритм Джонса-Плассмана [8]. Его идея заключается в том, чтобы раскрасить некоторое множество независимых вершин параллельно, а далее в порядке приоритета раскрасить всех соседей уже цветных вершин. Вершины обрабатываются в порядке приоритета. Функция приоритета подается параметром.

В исходной статье [8] алгоритм раскраски графа состоит из двух фаз: начальной параллельной фазы, использующей функцию приоритета, и локальной фазы, которая использует один из возможных способов последовательного окрашивания.

Пошаговая интерпретация алгоритма заключается в следующем:

1. Для каждой вершины графа назначаем приоритет согласно заданной функции;
2. Помечаем все вершины как бесцветные;
3. Пока множество бесцветных вершин непустое:
   1. выбираем независимое множество вершин, которые имеют максимальный вес;
   2. параллельно назначаем найденным вершинам минимальный доступный цвет;
   3. удаляем обработанные вершины из множества бесцветных.

## Модифицированный алгоритм Джонса-Плассмана

Согласно работе [8], функция приоритета по максимальной степени вершины (Largest Degree First) была подставлена в алгоритм Джонса-Плассмана. В этом методе вершины раскрашиваются не в случайном порядке, а в порядке убывания степени, причем вершины наибольшей степени раскрашиваются в первую очередь. Этот подход ориентирован на использование меньшего количества цветов, чем стандартный алгоритм Джонса-Плассмана.

## Алгоритм Чаталюрека

Существует еще один подход: раскраска выполняется итерационно (спекулятивная раскраска). На каждой итерации процессы параллельно определяют цвета для своих локальных нераскрашенных вершин, затем обмениваются результатами и исправляют ошибки раскраски граничных вершин. К алгоритмам такого типа относятся алгоритмы Гебремедхин и Манна, Боумана [6], а также алгоритм Чаталюрека [7], рассмотренный далее.

Алгоритм Чаталюрека подходит для любой системы с общей памятью, в том числе для современных многоядерных платформ.

Алгоритм Чаталюрека состоит из следующих шагов:

1. Вершины делятся на равных блоков. – множество вершин -го потока;
2. Пока остались нераскрашенные вершины:
3. параллельно на каждом потоке назначаем каждой вершине минимальный доступный цвет;
4. находим множество конфликтных вершин (вершин, для которых не выполняется правило различно раскрашенных соседей), которое необходимо перекрасить. При этом для каждого неправильно раскрашенного ребра добавляем одну инцидентную ему вершину во множество перекрасок;
5. синхронизируем потоки;
6. множество нераскрашенных вершин состоит из множества перекрасок.

# Переносимые параллельные реализации

## Общие сведения

Ранее были описаны параллельные алгоритмы, реализованные на OpenMP – открытом стандарте для распараллеливания программ на языках С, С++ и Fortran. Он даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью. На данный момент известно несколько других открытых языков и библиотек для параллельного программирования, авторы которых заявляют о возможности написания одного кода, который может далее работать на совершенно разных вычислительных устройствах, в том числе, на центральных и графических процессорах. Возможность разработки универсального (переносимого) кода является бесспорным достоинством таких средств написания параллельных программ. Перспективными разработками являются недавно представленный язык программирования Data Parallel C++ и библиотека KOKKOS, именно они и изучаются в данной работе. Основной интерес представляет вопрос о том, будет ли код, изначально оптимизированный для CPU, эффективно работать на GPU, и какие изменения для этого потребуются. Рассмотрим кратко основные особенности применяемого инструментария.

KOKKOS C++ Performance Portability EcoSystem [14] — это решение производственного уровня для написания современных приложений на C++ независимо от аппаратного обеспечения. Он является частью ведущей инициативы по подготовке сообщества высокопроизводительных вычислений к следующему поколению суперкомпьютерных платформ. Экосистема состоит из нескольких библиотек, решающих основные проблемы переносимой разработки и обслуживания приложений. Тремя основными компонентами являются модель программирования, математические библиотеки и инструменты профилирования и отладки. KOKKOS реализует модель программирования на C++ для написания высокопроизводительных переносимых приложений, ориентированных на все основные платформы. Он предоставляет абстракции как для параллельного выполнения кода, так и для управления данными. Модель программирования KOKKOS предназначена для параллельных алгоритмов, использующих многоядерные чипы и разделяющих память между ядрами. Она включает в себя абстракции для часто используемых шаблонов параллельных вычислений, политики, предоставляющие подробные сведения о том, как применяются эти шаблоны вычислений, и пространства выполнения, указывающие, на каких ядрах выполняются параллельные вычисления. Модель программирования KOKKOS требует, чтобы разработчики приложений реализовывали свои алгоритмы с точки зрения шаблонов, политик и пространств KOKKOS, а затем сопоставляет эти алгоритмы с целевой архитектурой в соответствии со специфическими для данной архитектуры правилами, необходимыми для достижения наилучшей производительности.

Data Parallel C++ – это модель параллельного программирования, основанная на новейших стандартах языка C++ [15]. Data Parallel C++ базируется на Khronos SYCL [15] – модели программирования более высокого уровня для повышения производительности кода на различных аппаратных ускорителях. Это предметно-ориентированный встраиваемый язык с одним исходным кодом, основанный на чистом C++17. SYCL — это отраслевой стандарт Khronos, который добавляет параллелизм данных к C++ для гетерогенных систем. Data Parallel C++ — это проект компилятора с открытым исходным кодом, изначально созданный сотрудниками Intel для поддержки параллелизма данных в C++. Компилятор Data Parallel C++ основан на SYCL и включает несколько расширений и широкую гетерогенную поддержку для устройств GPU, CPU и FPGA.

Для всех трех технологий программирования была произведена оптимизация производительности за счет замены массивов конфликтных цветов, получаемых для шага разрешения конфликтов, на битовые сдвиги. Так как на графах, подобранных для экспериментов, массивы конфликтных цветов не были слишком велики, то каждое добавление конфликтного цвета в массив было заменено на битовый сдвиг по номеру данного цвета. Такой способ идейно был позаимствован из встроенной версии раскраски вершин библиотеки KOKKOS.

## Переносимая параллельная реализация с использованием модели KOKKOS

Для модификации параллельных алгоритмов под модель KOKKOS были использованы конструкция MDRangePolicy (встроенное средство ранжирования модели KOKKOS для распараллеливания плотно вложенных циклов), а также встроенное распараллеливание одиночного цикла. Основное отличие реализации на KOKKOS от реализации на OpenMP – KOKKOS самостоятельно распределяет работу по ресурсам исполнения. Для одиночного цикла каждая итерация принимается за единицу работы, которая идентифицируется номером итерации цикла, а диапазон цикла определяет общий объем работы. Указывается диапазон итераций и тело вычислений, далее KOKKOS определяет, как будет распределяться работа по имеющимся ресурсам. Вычислительные тела передаются как функторы. Рабочие элементы передаются шаблону и распределяются по функторам один за другим. Тело функтора имеет доступ ко всем нужным данным через элементы данных функтора. Для плотно вложенных циклов указывается размерность пространства циклов, списки инициализаторов начала и конца каждого цикла, и лямбда принимает соответствующее число индексов (Приложение А, строка 381).

