МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Вятский государственный университет
Факультет прикладной математики и телекоммуникаций
Кафедра прикладной математики и информатики

Допускаю к защите
Заведующий кафедрой

_____/ Иномистов В.Ю. /
(подпись) (Ф. И. О)

РАЗРАБОТКА БЕНЧМАРКА, ОЦЕНИВАЮЩЕГО ПАРАЛЛЕЛЬНЫЕ СИСТЕМЫ В ОТНОШЕНИИ ЗАДАЧ КЛАССА DATA-INTENSIVE

Пояснительная записка для дипломной работы

ТПЖА.010551.029 ПЗ

Разработал студент гр. ПМ-51 _____/Кислицын И.К. /_____

Руководитель к.т.н., доцент _____/Иномистов В.Ю. /_____

Нормконтролёр ст. преподаватель _____/Фищева И.Н. /_____
(подпись) (Ф. И. О.) (дата)

Киров 2014

Содержание

Введение

Существует большое количество вариантов аппаратной реализации параллельных вычислительных систем, однако они могут иметь разную эффективность с точки зрения скорости решения задач.

Задачи в целом можно разделить на две категории:

- compute-intensive (CI);
- data-intensive (DI).

На протяжении долгого времени суперкомпьютеры использовались в основном для решения задач CI. Этот класс содержит задачи, в которых основная часть времени уходит на вычисления, а хранимые в памяти данные невелики. К задачам такого класса можно отнести моделирование физических процессов, приближённые вычисления, криптографию и криптоанализ [1].

Распараллеливание задач класса CI сводится к разбитию на подзадачи, которые в дальнейшем запускаются на отдельных узлах параллельной системы. Ускорение достигается за счёт того, что несколько вычислительных операций производятся одновременно.

Эффективность аппаратной реализации параллельных систем, ориентированных на решение задач данного класса будет сильно зависеть от вычислительной мощности отдельных узлов, так как чем она выше, тем выше ускорение, полученное от распараллеливания. С другой стороны, качество коммуникационных линий между узлами не столь сильно влияет на эффективность, так как объёмы передаваемых данных малы.

В 1993 году началась разработка архитектуры компьютерных систем, поддерживающей параллелизм по данным, названной MPI (Message Passing Interface). Первая версия стандарта была разработана в 1994 году, однако содержала много неточностей. В 1995 году вышла доработанная и исправленная версия стандарта (1.1) [2], первая реализация которого появилась в 2002 году. На данный момент существует 3.0 версия стандарта (2012 год) [2], которая имеет несколько реализаций.

Благодаря появлению MPI стало возможным быстрое решение другого класса задач. Оно сводится к разделению данных между узлами системы, и параллельной их обработке. Ускорение достигается за счёт того, что несколько единиц данных обрабатываются в один момент времени. Такие задачи изначально назывались Data-intensive system. В дальнейшем они были обобщены до класса DI.

Задача принадлежит классу DI, если при решении необходимо активно использовать систему ввода-вывода, либо объём хранимых данных существенно больше памяти узла, и при этом низки требования к его вычислительной мощности.

Исследование подходов к решению таких задач актуально, что подтверждается [3] финансированием их средствами National Science

Foundation. С 2009 по 2010 годы проходило одно из таких исследований, в котором подверглись анализу:

- подходы к параллельному программированию с точки зрения параллельной обработки данных в задачах класса DI;
- разработка абстракций для программирования, включая модели, языки и алгоритмы, позволяющие в естественном виде описывать параллельную обработку данных;
- проектирование архитектур параллельных систем, решающих задачи класса DI, и обладающих высокими уровнями доступности, эффективности, надёжности и масштабируемости;
- определение приложений, которые смогут использовать данную парадигму программирования, и того, как она должна развиваться, чтобы поддерживать новые приложения.

Для того чтобы определить, насколько хорошо система решает тот или иной класс задач, используются тесты производительности (бенчмарки).

Существует большое количество бенчмарков, оценивающих вычислительную мощность параллельных систем в отношении задач класса Compute-intensive, но наиболее часто для оценки используются различные реализации LINPACK.

Основная идея бенчмарков LINPACK – оценить, насколько быстро параллельная система решает СЛУ порядка *n*. Исходя из скорости решения, выводится итоговая оценка, измеряющаяся в FLOPS (floating-point operations per second, операций с плавающей запятой в секунду).

В 2010 [4] году был впервые представлен бенчмарк, оценивающий параллельные системы в отношении задач класса Data-intensive: Graph500. Со временем была разработана спецификация и несколько последовательных и параллельных его реализаций.

Основная идея данного бенчмарка заключается в решении задачи поиска в ширину (BFS, breath-first search) в большом графе, распределённом по узлам системы. Исходя из размера графа и скорости решения задачи, выводится итоговая оценка, измеряющаяся в TEPS (traversed edges per second, посещённых рёбер в секунду).

Несмотря на то, что реализации данного бенчмарка активно используются для оценки параллельных систем, они не лишены недостатков.

Данная работа посвящена созданию аналога Graph500, не теряющего его достоинств и исправляющего недостатки, так как в силу определённых причин исправление их в Graph500 нецелесообразно.

Было принято решение дать разрабатываемому бенчмарку имя DGraphMark (distributed graph benchmark, бенчмарк на распределённых графах), тем самым не ограничивая круг тестовых задач, применяющихся в нём, а расширяя на все задачи на графах, так как многие их них принадлежат классу data-intensive.

1 Класс задач Data-intensive

Как было сказано во введении, задачи класса Data-intensive оперируют большим объёмом данных, размер которых существенно превышает память современного вычислительного узла (до нескольких петабайт). При этом в таких задачах операций над данными существенно больше, чем вычислительных [5].

1.1 Примеры

Многие задачи на графах, в связи с тривиальностью вычислительных операций, принадлежат классу DI. Примеров много:

- поиск кратчайшего пути между заданными вершинами;
- поиск вхождения графа заданного вида в другой граф;
- задачи BFS (breadth-first search, поиск в ширину) и DFS (depth-first search, поиск в глубину);
- поиск вершин, через которые проходит наибольшее количество кратчайших путей, — задача BWC (Betweenness centrality).

Некоторые задачи по обработке больших данных (Big data) сводятся к задачам на графах, особенно если данные можно представить как граф. К ним относятся, например, анализ данных социальных сетей с целью выявления закономерностей и анализ зависимостей между финансовыми данными, необходимый для предсказания поведения рынка.

Также к классу DI относятся задачи, решаемые архитектурно-программным комплексом MapReduce [6] от Google. Он позволяет в функциональном стиле обрабатывать данные, используя две функции: map и reduce. Функция map принимает входные данные и генерирует множество ключей-значений. Reduce объединяет и обрабатывает эти значения по определённому ключу. В итоге получается набор обработанных данных, который некоторым образом отражает закономерности исходных.

1.2 Особенности реализации

Эффективность решения зависит от программных и аппаратных особенностей.

Программная эффективность зависит от качества алгоритма решения задачи, а также используемых программных инструментов и особенностей самой задачи.

Аппаратная, в отличие от задач класса CI, зависит, прежде всего, от скорости, надёжности, доступности и уровня безотказности линий коммуникаций между узлами.

В связи с этим эффективность распараллеливания задач класса Data-intensive прямо зависит от качества линий коммуникаций параллельной системы, и тем ниже, чем больше информационных обменов применено в алгоритмах их решения.

Повысить эффективность решения задачи, таким образом, можно, либо уменьшив информационные обмены в алгоритмах решения, либо увеличив количество узлов, на которых распараллеливается задача.

Однако оба пути имеют ограничения в связи с тем, что между данными в задаче могут существовать зависимости.

1.3 Влияние зависимостей в данных на эффективность решения

Выделяют два вида зависимостей [7]:

- прямая (узлу требуются исходные данные другого узла);
- условная (узлу требуются данные, вычисленные определённым узлом на определённом этапе).