## Переносимая параллельная реализация с использованием модели Data Parallel C++

Для модификации параллельных алгоритмов под модель Data Parallel C++ было использовано многомерное распараллеливание вложенных и одиночных циклов, аналогично тому, как это было сделано для модели KOKKOS. Были использованы имеющиеся в ней средства распараллеливания циклов. Так же, как и предыдущая модель, Data Parallel C++ самостоятельно распределяет объемы работы по имеющимся физическим ресурсам. Для одиночного цикла каждая итерация – это единица работы с номером, соответствующим номеру итерации, а диапазон цикла определяет общий объем работы. Требовалось указать тело функции вычислений и задать диапазон цикла. Передача вычислений была реализована через функторы. Шаблон получает объем работы и распределяет его по функторам, которые имеют доступ к данным через элементы данных функтора. Так же, как и для KOKKOS, были использованы лямбда-выражения. Дополнительно потребовалось переработать механизмы выделения и освобождения памяти для получения оптимальной производительности. Для работы с раскраской подграфов и разрешения конфликтов были разработаны специальные функции-ядра.

## Реализация алгоритма Чаталюрека для GPU

В данной работе для работы на GPU был выбран алгоритм Чаталюрека. Свойство переносимости кода, написанного на KOKKOS и Data Parallel C++ позволило не писать новую реализацию, а лишь усовершенствовать написанную для CPU.

Для всех трех технологий программирования была произведена оптимизация производительности за счет изменения структуры данных: массив конфликтных цветов, получаемых для шага разрешения конфликтов, был заменен на битовые сдвиги. Каждое добавление конфликтного цвета в массив было заменено на битовый сдвиг по номеру данного цвета. Алгоритмические преобразования алгоритма не выполнялись, но было применено кэширование данных в объемных по рабочему пространству циклах для наиболее эффективной работы с памятью.

# Программная реализация

Реализация выполнена на языке программирования C++, для распараллеливания применялись технологии OpenMP, Data Parallel C++ и KOKKOS.

## Файловая структура программы

Исходный код программы находится в следующих файлах:

1. graphio.h – заголовочный файл ввода-вывода графовой конструкции;
2. graphio.c – файл с реализацией функций ввода-вывода;
3. coloring.h – заголовочный файл с описанием функций приоритета;
4. coloring.c – файл с реализацией функций приоритета;
5. greedycoloring.c – файл с реализацией последовательного жадного алгоритма;
6. parmeths.c – файл с реализацией параллельных алгоритмов с применением технологии OpenMP;
7. parmethskokkos.c – файл с реализацией параллельных алгоритмов с применением технологии KOKKOS;
8. parmethsycl.c – файл с реализацией параллельных алгоритмов с применением технологии Data Parallel C++;
9. chatalyurekopt.c – файл с реализацией оптимизированного под GPU алгоритма Чаталюрека с применением технологии Data Parallel C++;
10. main.c – общий файл запуска алгоритмов с замерами времени.

## Структуры данных

В программе использовались следующие структуры данных для представления графа:

1. typedef struct edge; // структура ребра графа
2. typedef struct crsGraph; // структура графа на списках смежности

Структура содержит несколько полей:

* 1. int\* Adjncy; // массив номеров вершин, связанных с данной, последовательно для каждой вершины
  2. int\* Xadj; // массив индексов начала списка смежности каждой вершины в массиве Adjncy с фиктивным элементом – индексом последнего элемента массива Adjncy, увеличенным на единицу
  3. double\* Eweights; // массив весов ребер
  4. int V; // количество вершин графа

## Модульная структура программы

Программный комплекс имеет следующую структуру (Рисунок 4):

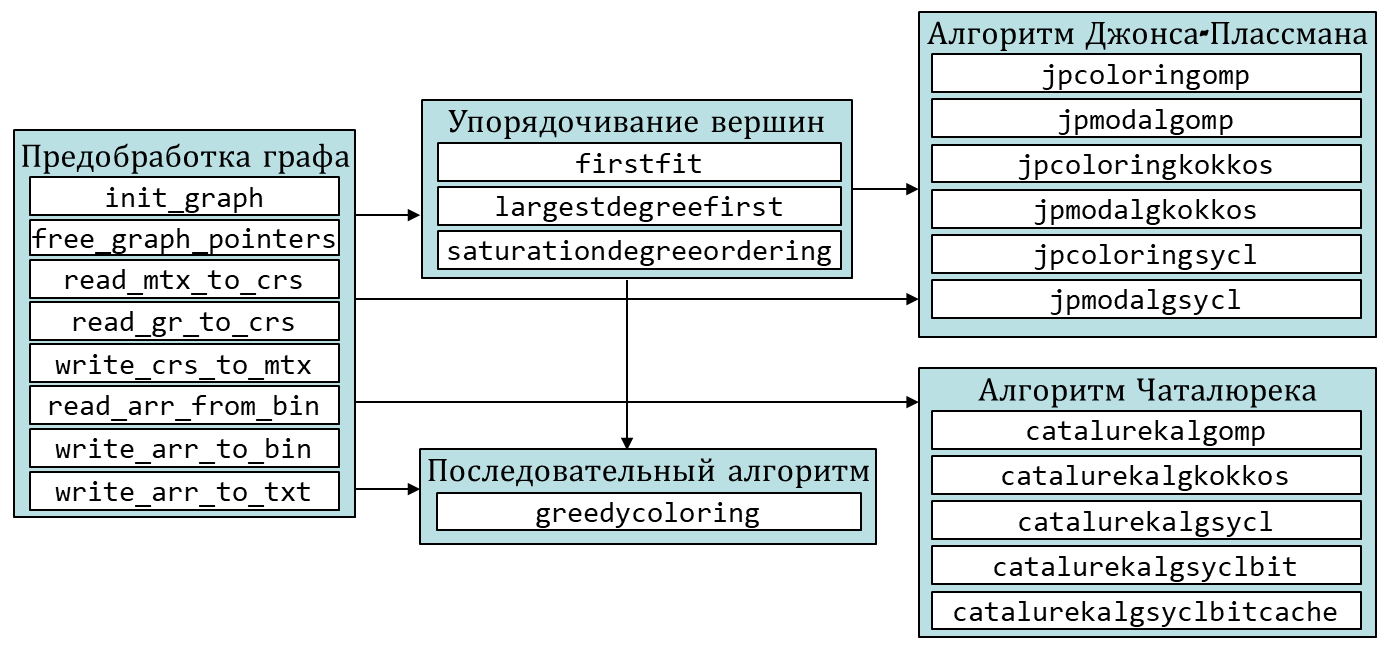


Рисунок . Структура программного комплекса

1. int init\_graph(crsGraph\* gr); // инициализация графа

*Входные параметры:* указатель на структуру графа;

*Результат:* собирает “пустой” граф, если удалось, возвращает нуль.