В некоторых задачах можно распределить данные таким образом, чтобы представленных зависимостей не было. В связи с тем, что в решении таких задач не нужно учитывать зависимости между данными, может быть достигнута высокая эффективность параллельного решения. Однако чаще встречаются задачи, в которых зависимости присутствуют.

От прямых зависимостей можно избавиться, введя избыточность хранимых данных. Однако введённая избыточность не должна быть большой, так как в противном случае снизится эффективность распараллеливания вследствие того, что объём исходных данных, переданных узлу, может уменьшиться в силу ограниченности памяти узла.

Вопрос об эффективном доступе к начальным данным может быть решён также и на аппаратном уровне: системы с общей памятью лишены данного недостатка. Также можно модифицировать исходную аппаратную архитектуру таким образом, чтобы линии коммуникаций, идущие от начального узла, имели наибольшую пропускную способность, и подключались к как можно большему (в условиях выбранной архитектуры) количеству узлов.

Обмены данными, вызванные наличием условных зависимостей, невозможно исключить из решения задачи. В некоторых частных случаях узел может заранее отправить необходимые условные данные другому узлу (например, оповещение о завершении работы или о начале доступности).

В более общих ситуациях узел не знает, какие данные и какому узлу он должен будет предоставить, что приводит к необходимости выполнения динамической синхронизации между узлами.

Также условные зависимости снижают эффективность масштабирования задач: чем больше количество узлов, тем больше условными обменов данными между ними.

Снизить влияние таких зависимостей на программном уровне можно, переорганизовав порядок вычислений. Например, в период ожидания доставки, можно выполнять действия локального характера, не требующие дополнительных внешних данных, либо предоставлять другим узлам необходимые им условные данные.

Действенность данных методов зависит от силы условной зависимости между задачами и аппаратной реализации параллельной системы.

В случае высокого уровня условных зависимостей, выполнение действий локального характера может не принести никакого эффекта, так как основное время будет расходоваться на обмен данными между узлами. Связано это с тем, что скорость обработки локальных данных существенно выше скорости взаимодействий в параллельной системе (последняя определяется качеством линий коммуникаций).

Выполнение же динамической синхронизации в период ожидания ответа, однозначно повышает эффективность решения задачи. Однако при плохом качестве линий коммуникаций, повышается время передачи данных между узлами, что приводит к снижению эффективности решения задачи, так как существенное время уйдёт на обмен, а не на обработку данных.

Видно, что эффективность решения задач, имеющих условные зависимости между данными, существенно зависит от качества реализации аппаратной части параллельной системы.

Для того, чтобы реализовать бенчмарк DGraphMark, необходимо провести подробный анализ аналогов, выявить их достоинства и недостатки. Рассмотренные в данном разделе особенности реализации задач класса data-intensive могут быть использованы, как при анализе аналогов, так и при построении алгоритмов DGraphMark. Разнообразие задач, описанное в подразделе 1.1 должно быть учтено при проектировании для возможности расширения бенчмарка на любую из представленных задач.

2 Обзор бенчмарка Graph500

Производители зациклены на адаптацию параллельных систем под тесты LINPACK. Однако наравне с FLOPS есть и другие параметры для оценки параллельных систем, и при наращивании одной без существенного улучшения других не будет достигнуто существенное ускорение в реальных задачах.

Данный факт не является неожиданным: уже в 2007 году было выпущено большое количество статей, отражающих актуальность данного вопроса [8].

Разработка Graph500 является ожидаемым ответом сообщества на данную проблему. Согласно спецификации [4], главная цель этого бенчмарка – поспособствовать развитию суперкомпьютерных технологий посредством создания новой оценки для параллельных систем – TEPS (traversed edges per second, посещённых рёбер в секунду).

Основная суть бенчмарка заключается в создании и распределении большого графа (Кронекеровского типа), и дальнейшем запуске на нём задачи класса DI. В качестве такой задачи выбран BFS.

В спецификации предусмотрено шесть размеров задач:

- toy (17 ГБ, около 10^{10} Б, уровень 10);
- mini (140 ГБ, около 10^{11} Б, уровень 11);
- small (1 ТБ, около 10^{12} Б, уровень 12);
- medium (17 ТБ, около 10^{13} Б, уровень 13);
- large (140 ТБ, около 10^{14} Б, уровень 14);
- huge (1.1 ПБ, около 10^{15} Б, уровень 15).

2.1 Функциональное описание

Функционально бенчмарк состоит из пяти этапов:

а) считывание начальных данных;

б) генерация случайного графа;

в) выбор 64 случайных вершин;

г) выполнение функций ядра бенчмарка;

д) анализ и вывод результатов.

2.1.1 Начальные данные

Начальные данные представляют собой параметры для генерации графа: *scale* и *edgefactor*.

Граф G представляет собой совокупность вершин V и рёбер E

$$G = (V, E). \qquad (2.1)$$

Параметр *scale* задаёт количество вершин графа

$$|V| = 2^{scale}. \quad (2.2)$$

Параметр *edgefactor* отвечает за среднее количество рёбер, смежных с каждой вершиной

$$\forall v \in \mathrm{V}\ \deg(v)\ \approx edgefactor, \quad (2.3)$$
$$|E| = |V|\, edgefactor. \quad (2.4)$$

Значения по умолчанию для входных параметров

$$scale = 8,$$
$$edgefactor = 16. \quad (2.5)$$

2.1.2 Генерация графа

На данном этапе генерируется случайный граф Кронекеровского типа с помощью алгоритма R-MAT [9]. Данный граф хорошо моделирует связи в Интернете и социальных сетях [10].

Правила генерации просты:

а) выбираются начальные диапазоны данных для источника и назначения ребра (от нуля до максимального индекса вершины);

б) диапазоны данных делятся пополам. В соответствии с некоторый вероятностью (таблица 2.1) решается, какую половину считать следующим диапазоном;

в) действия повторяются, пока не диапазоны не укажут на единственные вершины источника и назначения;

г) в список рёбер добавляется полученное ребро.

Таблица 2.1 – Вероятности выбора диапазонов

		Выбор диапазона вершины назначения	
		левый	правый
Выбор диапазона вершины источника	левый	$\alpha = 0.57$	$\beta = 0.19$
	правый	$\gamma = 0.19$	$\delta = 0.05$

Визуализация алгоритма на матрице смежности продемонстрирована на рисунке 2.1.

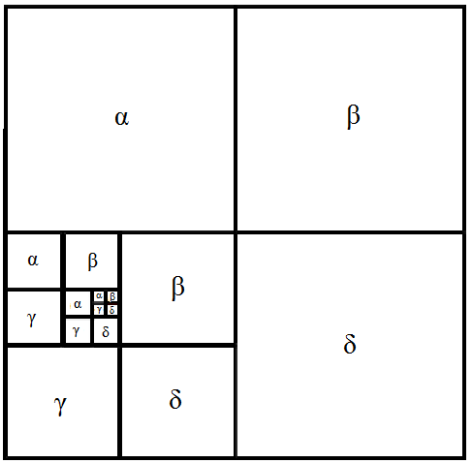

Рисунок 2.1 – Визуализация алгоритма R-MAT

Необходимо отметить, что в реализации Graph500 не происходит деориентации графа, то есть в результате генерации получается ориентированный граф. При деориентации все рёбра дублируются и переворачиваются, что делает их неориентированными.

2.1.3 Функции ядра

После того, как создан граф и выбрано 64 начальные вершины для запуска BFS, происходит вызов функций ядра.

Первая функция ядра, выполняющаяся один раз – graph construction, создание графа, идентичного построенному на втором шаге, но представленного в специальном виде.

Граф приводится к виду CSR (Compressed Sparse Row, сжатый по строкам). При этом все его рёбра сортируются по возрастанию, источника. Также строится массив быстрого доступа к началу и концу участка рёбер, исходящих из каждой вершины.

После построения графа начинается основная работа бенчмарка: 64 раза выполняется вторая функция ядра – BFS. На выходе получается массив родителей, представляющий построенное дерево поиска.

Массив родителей – структура данных, содержащая дерево следующим образом: для начальной вершины родитель – она сама, для всех вершин, связанных с ней родитель – начальная. Для остальных вершин родитель – вершина, через которую был найден путь от начальной к данной.

Построенный массив родителей необходимо проверить, так как в ходе работы могут возникнуть непредсказуемые ошибки. Существует несколько проверок:

- массив родителей представляет собой дерево (нет циклов);
- вершина и её родитель соединены ребром в начальном графе;
- ребро соединяет вершины, глубина которых отличается на единицу.

2.1.4 Вывод результатов

Некоторые этапы бенчмарка имеют оценку времени:

- генерация графа;
- построение графа;
- BFS (все 64 раза);
- валидация результата (все 64 раза).

После окончания работы выводится результата в виде «ключ: значение», и содержащий следующие параметры:

- начальные параметры;
- количество узлов в параллельной системе;
- количество запусков;
- время генерации графа;
- время построения графа;
- статистика для времени BFS;
- статистика для времени валидации;
- статистика для оценок бенчмарка.

Оценка бенчмарка вычисляется как

$$mark = \frac{|E_{traversed}|}{time_{BFS}}, \quad (2.6)$$

где $E_{traversed}$ – множество посещённых в BFS рёбер,
$time_{BFS}$ – время BFS,
$mark$ – оценка бенчмарка.

В случае связности графа каждое ребро рассматривается только один раз и выражение для вычисления оценки можно упростить

$$|E_{traversed}| = |E| = edgefactor\ 2^{scale}, \quad (2.7)$$

$$mark = \frac{edgefactor\ 2^{scale}}{time_{BFS}}. \quad (2.8)$$

Статистическая обработка массива 64 значений приводит к формированию следующих данных:

- математическое ожидание;
- стандартное отклонение (оценка несмещённой дисперсии);
- минимум;
- первый квантиль ($\chi_{1/4}$);
- медиана ($\chi_{1/2}$);
- третий квантиль ($\chi_{3/4}$);
- максимум.

Несмотря на то, что спецификация хорошо составлена, сказать такого же о реализации нельзя. В следующем подразделе будут разобраны достоинства и недостатки текущих реализаций.

2.2 Анализ реализаций

На данный момент имеется несколько реализаций для спецификации Graph500 [1]:

- последовательная на GNU Octave;
- две версии для Cray-XMT;
- последовательная на C;
- параллельная на C (OpenMP);
- параллельная на C (MPI P2P, point to point);
- параллельная на C (MPI RMA, remote memory access).

Имеет смысл рассматривать только реализации, основанные на MPI, так как только с их помощью возможно оценить производительность параллельной системы с разделёнными узлами. Связано это с тем, что задачи класса DI обрабатывают большой объём данных, и он не может храниться в памяти одного узла.

В спецификации [4] указано несколько критериев для оценки качества реализации.

В порядке убывания важности:

- полное следование спецификации;
- возможность запустить задачу наибольшего размера для предложенной машины;
- минимальное время выполнения задач;
- минимальный размер кода;
- минимальное время разработки;
- максимальная ремонтопригодность;
- максимальная расширяемость.

При анализе исходного кода реализаций, основанных на MPI, было обнаружено существенное количество недостатков, многие из которых идут в разрез с представленными критериями качества.

2.2.1 Неоптимальность работы с памятью

На рисунках 2.2 и 2.3 показано распределение памяти при запуске бенчмарка (P2P) на двух и восьми узлах при одинаковом размере $scale = 16$.

Видна и очевидна неравномерность распределения данных, большое количество избыточных, а также на рисунке 2.2 видна неравномерность загрузки вычислительных ядер.

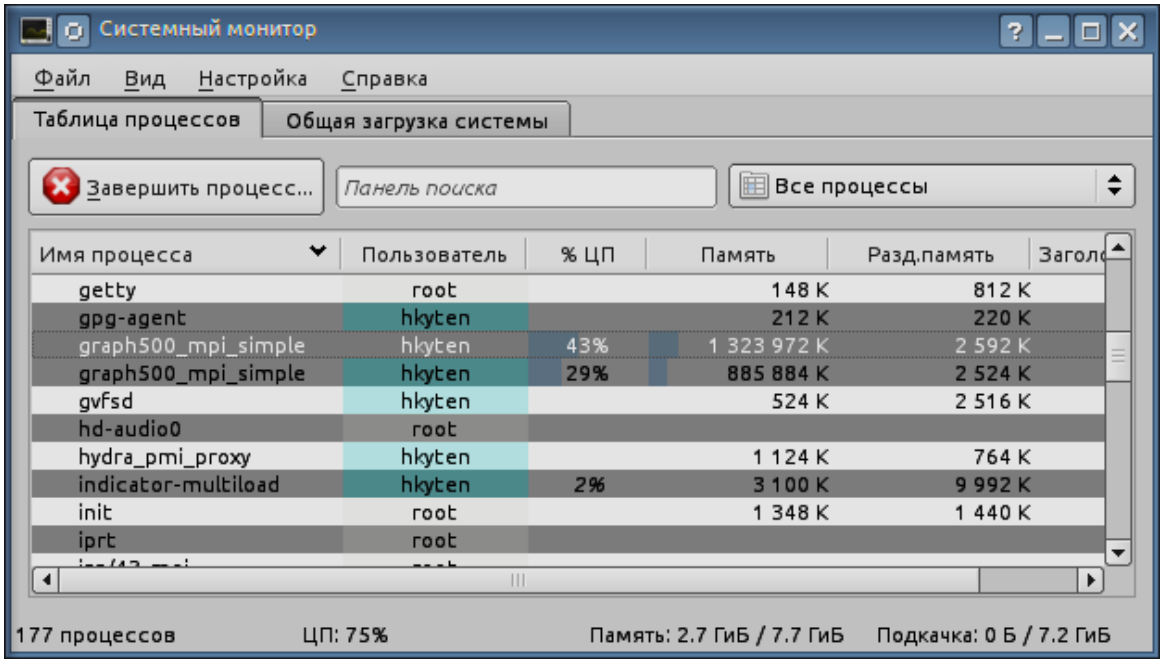

Рисунок 2.2 – Монитор ресурсов при запуске graph500_mpi_simple на двух узлах при размере задачи 16

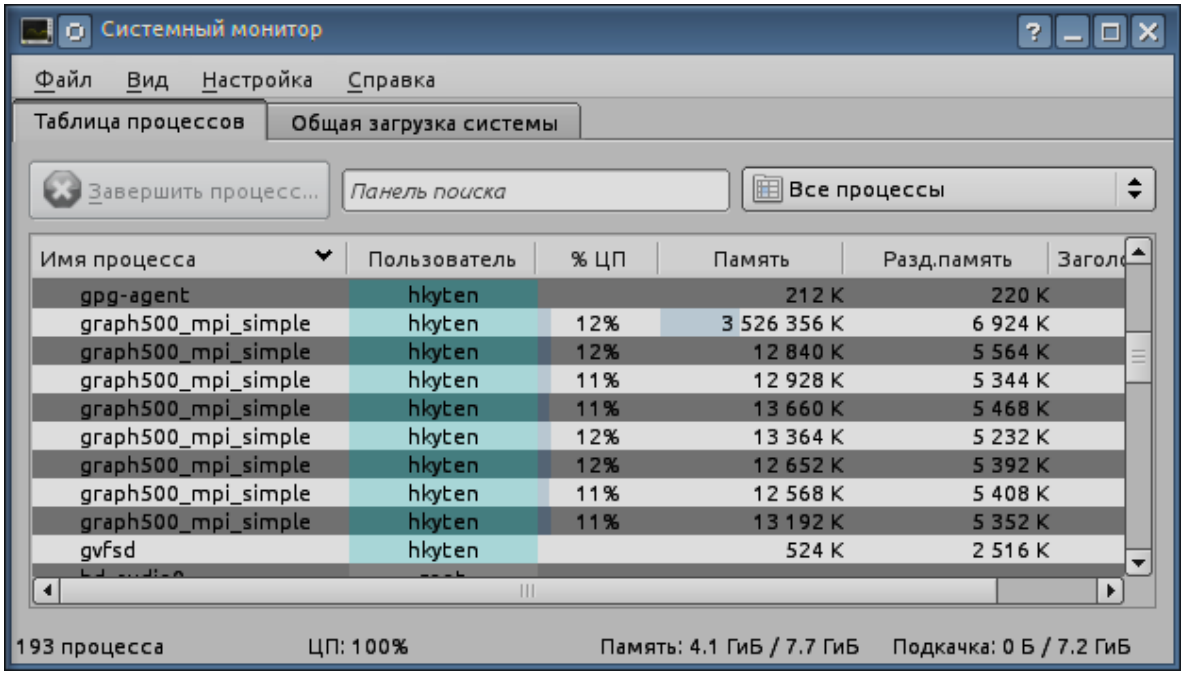

Рисунок 2.2 – Монитор ресурсов при запуске graph500_mpi_simple на восьми узлах при размере задачи 16