1. int free\_graph\_pointers(crsGraph\* gr); // освобождение указателей

*Входные параметры:* указатель на структуру графа;

*Результат:* если граф пустой, выдает сообщение и возвращает единицу, иначе освобождает все массивы структуры и возвращает нуль.

1. int read\_mtx\_to\_crs(crsGraph\* gr, const char\* filename); // чтение mtx в списки смежности

*Входные параметры:* указатель на структуру графа, строка с именем файла mtx;

*Результат:* собирает граф в структуру crs из файла mtx, если удалось, возвращает нуль.

1. int read\_gr\_to\_crs(crsGraph\* gr, const char\* filename); // чтение графа в списки смежности

*Входные параметры:* указатель на структуру графа, строка с именем файла;

*Результат:* собирает граф в структуру crs из файла, если удалось, возвращает нуль.

1. int write\_crs\_to\_mtx(crsGraph\* gr, const char\* filename); // запись графа в mtx

*Входные параметры:* указатель на структуру графа, строка с именем файла mtx;

*Результат:* записывает граф crs в файл mtx, если удалось, возвращает нуль.

1. int read\_arr\_from\_bin(double\* arr, int size, const char\* filename); // чтение массива из бинарного файла

*Входные параметры:* указатель на структуру графа, размер файла, имя файла bin;

*Результат:* собирает граф в структуру crs из файла bin, если удалось, возвращает нуль.

1. int write\_arr\_to\_bin(double\* arr, int size, const char\* filename); // запись массива в бинарный файл

*Входные параметры:* указатель на структуру графа, размер файла, имя файла bin;

*Результат:* записывает граф crs в файл bin, если удалось, возвращает нуль.

1. int write\_arr\_to\_txt(double\* arr, int size, const char\* filename); // запись массива в текстовый файл

*Входные параметры:* указатель на структуру графа, размер файла, имя файла txt;

*Результат:* записывает граф crs в файл txt, если удалось, возвращает нуль.

1. void firstfit(crsGraph& gr, int\* priority); // функция приоритета FF

*Входные параметры:* ссылка на структуру графа, массив приоритетов;

*Результат:* заполняет и возвращает массив приоритетов.

1. void largestdegreefirst(crsGraph& gr, int\* priority); // функция приоритета LDF

*Входные параметры:* ссылка на структуру графа, массив приоритетов;

*Результат:* заполняет и возвращает массив приоритетов.

1. void saturationdegreeordering(crsGraph& gr, int\* priority, int\* colors); // функция приоритета SDO

*Входные параметры:* ссылка на структуру графа, массив приоритетов;

*Результат:* заполняет и возвращает массив приоритетов.

1. void greedycoloring(crsGraph& gr, int\* priority, int\* colors); // жадная раскраска

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpcoloringomp(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Джонса-Плассмана с использованием технологии OpenMP

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpmodalgomp(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением модифицированного алгоритма Джонса-Плассмана с использованием технологии OpenMP

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void catalurekalgomp(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Чаталюрека с использованием технологии OpenMP

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpcoloringkokkos(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Джонса-Плассмана с использованием технологии KOKKOS

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpmodalgkokkos(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением модифицированного алгоритма Джонса-Плассмана с использованием технологии KOKKOS

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void catalurekalgkokkos(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Чаталюрека с использованием технологии KOKKOS

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpcoloringsycl(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Джонса-Плассмана с использованием технологии Data Parallel C++

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void jpmodalgsycl(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением модифицированного алгоритма Джонса-Плассмана с использованием технологии Data Parallel C++

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void catalurekalgsycl(crsGraph& gr, int\* priority, int\* colors); // раскраска с применением алгоритма Чаталюрека с использованием технологии Data Parallel C++

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void catalurekalgsyclbit(crsGraph& gr, int\* priority, int\* colors); // модифицированная раскраска с применением алгоритма Чаталюрека с использованием технологии Data Parallel C++ и битовых операций для массива конфликтных цветов

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

1. void catalurekalgsyclbitcache(crsGraph& gr, int\* priority, int\* colors); // модифицированная раскраска с применением алгоритма Чаталюрека с использованием технологии Data Parallel C++, битовых операций для массива конфликтных цветов и кэширования

*Входные параметры:* ссылка на структуру графа, массив приоритетов, массив цветов;

*Результат:* заполняет и возвращает массив цветов.

# Результаты экспериментов

## Методика проведения экспериментов

Тестирование проводилось на графах с количеством вершин от 0.25 млн до 14.76 млн из коллекции Suite Sparse Matrix Collection [16] (Таблица 1):

Таблица . Сведения о тестовых графах

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер графа** | **Название графа** | **Число вершин графа** | **Число ребер графа** |
| **1** | one34by | 259 789 | 1 052 008 |
| **2** | bcsstk12 | 715 176 | 1 344 024 |
| **3** | bcsstk3 | 952 203 | 3 855 918 |
| **4** | compronb | 1 391 349 | 5 634 226 |
| **5** | will12 | 1 564 794 | 6 023 014 |
| **6** | pre36 | 5 012 457 | 19 293 338 |
| **7** | raw34 | 6 456 202 | 26 144 202 |
| **8** | dersame | 8 222 012 | 33 052 488 |
| **9** | mianse2 | 11 054 532 | 44 765 002 |
| **10** | nu16ddk | 14 758 344 | 59 763 484 |

Вычислительные эксперименты проводились на трех тестовых системах. Их характеристики приведены ниже.

Тестовая система 1: домашний компьютер с центральным процессором AMD E2-7110 APU (1800 MHz, 4 GB);

Тестовая система 2: узел кластера Intel DevCloud с центральным процессором Intel Xeon Gold (2400 MHz, 192 GB);

Тестовая система 3: узел кластера Intel DevCloud с дискретным графическим процессором Intel Iris XE Max (1650 MHz, 4 GB, 68 GB/s).

Для наглядного сравнения были собраны результаты работы алгоритмов из таких библиотек, как последовательная и параллельная версии ColPack [17] и параллельная графовая библиотека Kokkos [14]. Корректность раскраски в каждом случае была проверена сравнением с результатом запуска библиотечной функции и по определению корректной раскраски.

Введем обозначения, использованные при анализе таблиц:

1. *SeqFF* – последовательный алгоритм с подстановкой функции First Fit;
2. *SeqLDF* – последовательный алгоритм с подстановкой функции Largest Degree First;
3. *SeqSDO* – последовательный алгоритм с подстановкой функции Saturation Degree Ordering;
4. *SeqColPack* – встроенная реализация раскраски последовательной версии библиотеки ColPack;
5. *JP* – параллельный алгоритм Джонса-Плассмана;
6. *ModJP* – модифицированный алгоритм Джонса-Плассмана;
7. *Catalyurek* – алгоритм Чаталюрека;
8. *ParColPack* – встроенная реализация раскраски параллельной версии библиотеки ColPack;
9. *KokkosVB* – встроенная реализация раскраски графовой библиотеки Kokkos;
10. *Kokkos* – собственная реализация алгоритма Чаталюрека на KOKKOS;
11. *DPC* – собственная реализация алгоритма Чаталюрека на Data Parallel C++;
12. *KokkosVBBIT* – встроенная реализация раскраски с использованием битовых операций из встроенной библиотеки KOKKOS;
13. *BitKokkos* – собственная реализация алгоритма Чаталюрека с битовой модификацией на KOKKOS;
14. *BitDPC* – собственная реализация алгоритма Чаталюрека с битовой модификацией на Data Parallel C++;
15. *CacheBitKokkos* – собственная реализация алгоритма Чаталюрека с битовой модификацией и кэшированием на KOKKOS;
16. *CacheBitDPC* – собственная реализация алгоритма Чаталюрека с битовой модификацией и кэшированием на Data Parallel C++;
17. *T* – время работы в секундах;
18. *С* – количество цветов раскраски.