Необходимо оценить размер занимаемой памяти при решении задачи BFS реализацией Graph500. Для этого необходимо рассмотреть используемые структуры данных.

Граф хранится как список из $|E|$ рёбер. Каждое представляет собой два 64-битных беззнаковых числа. Итоговый размер графа (для стандартного *edgefactor*) выражается как

$$size(G) = |V|\ edgefactor\ 2\ size(uint64_t), \quad (2.9)$$

$$size(G) = 2^{scale} \cdot\ 16 \cdot\ 2 \cdot\ 64\ bit = 2^{scale+11} bit = 2^{scale+8}\ byte.$$

Для представления графа в формате CSR требуется дополнительный массив CSR, равный по размеру количеству вершин

$$size(CSR) = |V|\ size(uint64_t), \quad (2.10)$$

$$size(CSR) =\ 2^{scale}\ 64\ bit = 2^{scale+3}\ byte.$$

Для представления массива родителей требуется такой же объём данных, как и для массива CSR. Также в BFS используется две очереди такого же размера

$$size(BFS) = 3\ size(CSR). \quad (2.11)$$

При валидации результата на каждом узле создаётся четыре массива, количество элементов в которых равно 2^{21}. Объём данных, занимаемый при валидации, составляет при этом

$$size(validation) = size \cdot\ 4 \cdot\ 2^{21} \cdot\ sizeof(uint64_t)\ , \quad (2.12)$$

где size – количество узлов в параллельной системе.

$$size(validation) = size\ \cdot 2^{29}\ bit = size \cdot 2^{26}\ byte,$$

В итоге при выполнении должна выделяться память, не сильно превышающая величину size(data), вычисляемую как

$$size(data) = size(G) + size(CSR) + size(BFS) + size(validation) \quad (2.13)$$

$$size(data) = size(G) +\ size(CSR) + 3\ size(CSR) + size(validation) =$$
$$= size(G) + 4\ size(CSR) + size(validation)$$
$$= 2^{scale+8} + 4\ \cdot 2^{scale+3} + size \cdot 2^{26} =$$
$$= 2^{scale+5} \cdot (2^3 + 1) + size \cdot 2^{26} == 9 \cdot 2^{scale+5} + size \cdot 2^{26}\ byte$$

Таким образом, для первого случая (два узла, размер задачи 16) размер данных составит около

$$size(data) = 9 \cdot 2^{16+5} + 2 \cdot 2^{26} = \\ = 2^{21} \cdot (9 + 2^6) \approx 1.3 \cdot 2^{27} \approx 134 \text{ МБ}$$

Данные, приведённые на рисунке 2.2, показывают, что реально выделяемый размер памяти существенно выше расчётного.

Для второго случая (восемь узлов и размер задачи 16) размер данных составит около

$$size(data) = 9 \cdot 2^{16+5} + 8 \cdot 2^{26} = \\ = 2^{21} \cdot (9 + 2^{11}) \approx 2^{29} = 536 \text{ МБ}$$

Выходит, что и для второго случая ситуация аналогична. Из этого можно сделать вывод, что присутствует высокая неоптимальность работы с памятью, вызванная либо её утечками, либо расходом её на дополнительные не необходимые структуры данных. Данный факт является следствием нарушение второго критерия качества, установленного спецификацией.

2.2.2 Неоптимальность работы с OpenMP

В коде валидатора повсеместно встречаются OpenMP инструкции для распараллеливания циклов. При компиляции бенчмарка без ключа –fopenmp (с выключенной поддержкой такого рода инструкций) он начинает работать быстрее: в 50-100 раз меньше времени на валидацию на малых размерах задач, 20% выигрыш в производительности на больших (22) размерах.

Объясняется данный факт тем, что при использовании инструкций не учитывалось то, что циклы, к которым они применены, имеют зависимости по данным и не могут быть распараллелены. В малых задачах, к тому же, добавляются расходы на частое создание/удаление потоков, выполняющих незначительные вычисления.

Пример. Проверяется корректность значений массива предков *pred* (должны лежать от нуля до индекса максимальной вершины).

```
int any_range_errors = 0;
#pragma omp parallel for reduction(||:any_range_errors)
for (i = i_start; i < i_end; ++i) {
    int64_t p = get_pred_from_pred_entry(pred[i]);
    if (p < -1 || p >= nglobalverts) {
        fprintf(stderr, "%d: Validation error: parent of vertex
%" PRId64 " is out-of-range value %" PRId64 ".\n", rank,
vertex_to_global_for_pred(rank, i), p);
        any_range_errors = 1;
    }
}
```

Неоптимален цикл по двум причинам:

- внутри цикла используется функция вывода в консоль данных. Из-за неё добавляется большое количество разделённых (shared) переменных к потокам, что замедляет их создание и работу (они вызываются функцией fprintf, и для каждого потока создаётся отдельная копия);
- используется условный оператор, которого можно было бы избежать.

Оптимизация:

```
int any_range_errors = 0;
#pragma omp parallel for reduction(||:any_range_errors)
for (i = i_start; i < i_end; ++i) {
    int64_t p = get_pred_from_pred_entry(pred[i]);
    any_range_errors |= p < -1 || p >= nglobalverts;
}
if(any_range_errors) {
    fprintf(stderr, "%d: Validation error: parent of vertex
            is out of range ", rank);
}
```

Вывод информации об ошибке можно и нужно вынести за пределы цикла, так как информация о номере вершины не имеет смысла, так как граф большой и случайный. Отследить причину данной ошибки по указанной информации невозможно.

Таким образом, нарушен третий критерий качества реализации: о наибольшей скорости выполнения задачи.

2.2.3 Отсутствие стандартов кодирования

Отсутствие стандартов кодирования при разработке больших проектов приводит к высокой стоимости исправления ошибок, большого порога вхождения в проект, и в целом усложняет его [11].

Код проекта Graph500 нарушает многие повсеместно используемые стандарты качества ПО [11]:

- неразумно выбранные имена переменных, структур данных и методов (не отображают сущности хранимых данных);
- путаница в типах (size_t, int64_t, uint64_t, ptrdiff, др), использующихся для хранения одних и тех же данных;
- недостаточное проектирование, что приводит к невозможности расширить структуру кода без существенных затрат времени;
- повсеместное использование больших методов, которые выполняют больше одной задачи, что увеличивает шанс ошибок и среднее время их исправления [11];
- код пронизан макросами, избавление от которых в короткие сроки не всегда возможно в силу ошибок проектирования.

Эти и многие другие проблемы в организации, стиле и проектировании кода приводят к нарушению оставшихся критериев качества.

Была произведена попытка исправления указанных ошибок, но оказалась безуспешна, в связи с их количеством. При рефакторинге метода RMA-версии BFS было обнаружено несколько ошибок в работе с памятью, исправление которых несколько повысило производительность. Однако было отмечено, что исправление ошибок проектирования и выбора инструмента (язык C) невозможны. Связано это с тем, что они приведут к постепенному полному переписыванию бенчмарка, что рациональнее сделать с нуля.

Язык C++ предоставляет больше гибкости при разработке, и при этом в ходе анализа было замечено, что переход на компилятор g++ вместо gcc не привёл к изменению результатов, в связи с чем можно сделать вывод: C++ как инструмент для разработки лучше подходит для данной задачи, так как несёт с собой при грамотном исполнении повышение производительности за счёт более прозрачной работы с ресурсами.

К тому же, существенным недостатком данного бенчмарка является то, что невозможно без существенных издержек ввести поддержку другого класса задач, отличного от поискового алгоритма. Этот недостаток также опирается на модель представления проекта в C, и легко решается с переходом на C++, при перепроектировании.

В данном разделе был проанализирован бенчмарк Graph500. Были рассмотрены основные его алгоритмы, достоинства и недостатки. Основываясь на проведённом анализе, необходимо провести проектирование бенчмарка DGraphMark, в результате которого должны быть решены архитектурные ошибки Graph500, а также заложена возможность масштабирования на другие задачи класса data-intensive и другие алгоритмы.

3 Проектирование бенчмарка DGraphMark

В разделе 2 были рассмотрены основные достоинства и недостатки Graph500, а также описана функциональная сторона бенчмарка. На рисунке 3.1 она изображена в виде диаграммы деятельности UML.

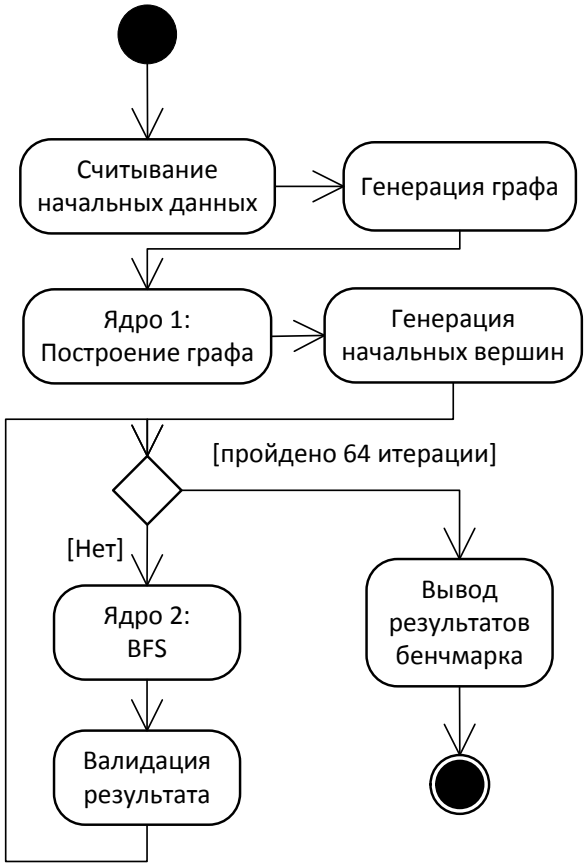

Рисунок 3.1 – Диаграмма деятельности Graph500

Видно, что присутствует жёсткая привязка к BFS и вообще поисковой задаче. Для того, чтобы программа была более хорошо масштабируема и адаптируема к другим типам задач на графах (подраздел 1.1), необходимо ввести несколько обобщений.

В спецификации Graph500 упоминается сущность Controller. Её деятельность совпадает с деятельностью Graph500. Задача её – правильным образом вызывать функциональные этапы и хранить результаты между вызовами.

Для того, чтобы была возможность абстрагироваться от конкретной задачи, необходимо ввести сущность Benchmark, которая будет содержать в себе детали реализации деятельности бенчмарка.

Контроллер при этом должен выполнять следующие задачи: считывать начальные данные, запускать бенчмарк и формировать результат работы. Диаграмма деятельности бенчмарка в таком случае обобщается не более, чем необходимо (рисунки 3.2 и 3.3).

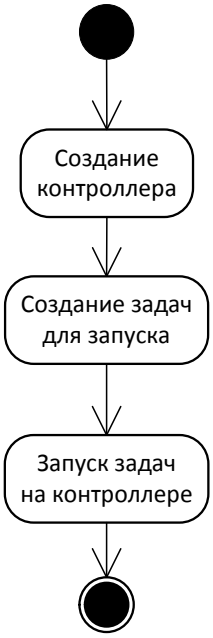

Рисунок 3.2 – Диаграмма деятельности DGraphMark

Рисунок 3.3 – Диаграмма деятельности запуска задач на контроллере

Необходимо также спроектировать сущности графа, ребра, вершины, генератора графов, задачи и валидатора, так как они неявно присутствуют в описанных выше схемах.

Также необходимо учесть существующий в Graph500 набор статистических данных, и по возможности реализовать в проектировании возможность их получения.

3.1 Проектирование сущности графа

На рисунке 3.4 изображены диаграммы классов, определяющие сущность графа.

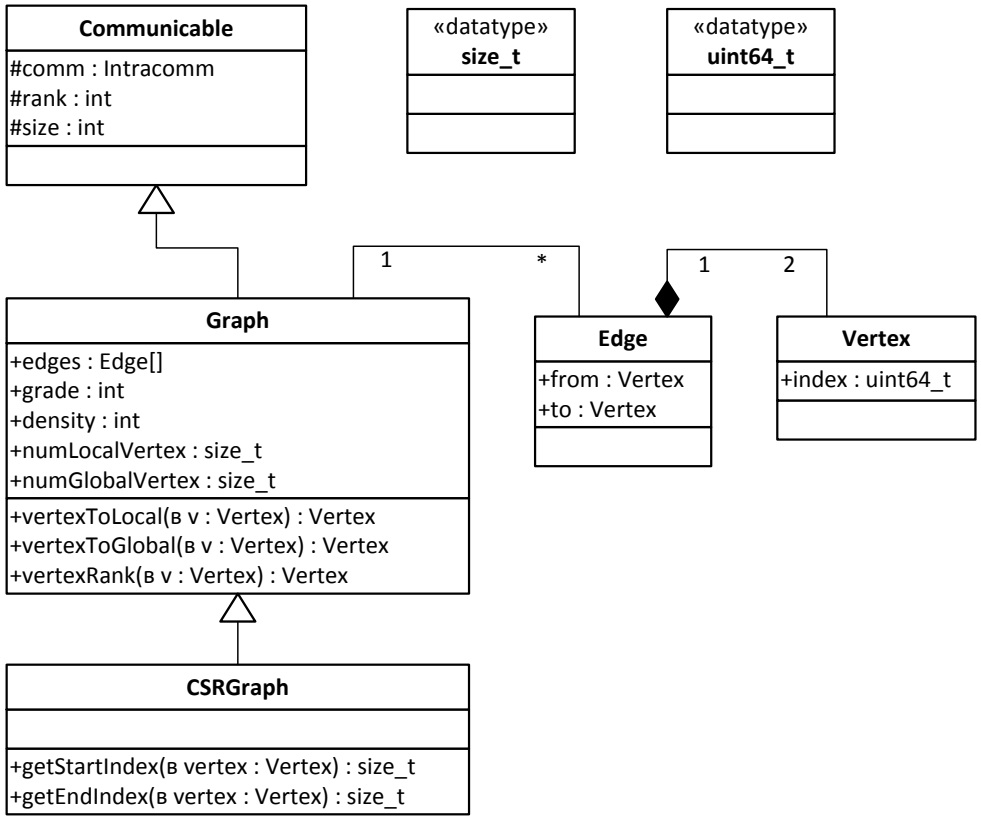

Рисунок 3.4 – Диаграммы классов для сущности графа

Большинство спроектированных сущностей, как в данном, так и в следующих подразделах наследуются от абстрактного класса Communicable. Он предоставляет доступ к основным параметрам MPI, включая индекс текущего узла (rank), количество узлов (size) и объект для выполнения операций внутри коммуникатора. Для экономии пространства и концентрации на модели, в дальнейших схемах он не будет отображён.