## Последовательный жадный алгоритм

Был проведен запуск реализации последовательного жадного алгоритма с поочередной подстановкой функций приоритета First Fit, Largest Degree First и Saturation Degree Ordering на тестовой системе 1 для сравнения эффективностей всех трех случаев.

По результатам экспериментов на тестовой системе 1, представленным далее (Таблица 2), видно, что самым качественным способом последовательной раскраски является функция упорядочивания вершин по степени насыщенности (SDO). С другой стороны, эта функция является самой медленной, две другие на больших графах дают заметно более быстрые результаты, но количество цветов на них выше. Это объясняется вычислительной сложностью представленных функций: Saturation Degree Ordering (SDO) – , Largest Degree First (LDF) и First Fit (FF) – .

Таблица . Сравнение времени работы последовательных алгоритмов (в секундах). Тестовая система 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **SeqFF** | | **SeqLDF** | | **SeqSDO** | | **SeqColPack** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **1** | 0.41 | 39 | 0.41 | 39 | 0.90 | 37 | 0.41 | 39 |
| **2** | 0.31 | 19 | 0.40 | 19 | 0.71 | 17 | 0.40 | 18 |
| **3** | 8.82 | 93 | 8.71 | 93 | 11.32 | 89 | 8.71 | 93 |
| **4** | 12.90 | 109 | 12.90 | 110 | 17.20 | 102 | 12.89 | 109 |
| **5** | 36.90 | 157 | 37.10 | 157 | 68.70 | 154 | 36.78 | 157 |

Замедление реализации с функцией приоритета Saturation Degree Ordering относительно реализаций с функциями приоритета Largest Degree First и First Fit составляет примерно 1.85 раза, улучшение по качеству раскраски – до 27%.

Лучшими по времени последовательными алгоритмами оказались First Fit и Largest Degree First, по качеству раскраски – Saturation Degree Ordering. При сравнении с последовательной реализацией библиотеки ColPack скорость и качество раскраски сходны с данными алгоритмов First Fit и Largest Degree First.

## Параллельные алгоритмы

Был проведен запуск реализаций на OpenMP параллельных алгоритмов Чаталюрека и Джонса-Плассмана с поочередной подстановкой функций приоритета First Fit и Largest Degree First и встроенной функции раскраски параллельной библиотеки ColPack на тестовой системе 1 для сравнения их эффективностей.

Подстановка функции приоритета Largest Degree First в алгоритм Джонса-Плассмана оказалась довольно эффективной и ускорила стандартную версию алгоритма (Таблица 3). Было получено ускорение до двух раз на маленьких графах и до 20% на графах большой размерности при приближенно одинаковом количестве цветов. Функции Saturation Degree Ordering и First Fit не имеют практического применения в параллельных алгоритмах, так как при их подстановке полученная последовательно производительность ухудшается.

Результаты сравнения собственных реализаций параллельных алгоритмов раскраски и реализации из библиотеки ColPack на тестовой системе 1 приведены в следующей таблице (Таблица 3):

Таблица . Сравнение времени работы параллельных алгоритмов на тестовой системе 1. Время указано в секундах

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **JP** | | **ModJP** | | **Catalyurek** | | **ParColPack** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **1** | 0.11 | 39 | 0.07 | 37 | 0.03 | 37 | 0.07 | 37 |
| **2** | 0.10 | 19 | 0.05 | 17 | 0.03 | 19 | 0.05 | 19 |
| **3** | 1.50 | 92 | 1.45 | 89 | 0.70 | 92 | 1.00 | 92 |
| **4** | 2.22 | 107 | 1.90 | 107 | 0.90 | 107 | 1.30 | 107 |
| **5** | 5.23 | 156 | 5.10 | 156 | 2.94 | 156 | 3.20 | 156 |

Из параллельных реализаций алгоритм Чаталюрека оказался наиболее быстрым, при этом качество раскраски для всех алгоритмов близко. Ускорение относительно стандартного алгоритма Джонса-Плассмана составило до трех раз, относительно модифицированного алгоритма – около двух раз. При сравнении с параллельной версией библиотеки ColPack было выявлено, что реализация алгоритма Чаталюрека работает быстрее на 30%.

При тестировании параллельных алгоритмов на тестовой системе 2 были получены следующие результаты (Таблица 4):

Таблица . Сравнение времени работы параллельных алгоритмов на тестовой системе 2.

Время указано в секундах

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **JP** | | **ModJP** | | **Catalyurek** | | **ParColPack** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **4** | 0.46 | 107 | 0.37 | 106 | 0.19 | 107 | 0.42 | 106 |
| **5** | 0.83 | 155 | 0.81 | 154 | 0.66 | 155 | 0.91 | 156 |
| **6** | 1.57 | 401 | 1.39 | 399 | 0.83 | 401 | 1.01 | 400 |
| **7** | 1.76 | 424 | 1.67 | 424 | 0.87 | 424 | 1.12 | 424 |
| **8** | 2.59 | 430 | 2.55 | 428 | 1.11 | 428 | 1.40 | 430 |

Результаты работы на тестовой системе 2 показали, что графы размером меньше миллиона вершин не дают значимых времен, по которым возможно бы было строить какие-то выводы, поэтому были взяты графы больших размерностей.

Самым быстрым оказался алгоритм Чаталюрека: до двух раз быстрее относительно двух других алгоритмов и до 20% быстрее алгоритма из библиотеки ColPack. Качество раскраски получилось приближенно одинаковым для всех алгоритмов.

Далее, при помощи профилировщика было получено соотношение времени, потраченного программой на разрешение конфликтов, и времени, потраченного на раскраску.Если рассматривать зависимости времени разрешения конфликтов от числа потоков, задействованных программой, то получим, что на маленьких графах мы наблюдаем небольшое уменьшение числа конфликтов с ростом числа потоков, в то время как на сравнительно больших графах – рост числа конфликтов с ростом числа запущенных в работу потоков. Наглядно зависимость отражена следующей диаграммой (Рисунок 5):

Рисунок . Доля времени разрешения конфликтов от общего времени работы программы

Чтобы сделать вывод о наиболее или наименее эффективном алгоритме, был построен график ускорений относительно самой быстрой из последовательных реализаций (Рисунок 6):

Рисунок . Ускорение параллельных версий от числа вершин графа

Таким образом, алгоритм Джонса-Плассмана и модифицированный алгоритм дают примерно одинаковое ускорение на одинаковых графах – от 3.1 до 7.8 раз, а алгоритм Чаталюрека дает заметно лучшее ускорение – до 14.2 раз.