Для представления вершины графа необходимо использовать тип данных uint64_t, одинаково реализованный во всех компиляторах.

Введение единого типа данных позволяет решить проблему путаницы типов, возникшую в Graph500. Таким образом, граф может состоять не более чем из 2^{64} вершин.

Также необходимо, чтобы количество узлов параллельной системы было равным степени двойки. Введение данного ограничения позволяет существенно упростить балансировку данных между узлами.

В формулах с 3.1 по 3.5 выражены основные числовые выражения, характеризующие модель графа:

$$numVert = 2^{grade}, \quad (3.1)$$

$$commGrade = \log_2^{size}, \quad (3.2)$$

$$diffGrade = grade - commGrade, \quad (3.3)$$

$$numVertLocal = 2^{diffGrade}, \quad (3.4)$$

$$numVert = 2^{commGrade + diffGrade} = size \cdot numVertLocal, \quad (3.5)$$

где *grade* – размер графа, сложность задачи,

size – количество узлов,

numVert – общее количество вершин (глобальные вершины),

numVertLocal – количество вершин, привязанных к каждому узлу (локальные вершины),

diffGrade – двоичный логарифм локального количества вершин,

commGrade – двоичный логарифм количества узлов.

Формула 3.5 показывает, что индекс вершины можно разделить на две части: первые *commGrade* бит отвечают за номер узла, а оставшиеся за локальный индекс вершины. В таком случае, вершины распределяются равномерно по всем узлам, так как каждому достаются *numVertLocal* вершин.

На рисунке 3.5 наглядно показано, как связаны основные параметры вершины.

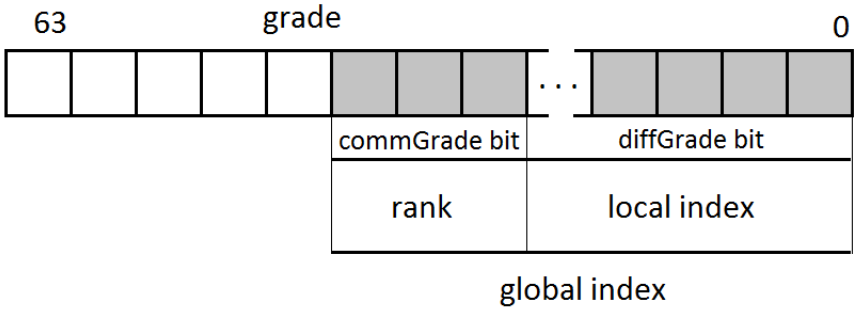

Рисунок 3.5 – Распределение данных в глобальном индексе вершины

Ребро представляет собой упорядоченный набор из двух вершин. Наименование у них следующее: *from* и *to*: источник и назначение вершины.

Граф представляет собой набор рёбер и нескольких важных параметров: размер (*grade*), плотность (*density*, среднее количество рёбер, связанных с вершиной), глобальное и локальное количество вершин.

Граф должен создаваться таким образом, чтобы часть *rank*, из каждой вершины *from* в наборе рёбер была равна номеру текущего узла в параллельной системе. То есть локальная версия графа, которая хранится на одном узле, должна содержать только те рёбра, которые из него исходят.

В связи с тем, что рёбра в модели графа ориентированы, сам граф тоже получается ориентированным. Для того, чтобы сделать граф неориентированным, необходимо дублировать все рёбра, изменив направление, и переслать в соответствующие узлы.

3.2 Проектирование генератора графов

На рисунке 3.6 изображена модель генератора графа.

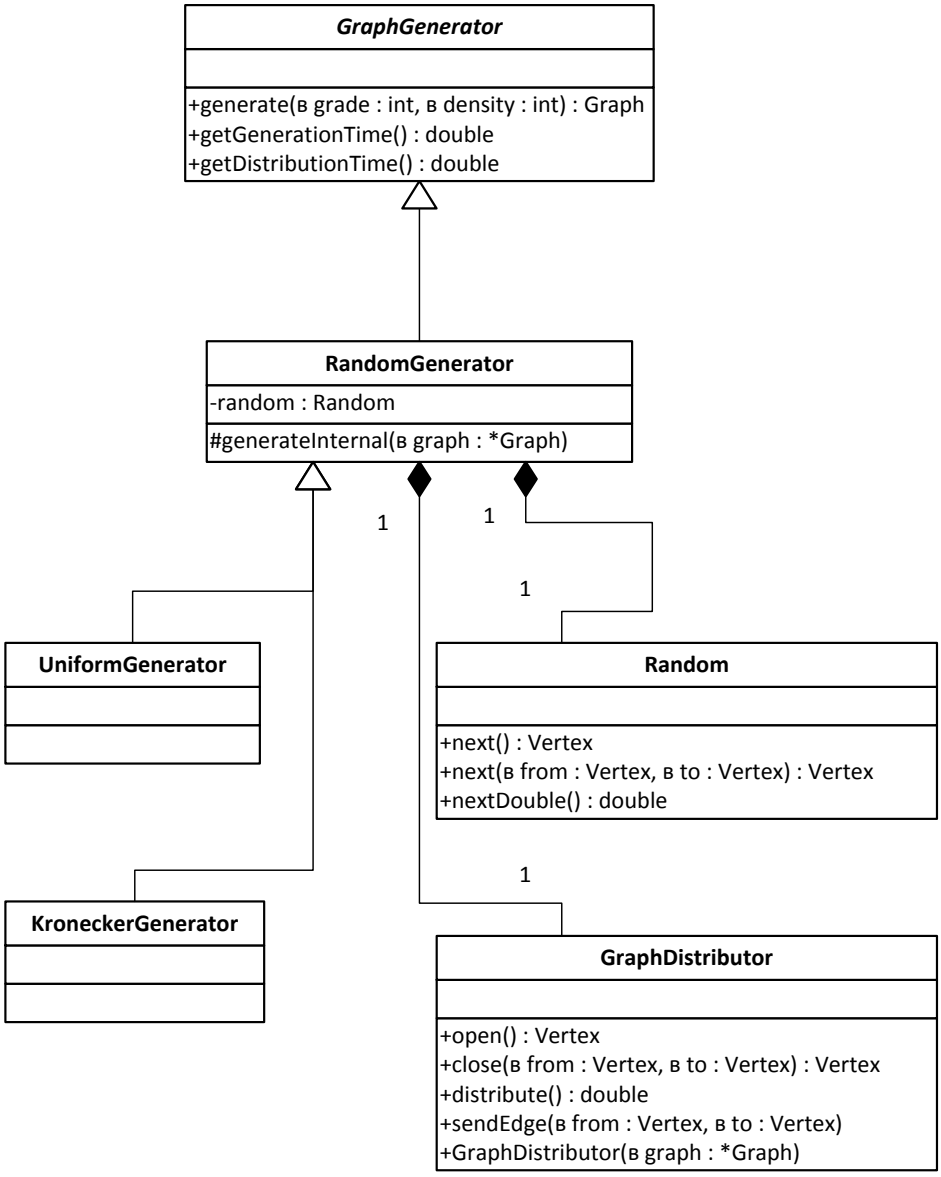

Рисунок 3.6 – Диаграммы классов модели генератора графа

Основная задача генератора – создание графа, второстепенная – его деориентирование. Также для ведения статистики интерфейс должен предоставлять доступ к длительности данных процессов.

Генератор графов (GraphGenerator) – интерфейс, описывающий поставленные задачи.

На модели представлена необходимая с точки зрения поставленной задачи абстракция – генератор случайных графов (RandomGenerator). В нём должен быть описан общий процесс генерации графа и его деориентации, а частные алгоритмы наполнения графа рёбрами должны реализовываться наследниками в методе generateInternal.