Для сравнения качества раскраски был построен график количества цветов в зависимости от размера графа (Рисунок 7):

Рисунок . Количество цветов раскраски от числа вершин графа

Из данного рисунка видно, что количество цветов раскраски приблизительно совпадает для всех приведенных алгоритмов.

## Реализация с использованием KOKKOS и Data Parallel C++

Был проведен запуск реализаций на KOKKOS и Data Parallel C++ параллельных алгоритмов Чаталюрека и Джонса-Плассмана с поочередной подстановкой функций приоритета First Fit и Largest Degree First и встроенной функции раскраски параллельной графовой библиотеки KOKKOS на тестовой системе 2 для сравнения производительностей.

При тестировании реализации на KOKKOS параллельных алгоритмов на тестовой системе 2 были получены следующие результаты (Таблица 5):

Таблица . Сравнение времени работы параллельных алгоритмов на тестовой системе 2 для Kokkos.

Время указано в секундах

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **JP** | | **ModJP** | | **Catalyurek** | | **KokkosVB** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **4** | 0.46 | 107 | 0.37 | 106 | 0.19 | 107 | 0.17 | 107 |
| **5** | 0.83 | 155 | 0.81 | 154 | 0.66 | 155 | 0.59 | 155 |
| **6** | 1.57 | 401 | 1.39 | 399 | 0.83 | 401 | 0.73 | 401 |
| **7** | 1.76 | 424 | 1.67 | 424 | 0.87 | 424 | 0.78 | 424 |
| **8** | 2.59 | 430 | 2.55 | 428 | 1.11 | 428 | 0.98 | 428 |

Из данной таблицы видно, что реализация на KOKKOS не уступает в производительности реализации на OpenMP при одинаковом качестве раскраски.

Результаты работы показали, что время работы реализации из встроенной библиотеки KOKKOS оказались ближе всего к времени работы собственной реализации алгоритма Чаталюрека. На основе этого можно сделать предположение, что в графовой библиотеке KOKKOS раскраска реализована на основе именно этого алгоритма. При анализе открытого кода данная гипотеза подтвердилась.

При тестировании реализации на Data Parallel C++ параллельных алгоритмов на тестовой системе 2 были получены следующие результаты (таблица 6Таблица 6):

Таблица . Сравнение времени работы параллельных алгоритмов на тестовой системе 2 для Data Parallel C++.

Время указано в секундах

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **JP** | | **ModJP** | | **Catalyurek** | |
| **T** | **C** | **T** | **C** | **T** | **C** |
| **4** | 0.46 | 107 | 0.37 | 106 | 0.19 | 107 |
| **5** | 0.83 | 155 | 0.81 | 154 | 0.66 | 155 |
| **6** | 1.57 | 401 | 1.39 | 399 | 0.83 | 401 |
| **7** | 1.76 | 424 | 1.67 | 424 | 0.87 | 424 |
| **8** | 2.59 | 430 | 2.55 | 428 | 1.11 | 428 |

Из данной таблицы видно, что реализация на Data Parallel C++ работает аналогично реализации на KOKKOS и по производительности, и по качеству раскраски. Это дает нам право выдвинуть гипотезу о том, что данные модели программирования схожи по внутренней структуре.

Далее был проведен сравнительный анализ реализаций каждого алгоритма с применением всех трех моделей программирования (Рисунок 8, Рисунок 9, Рисунок 10):

Рисунок . Время работы реализации алгоритма Джонса-Плассмана с применением OpenMP, Kokkos и Data Parallel C++

Рисунок . Время работы реализации модифицированного алгоритма Джонса-Плассмана с применением OpenMP, Kokkos и Data Parallel C++

Рисунок . Время работы реализации алгоритма Чаталюрека с применением OpenMP, Kokkos и Data Parallel C++

По результатам сравнения времени работы реализаций на OpenMP, KOKKOS и Data Parallel C++ во всех трех случаях можно сделать вывод, что применение всех этих технологий на CPU сопоставимо по скорости и качеству, так как при равных временах были получены одинаковые количества цветов раскраски.

## Запуск реализаций алгоритма Чаталюрека на GPU

Был проведен запуск реализаций на KOKKOS и Data Parallel C++ параллельного алгоритма Чаталюрека и встроенной функции раскраски параллельной графовой библиотеки KOKKOS на тестовой системе 3 для анализа эффективности этих реализаций при запусках на графическом процессоре, а также запуск их модифицированных версий на тестовой системе 2 для сравнения результатов оптимизации на центральном и графическом процессорах.

При тестировании на тестовой системе 3 реализаций алгоритма Чаталюрека на KOKKOS и Data Parallel C++ были получены следующие результаты (таблица 7):

Таблица . Сравнение времени работы параллельных реализаций алгоритма Чаталюрека на тестовой системе 3.

Время указано в секундах

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **Kokkos** | | **DPC** | | **KokkosVB** | | **KokkosVBBIT** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **8** | 1.13 | 428 | 1.13 | 428 | 1.04 | 428 | 0.70 | 428 |
| **9** | 2.05 | 623 | 2.05 | 623 | 1.88 | 623 | 1.27 | 623 |
| **10** | 2.20 | 814 | 2.20 | 814 | 2.02 | 814 | 1.34 | 814 |

По результатам тестирования было выявлено, что встроенная раскраска библиотеки KOKKOS с применением битовых сдвигов дает ускорение около 30% по сравнению с раскраской с применением массивов конфликтных цветов. Вследствие этого было решено внедрить битовые операции в собственные реализации раскраски на KOKKOS и Data Parallel C++.

При тестировании реализаций с модификацией на тестовой системе 3 были получены следующие результаты (таблица 8):

Таблица . Сравнение времени работы модифицированных параллельных реализаций алгоритма Чаталюрека на тестовой системе 3.

Время указано в секундах

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **BitKokkos** | | **BitDPC** | | **KokkosVB** | | **KokkosVBBIT** | |
| **T** | **C** | **T** | **C** | **T** | **C** | **T** | **C** |
| **8** | 0.76 | 428 | 0.76 | 428 | 1.04 | 428 | 0.70 | 428 |
| **9** | 1.39 | 623 | 1.39 | 623 | 1.88 | 623 | 1.27 | 623 |
| **10** | 1.46 | 814 | 1.46 | 814 | 2.02 | 814 | 1.34 | 814 |

Из приведенной таблицы видно, что при помощи битовых операций удалось получить ускорение относительно предыдущей реализации на 50% и приблизиться к производительности реализации алгоритма раскраски из библиотеки KOKKOS с битовыми массивами.