Предусмотрено два класса-обёртки, реализующих алгоритмы генерации: равномерной (UniformGenerator) и Кронекера (KroneckerGenerator). Подробнее алгоритмы рассмотрены в разделе 4.

3.3 Проектирование задач

На рисунке 3.7 представлена модель задач.

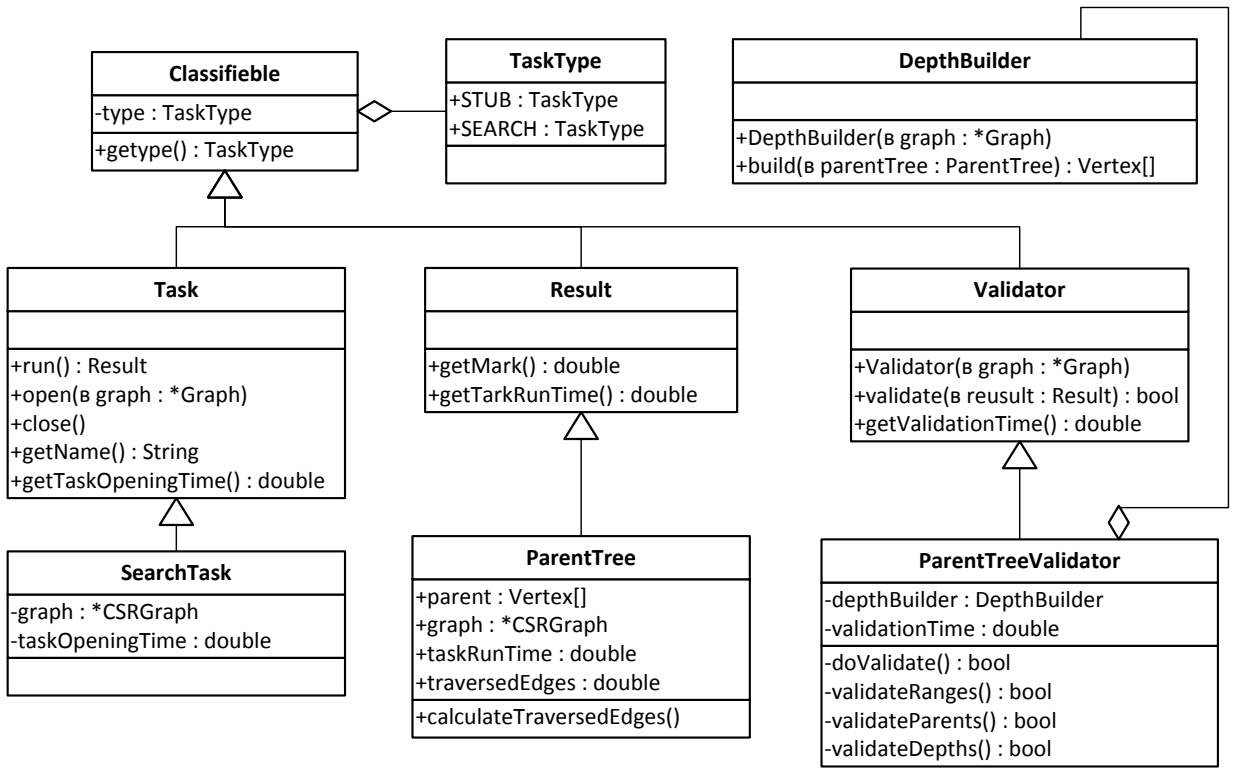

Рисунок 3.7 – Диаграммы классов для сущностей задания, результата и валидатора

Для представления задач, которые должны запускаться бенчмарками, используется интерфейс Task. Он предоставляет доступ к методам открытия, закрытия и запуска задачи, а также даёт доступ к необходимым статистическим данным. Предполагается, что входные данные для задачи будут передаваться либо через конструктор, либо через set-методы в реализациях. Для упрощения идентификации в итоговой статистике каждая задача должна иметь уникальное имя, задающееся в методе getName.

При открытии задачи выделяется память на необходимые параметры, а также происходит приведение графа к определённому виду, соответствующему требованиями задачи. В случае с задачей поиска (SearchTask), при инициализации должна выделяться память под очереди, а граф должен быть приведён к виду CSR.

Результатом решения задачи является объект, наследующий интерфейс Result. Он предоставляет доступ к методам получения времени работы задачи, а также оценки, вычисленной по результатам решения. Кроме того, в результате могут содержаться данные задачи, которые облегчат валидацию.

Результат решения задачи поиска – дерево предков (ParentTree). Оно должно хранить ссылку на граф, в котором было построено, а также должно предоставлять доступ к массиву родителей. Значение массива предков по индексу равному номеру локальной вершины должно быть равно глобальному идентификатор вершины, из которой в процессе решения задачи поиска был осуществлён переход в данную вершину. Если вершина не была посещена, то по соответствующему индексу должно стоять значение *numGlobalVertex* для текущего графа.

Валидатор (Validator) – интерфейс, позволяющий проверить результат решения на корректность. В зависимости от типа задачи могут быть использованы разные методы проверки. Также интерфейс предоставляет доступ к методу получения времени проверки, необходимому в статистике.

В случае валидаторе дерева предков (ParentTreeValidator) должны использоваться все проверки, использующиеся в Graph500 (пункт 2.1.3), что предусмотрено моделью.

Для того чтобы была возможность отличить задачу одного типа от другой, было принято решение унаследовать все перечисленные интерфейсы от Classifieble. Данный интерфейс позволяет определить, к какому типу (TaskType) относится задача, валидатор, и результат. При этом становится возможным быстро отличать реализации данных классов, и давать сбой при некорректном использовании.

К примеру, если в валидатор поступает результат от решения задачи недопустимого типа, можно дать сбой с корректной ошибкой.

3.4 Проектирование бенчмарков и контроллеров

3.4.1 Проектирование бенчмарков

Бенчмарк – сущность, отвечающая за запуск задач на заданном графе определённое количество раз. Также для того, чтобы осуществить требования по сбору статистических данных, необходимо предусмотреть метод для предоставления статистики. Сбор данных должен происходить именно в бенчмарке в связи с тем, что только он контактирует с задачей, валидатором и решением задачи одновременно.

В связи с тем, что разные типы задач имеют различные начальные данные, было принято решение разделить сущность бенчмарка на две: базовую, общую для всех, и специфичную, реализующую запуск задачи.

На рисунке 3.8 представлена спроектированная иерархия классов бенчмарка.

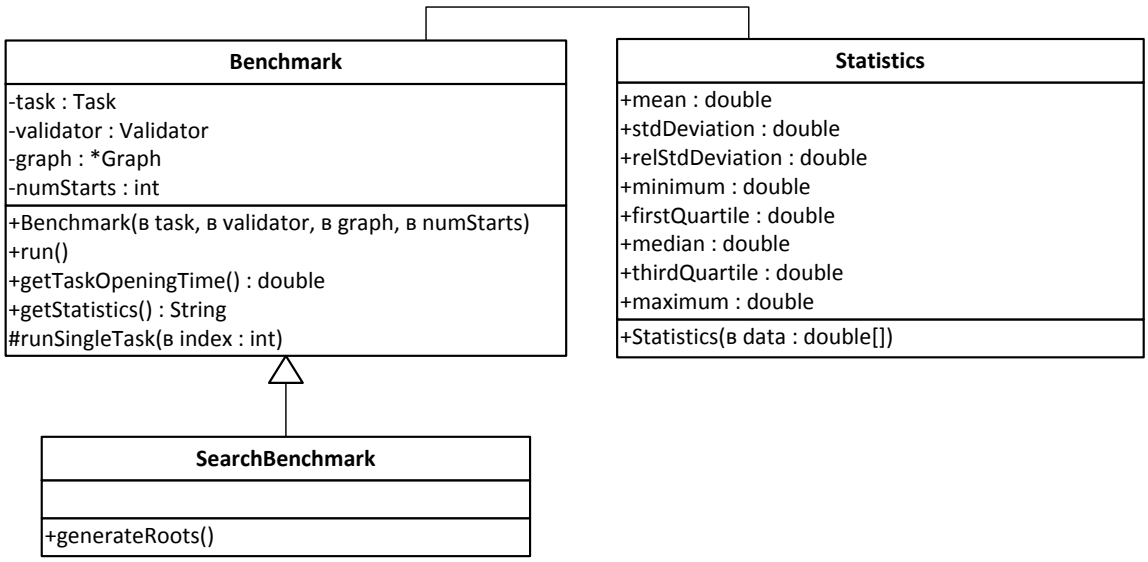

Рисунок 3.8 – Диаграмма классов для сущностей бенчмарка

Сущность бенчмарка представляет абстрактный класс Benchmark. Запуск бенчмарка (метод run) должен приводить к запуску задач numStarts раз через метод runSingleTask, реализованный у наследников. Одновременно с этим наследники должны заполнять статистические данные о времени выполнения задачи, времени валидации и оценке.