Далее было принято решение провести оптимизацию полученных реализаций с учетом требований архитектуры графического процессора – модифицировать работу с памятью. Было добавлено кэширование данных в циклах, сокращающее число дорогостоящих копирований данных. Результаты тестирования полученных реализаций на тестовой системе 3 представлены в следующей таблице (таблица 9):

Таблица . Сравнение времени работы модифицированных параллельных реализаций алгоритма Чаталюрека с кэшированием на тестовой системе 3.

Время указано в секундах

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **CacheBitKokkos** | | **CacheBitDPC** | | **KokkosVBBIT** | |
| **T** | **C** | **T** | **C** | **T** | **C** |
| **8** | 0.59 | 428 | 0.59 | 428 | 0.70 | 428 |
| **9** | 1.07 | 623 | 1.07 | 623 | 1.27 | 623 |
| **10** | 1.12 | 814 | 1.12 | 814 | 1.34 | 814 |

Из данной таблицы видно, что при помощи кэширования удалось получить ускорение относительно предыдущей реализации около 30%, ускорение относительно встроенной раскраски библиотеки KOKKOS с битовыми массивами – около 20%. Было выдвинуто предположение, что такое ускорение было обеспечено более точным определением размеров кэша. Качество раскраски при этом не изменилось.

При запусках конечной версии на тестовой системе 2 были получены следующие результаты (таблица 10):

Таблица . Сравнение времени работы модифицированных параллельных реализаций алгоритма Чаталюрека с кэшированием на тестовой системе 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Номер графа** | **CacheBitKokkos** | | **CacheBitDPC** | | **KokkosVBBIT** | |
| **T** | **C** | **T** | **C** | **T** | **C** |
| **8** | 0.67 | 428 | 0.67 | 428 | 0.66 | 428 |
| **9** | 1.22 | 623 | 1.22 | 623 | 1.20 | 623 |
| **10** | 1.29 | 814 | 1.29 | 814 | 1.27 | 814 |

Таким образом, за счет кэширования удалось получить ускорение и для центрального процессора. Качество раскраски остается неизменным.

Для всех реализаций были собраны результаты на графах с количеством вершин от 8.22 млн до 14.76 млн на центральном и графическом процессоре и для наглядного представления результатов построены следующие графики (Рисунок 11, Рисунок 12):

Рисунок . Время работы всех реализаций алгоритма Catalyurek на центральном процессоре от числа вершин графа

Рисунок . Время работы реализаций на KOKKOS и Data Parallel C++ алгоритма Catalyurek на графическом процессоре от числа вершин графа

Из графиков видно, что реализации одного типа на конкретном виде процессора близки друг к другу по времени вне зависимости от технологии, которая применялась при реализации.

# Заключение

В работе были рассмотрены и реализованы последовательные и параллельные алгоритмы раскраски графов с использованием различных функций приоритета. Были получены результаты вычислительных экспериментов на центральном и графическом процессорах. Произведено сравнение времени работы и эффективности полученных реализаций.

Было рассмотрено два типа параллельных алгоритмов: алгоритмы, основанные на раскраске независимого множества вершин, и алгоритмы, основанные на случайной раскраске вершин с последующим исправлением конфликтов. Производительность параллельных алгоритмов первого типа ограничена затратами на коммуникацию: чем больше порядок графа, тем больше ребер, не локальных для процессов и, соответственно, больше общее время работы алгоритма. Для алгоритмов второго типа число конфликтов раскраски сравнимо близко при использовании до 32 потоков в параллельных запусках. Число цветов, используемое алгоритмами первого и второго типов, одинаково.

В ходе тестирования было выявлено, что самым быстрым из приведенных параллельных алгоритмов является алгоритм Чаталюрека, самым быстрым из приведенных последовательных – жадный алгоритм с функциями приоритета First Fit и Largest Degree First.

Были созданы и протестированы реализации алгоритмов Джонса-Плассмана и Чаталюрека с использованием различных моделей программирования: OpenMP, KOKKOS и Data Parallel C++. По результатам запусков было выявлено, что все три модели программирования позволяют создать очень близкие по времени версии параллельных алгоритмов. Сравнение собственных параллельных реализаций алгоритма Чаталюрека с реализацией, встроенной в библиотеку KOKKOS, показало, что они близки по производительности.

Был произведен запуск реализаций на KOKKOS и Data Parallel C++ алгоритма Чаталюрека на GPU. При запуске на графическом процессоре было выявлено, что за счет битовых операций и кэширования можно существенно оптимизировать исходную реализацию и получить ускорение на 20-30%. Кроме того, было показано, что при запуске модифицированной для графического процессора версии на центральном процессоре результаты также улучшаются.

В целом проведенные эксперименты показывают, что при решении задачи раскраски графа средства для переносимого гетерогенного программирования KOKKOS и Data Parallel C++ (SYCL) показали схожие результаты, практически не уступили комбинации С++ и OpenMP в тех случаях, когда такое сравнение возможно. При этом при решении данной задачи они позволили использовать код для расчетов и на центральных, и на графических процессорах без существенной потери производительности. Прямое сравнение производительности CPU и GPU в данном случае затруднено, поскольку данные устройства обладают разными вычислительными возможностями. Тем не менее, результаты работы на GPU выглядят по крайней мере приемлемыми.

# Список литературы

1. Gross J.L., Yellen J., Anderson M. Graph theory and its applications. – Chapman and Hall/CRC, 2018.
2. Deveci M. et al. Parallel graph coloring for manycore architectures //2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). – IEEE, 2016. – С. 892-901.
3. Brélaz D. New methods to color the vertices of a graph //Communications of the ACM. – 1979. – Т. 22. – №. 4. – С. 251-256.
4. Karp R.M. Reducibility among combinatorial problems //Complexity of computer computations. – Springer, Boston, MA, 1972. – С. 85-103.
5. Зыков А.А. О некоторых свойствах линейных комплексов //Математический сборник. – 1949. – Т. 24. – №. 2. – С. 163-188.
6. Boman E.G. et al. A scalable parallel graph coloring algorithm for distributed memory computers //European Conference on Parallel Processing. – Springer, Berlin, Heidelberg, 2005. – С. 241-251.
7. Çatalyürek Ü.V. et al. Graph coloring algorithms for multi-core and massively multithreaded architectures //Parallel Computing. – 2012. – Т. 38. – №. 10-11. – С. 576-594.
8. Jones M.T., Plassmann P.E. A parallel graph coloring heuristic //SIAM Journal on Scientific Computing. – 1993. – Т. 14. – №. 3. – С. 654-669.
9. Allwright J.R. et al. A comparison of parallel graph coloring algorithms //SCCS-666. – 1995. – С. 1-19.
10. Normann P. Parallel graph coloring: Parallel graph coloring on multi-core CPUs. – 2014.
11. Cohen J., Castonguay P. Efficient graph matching and coloring on the gpu //GPU Technology Conference. – 2012. – С. 1-10.
12. Grosset A.V.P. et al. Evaluating graph coloring on GPUs //Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. – 2011. – С. 297-298.
13. Li P. et al. High performance parallel graph coloring on GPGPUs //2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). – IEEE, 2016. – С. 845-854.
14. KOKKOS C++ Performance Portability Programming EcoSystem – <https://github.com/kokkos>
15. Data Parallel C++: the oneAPI Implementation of SYCL – [https://www.intel.com/content/www/us/en/developer/tools/oneAPI/data-parallel-c-plus-plus.html](https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html)
16. Suite Sparse Matrix Collection – <https://sparse.tamu.edu/>
17. Библиотека ColPack – <https://cscapes.cs.purdue.edu/coloringpage/>

# Приложение А. Фрагменты кода программы

int init\_graph(crsGraph\* gr) {

gr.Adjncy = NULL;

gr.Xadj = NULL;

gr.Eweights = NULL;

return 0;

}

int free\_graph\_pointers(crsGraph\* gr) {

free(gr.Adjncy);

free(gr.Xadj);

free(gr.Eweights);

gr.Adjncy = NULL;

gr.Xadj = NULL;

gr.Eweights = NULL;

return 0;

}

int read\_mtx\_to\_crs(crsGraph\* gr, const char\* filename) {

int N, i, row, col, nz\_size;

int \*edge\_num, \*last\_el;

double val;

fpos\_t position;

FILE \*file;

last\_el = (int\*)malloc(sizeof(int) \* gr.V);

edge\_num = (int\*)malloc(sizeof(int) \* gr.V);

for (i = 0; i < (gr.V); i++)

edge\_num[i] = 0;

nz\_size = gr.nz;

fgetpos(file, &position);

for(i = 0; i < nz\_size; i++) {

fscanf(file, "%d %d %lg", &row, &col, &val);

row--;

col--;

if (row == col) {

gr.nz--;

continue;

}

edge\_num[row]++;

if (mm\_is\_symmetric(gr.matcode)) {

edge\_num[col]++;

gr.nz ++;

}

}

if ((gr.Adjncy != NULL) || (gr.Xadj != NULL) || (gr.Eweights != NULL))

free\_graph\_pointers(gr);

gr.Adjncy = (int\*)malloc(sizeof(int) \* (gr.nz));

gr.Xadj = (int\*)malloc(sizeof(int) \* ((gr.V) + 1));

gr.Eweights = (double\*)malloc(sizeof(double) \* (gr.nz));

gr.Xadj[0] = 0;

for(i = 0; i < gr.V; i++) {

gr.Xadj[i + 1] = gr.Xadj[i] + edge\_num[i];

last\_el[i] = gr . Xadj[i];

}

fsetpos(file, &position);

for(i = 0; i < nz\_size; i++) {

fscanf(file, "%d %d %lg", &row, &col, &val);

row--;

col--;

if (row == col)

continue;

gr.Adjncy[last\_el[row]] = col;

gr.Eweights[last\_el[row]] = val;

last\_el[row]++;

if (mm\_is\_symmetric(gr.matcode)) {

gr.Adjncy[last\_el[col]] = row;

gr.Eweights[last\_el[col]] = val;

last\_el[col]++;

}

}

free(edge\_num);

free(last\_el);

fclose(file);

return 0;

}

int read\_gr\_to\_crs(crsGraph\* gr, const char\* filename) {

int i, row, col;

int \*edge\_num, \*last\_el;

double val;

char sym = 'c';

char str[101];

fpos\_t position;

FILE \*file;

while (sym == 'c') {

sym = fgetc(file);

if (sym == 'p') {

fscanf(file, "%100s %d %d", str, &gr.V, &gr.nz);

fgets(str, sizeof(str), file);

fgetpos(file, &position);

} else

fgets(str, sizeof(str), file);

}

last\_el = (int\*)malloc(sizeof(int) \* gr.V);

edge\_num = (int\*)malloc(sizeof(int) \* gr.V);

for (i = 0; i < (gr . V); i++)

edge\_num[i] = 0;

while ((sym = fgetc(file)) != EOF) {

if (sym == 'a') {

fscanf(file, "%d %d %lg", &row, &col, &val);

row--;

col--;

if (row == col)

gr . nz --;

else

edge\_num[row]++;

}

fgets(str, sizeof(str), file);

}

if ((gr.Adjncy != NULL) || (gr.Xadj != NULL) || (gr.Eweights != NULL))

free\_graph\_pointers(gr);

gr.Adjncy = (int\*)malloc(sizeof(int) \* (gr.nz));

gr.Xadj = (int\*)malloc(sizeof(int) \* ((gr.V) + 1));

gr.Eweights = (double\*)malloc(sizeof(double) \* (gr.nz));

gr.Xadj[0] = 0;

for(i = 0; i < gr . V; i++) {

gr.Xadj[i+1] = gr.Xadj[i] + edge\_num[i];

last\_el[i] = gr.Xadj[i];

}

fsetpos(file, &position);

while ((sym = fgetc(file)) != EOF) {

if (sym == 'a'){

fscanf(file, "%d %d %lg", &row, &col, &val);

row--;

col--;

if (row == col) {

fgets(str, sizeof(str), file);

continue;

}

gr.Adjncy[last\_el[row]] = col;

gr.Eweights[last\_el[row]] = val;

last\_el[row]++;

fgets(str, sizeof(str), file);

} else

fgets(str, sizeof(str), file);

}

free(edge\_num);

free(last\_el);

fclose(file);

return 0;

}

int write\_crs\_to\_mtx(crsGraph\* gr, const char\* filename) {

int i,j;

FILE\* f;

mm\_write\_banner(f, gr.matcode);

if(mm\_is\_symmetric(gr.matcode))

mm\_write\_mtx\_crd\_size(f, gr.V, gr.V, gr.nz/2);

else

mm\_write\_mtx\_crd\_size(f, gr.V, gr.V, gr.nz);

for(i = 0; i < gr.V; i++)

for(j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++)

if (i > gr.Adjncy[j] || !mm\_is\_symmetric(gr.matcode))

fprintf(f, "%d %d %lg\n", i + 1, gr.Adjncy[j] + 1, gr.Eweights[j]);

fclose(f);

return 0;

}

int read\_arr\_from\_bin(double\* arr, int size, const char\* filename) {

int result;

FILE\* file = fopen(filename, "rb");

result = fread(arr, sizeof(double), size, file);

fclose(file);

if (result == size)

return 0;

else

return 1;

}

int write\_arr\_to\_bin(double\* arr, int size, const char\* filename) {

FILE\* file = fopen(filename, "wb");

fwrite(arr, sizeof(double), size, file);

fclose(file);

return 0;

}

int write\_arr\_to\_txt(double\* arr, int size, const char\* filename) {

int i;

FILE\* file = fopen(filename, "w");

for(i = 0; i < size; i++)

fprintf(file, "%lg\n", arr[i]);

fclose(file);

return 0;

}

void firstfit(crsGraph& gr, int\* priority) {

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

priority[i] = i;

}

void largestdegreefirst(crsGraph& gr, int\* priority) {

int index = 0;

int\* degree = new int[gr.V];

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

degree[i] = gr.Xadj[i + 1] - gr.Xadj[i];

for (int i = 0; i < gr.V; i++) {

int max = -1;

for (int j = 0; j < gr.V; j++)

if (degree[j] >= max) {

max = degree[j];

index = j;

}

priority[i] = index;

degree[index] = -1;

}

delete[] degree;

}

void saturationdegreeordering(crsGraph &gr, int\* priority, int\* colors) {

vector<int> indicator;

int\* saturation = new int[gr.V];

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

saturation[i] = 0;

largestdegreefirst(gr, priority);

colors[priority[0]] = 0;

for (int i = gr.Xadj[priority[0]]; i < gr.Xadj[priority[0] + 1]; i++)

saturation[i]++;

saturation[priority[0]] = -100;

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

if (i != priority[0]) {

colors[i] = -1;

indicator.push\_back(i);

}

while (!indicator.empty()) {

int sat = -1;

int satnum = 0;

vector<int> satcol;

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

if (saturation[i] > sat) {

sat = saturation[i];

satnum = i;

satcol.clear();

for (int j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++)

satcol.push\_back(colors[j]);

}

for (vector<int>::iterator it = indicator.begin(); it != indicator.end(); it++)

if ((\*it) == satnum) {

indicator.erase(it);

break;

}

int mincol = 0;

int ready = 0;

while (!ready) {

ready = 1;

for (int i = 0; i < satcol.size(); i++)

if (satcol[i] == mincol) {

mincol++;

ready = 0;

}

}

colors[satnum] = mincol;

for (int i = gr.Xadj[satnum]; i < gr.Xadj[satnum + 1]; i++)

if (saturation[i] != -100)

saturation[i]++;

saturation[satnum] = -100;

}

delete[] saturation;

}

void greedycoloring(crsGraph &gr, int\* priority, int\* colors) {

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

colors[i] = -1;

bool\* available = new bool[gr.V];

for (int ac = 0; ac < gr.V; ac++)

available[ac] = 0;

for (int i = 1; i < gr.V; i++) {

for (int j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++)

if (colors[j] != -1)

available[colors[j]] = 1;

int ac;

for (ac = 1; ac <= gr.V; ac++)

if (available[ac] == 0)

break;

colors[i] = ac;

for (int j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++)

if (colors[j] != -1)

available[colors[j]] = 0;

}

delete[] available;

}

void catalurekalgomp(crsGraph& gr, int\* priority, int\* colors) {

int mconfl = -1;

int \*conflv = new int[N]();

bool \*isdet = new bool[N]();

static bool \*isused;

int cnt = 0;

for (int i = 0; i < N; ++i) {

int ecnt = gr.Xadj[i + 1] – gr.Xadj[i];

if (ecnt > cnt)

cnt = ecnt;

}

#pragma omp threadprivate(isused)

#pragma omp parallel

{

isused = new bool[cnt + 1]();

}

int c;

#pragma omp parallel for private(c)

for (int i = 0; i < N; i++) {

for (int j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++) {

int cc = colors[gr.Xadj[j]];

if (cc >= 0)

isused[cc] = true;

}

for (int j = 0; j < cnt + 1; j++)

if (!isused[j]) { for (int k = gr. Xadj[j]; k < gr. Xadj[j + 1]; k++) {

int ccc = colors[gr.Xadj[k]];

if (ccc >= 0)

isused[ccc] = false;

}

return j;

}

colors[i] = c;

}

int confc = 0;

do {

int c, conflin, temp;

int i, j, k;

#pragma omp parallel for private(j, k, c, conflin, temp)

for (i = 0; i < N; i++) {

c = colors[i];

for (j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++) {

if (colors[gr.Xadj[j]] == c) {

conflin = i < gr.Xadj[j] ? i : gr.Xadj[j];

if (!isdet[conflin]) {

isdet[conflin] = true;

#pragma omp atomic capture

temp = confc++;

conflv[temp] = conflin;

}

}

}

}

#pragma omp parallel for

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

isdet[conflv[i]] = false;

#pragma omp for private(c)

for (int i = 0; i < confc; i++) {

for (int c = gr.Xadj[i]; c < gr.Xadj[i + 1]; c++) {

int p = colors[gr.Xadj[c]];

if (p >= 0)

isused[p] = true;

}

for (int c = 0; c < cnt + 1; c++)

if (!isused[c]) {

for (int z = gr.Xadj[conflv[i]]; z < gr.Xadj[conflv[i] + 1]; z++) {

int cq = colors[gr.Xadj[z]];

if (cq >= 0)

isused[cq] = false;

}

}

colors[conflv[i]] = c;

}

++mconfl;

} while (confc > 0);

delete[] isdet;

delete[] conflv;

#pragma omp parallel

{

delete[] isused;

}

}

void catalurekalgkokkos(crsGraph& gr, int\* priority, int\* colors) {

int mconfl = -1;

int \*conflv = new int[N]();

bool \*isdet = new bool[N]();

static bool \*isused;

int cnt = 0;

for (int i = 0; i < N; ++i) {

int ecnt = gr.Xadj[i + 1] – gr.Xadj[i];

if (ecnt > cnt)

cnt = ecnt;

}

int c;

parallel\_for("ISUSED", N, [=](int i) {

isused = new bool[cnt + 1]();

for (int j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++) {

int cc = colors[gr.Xadj[j]];

if (cc >= 0)

isused[cc] = true;

}

for (int j = 0; j < cnt + 1; j++)

if (!isused[j]) {

for (int k = gr. Xadj[j]; k < gr. Xadj[j + 1]; k++) {

int ccc = colors[gr.Xadj[k]];

if (ccc >= 0)

isused[ccc] = false;

}

return j;

}

colors[i] = c;

});

int confc = 0;

do {

int c, conflin, temp;

int i, j, k;

parallel\_for("SOLVER", N, [=](int i) {

c = colors[i];

for (j = gr.Xadj[i]; j < gr.Xadj[i + 1]; j++) {

if (colors[gr.Xadj[j]] == c) {

conflin = i < gr.Xadj[j] ? i : gr.Xadj[j];

if (!isdet[conflin]) {

isdet[conflin] = true;

atomic\_add(temp, confc + 1);

atomic\_add(conflv[temp], conflin);

}

}

}

});

parallel\_for("ISDET", confc, [=](int i) {

isdet[conflv[i]] = false;

});

parallel\_for("ISUSED", confc, [=](int i) {

for (int c = gr.Xadj[i]; c < gr.Xadj[i + 1]; c++) {

int p = colors[gr.Xadj[c]];

if (p >= 0)

isused[p] = true;

}

for (int c = 0; c < cnt + 1; c++)

if (!isused[c]) {

for (int z = gr.Xadj[conflv[i]]; z < gr.Xadj[conflv[i] + 1]; z++) {

int cq = colors[gr.Xadj[z]];

if (cq >= 0)

isused[cq] = false;

}

}

colors[conflv[i]] = c;

});

++mconfl;

} while (confc > 0);

delete[] isdet;

delete[] conflv;

parallel\_for("DELIS", N, [=](int i) {

delete[] isused;

});

}