Статистика бенчмарка представляет собой анализ статистических данных, полученных в ходе работы. Для каждого из набора данных должны формироваться такие же параметры, как в статистике Graph500 (пункт 2.2.1).

Данные параметры необходимы для поддержания совместимости со статистическими параметрами Graph500, для ведения сравнения.

3.4.2 Проектирование контроллеров

Контроллер – сущность, отвечающая за считывание и распределение начальных данных, а также выполняющая запуск бенчмарков и задач.

По аналогии с бенчмарком, контроллер должен быть разделён на базовую и специфичную часть. Базовая часть должна отвечать за считывание начальных данных (*grade*, *density* и *numStarts*), а также за запуск бенчмарков. Специфичная часть должна отвечать за запуск отдельных задач (формировать соответствующий бенчмарк, и передавать его на выполнение базовой части).

Иерархия классов контроллера представлена на рисунке 3.9.

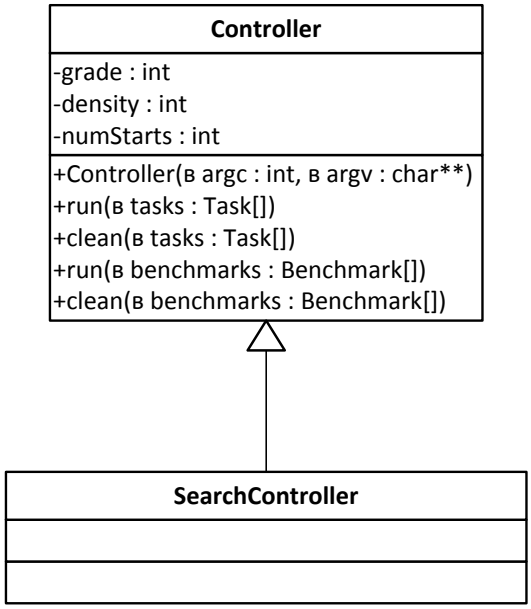

Рисунок 3.9 – Иерархия классов контроллера

Сущность контроллера представляется абстрактным классом Controller. В нём должны быть описаны методы запуска и очистки массива бенчмарков. Очистка нужна для того, чтобы после использования можно было освободить память, занимаемую массивом бенчмарков, проведя соответствующим образом деинициализацию.

Специфичная для задачи поиска сущность поискового контроллера должна реализовать аналогичные методы для массива задач. Так как при создании бенчмарка из задачи необходим граф, то в данном методе он должен создаваться. В связи с этим поисковый контроллер должен содержать генератор графов.

В данном разделе было проведено проектирование бенчмарка DGraphMark. На основе него необходимо сформировать реализацию.

4 Реализация бенчмарка DGraphMark

Прежде чем приступить к реализации, необходимо ввести ряд ограничений и требований к аппаратному и программному обеспечению, а также предусмотреть параметры для оценки качества реализации.

Также необходимо рассмотреть организацию проекта и используемые при сборке средства.

4.1 Общие требования к реализации

В связи с тем, что Graph500 ориентирован на запуск в UNIX-подобных операционных системах, DGraphMark также должен быть ориентирован на запуск в них. Необходимо установить минимальные ограничения, накладываемые на систему.

4.1.1 Программные требования

- В системе должны быть установлены компиляторы C++, поддерживающие стандарт C++98.
- В системе должны быть установлены компиляторы, реализующие стандарт MPI версии 2.0 и выше.
- В системе должен быть установлен инструмент сборки “make”.

4.1.2 Требования к реализации и критерии оценки качества

Требования к реализации должны содержать и дополнять требования к разработке Graph500.

В обязательном порядке реализация должна:

- реализовывать описанный в разделе 3 проект приложения;
- состоять из хорошо структурированного и документированного кода;
- следовать стандарту кодирования ядра Linux [12];
- вести журнал выполнения, аналогичный Graph500.

Качество реализации генератора графа и подготовительного этапа (первая функция ядра в терминах Graph500) оценивается по скорости выполнения генерации и сравнивается с таковым в Graph500.

Качество реализации BFS и валидатора оценивается по медиане времени выполнения, полученного в результате работы бенчмарка, и сравнивается с таковым в Graph500.

Качество использования памяти оценивается, начиная с размера задач большего 18, и сравнивается с таковым в Graph500.

4.1.3 Требования к лицензированию

Для обеспечения беспрепятственной возможности доработки и коммерческого использования бенчмарка в будущем, разработка должна осуществляться под свободной лицензией Apache версии 2.0. В связи с этим в корневой каталог должен быть добавлен файл с описанием лицензии, а в файлах всех исходных кодов должен присутствовать оговорённый лицензией копирайт.

4.2 Организация проекта

Начиная с данного раздела, все указанные файловые пути представлены в формате UNIX, а путь «./» обозначает корневой каталог бенчмарка.

Было принято решение разделить каталоги исходных и объектных кодов «./src/» и «./bin/» соответственно.

Для разделения сущностей, спроектированных в разделе 3, было принято решение создать следующие каталоги:

- «./src/benchmark/» для бенчмарков;
- «./src/controller/» для контроллеров;
- «./src/generator/» для генераторов;
- «./src/graph/» для графов, вершин и рёбер;
- «./src/mpi/» для вспомогательных классов, связанных с MPI;
- «./src/task/» для задач;
- «./src/util/» для вспомогательных утилит.

Для того чтобы реализации отдельных задач не смешивались, было принято решение отделить их, разместив в отдельных каталогах. Для задачи поиска были созданы следующие каталоги:

- «./src/benchmark/search/»;
- «./src/controller/search/»;
- «./src/task/search/».

Так как сборка приложения производится утилитой make, было принято решение поместить описание сборки в файл «./makefile».

Для упрощения автоматической сборки проекта, оболочка для запуска – файл «./src/main_dgmark.cpp». Различные конфигурации запуска должны формироваться на основании директив препроцессора.

Путь к файлу лицензии, необходимому в соответствии с пунктом 4.1.3, должен быть равен «./LICENSE».

Объектные коды должны располагаться в «./bin/» по путям, аналогичным иерархии исходных. Исполняемые файлы должны располагаться в «./bin/».

4.3 Реализация сборки проекта

Сборка проекта осуществляется утилитой make. Данная утилита позволяет задать процесс сборки в декларативном стиле.

4.3.1 Описание утилиты make

Основная идея заключается в задании целей сборки, их зависимостей, а также команд, которые выполняются при удовлетворении всех зависимостей. Зависимости являются другими целями, либо файлами, которые должны существовать. При проверке зависимости проверяется, изменялся ли файл, и была ли уже выполнена цель, связанная с этим файлом. Если он не изменялся, а цель была выполнена, то зависимость автоматически считается удовлетворённой. Иначе зависимость принимается за цель, и ищется в списке и выполняется

При запуске утилиты можно указать цель, в противном случае используется стандартная (all).

Синтаксис предусматривает возможность создания переменных, хранящих временное значение, указание нескольких целей, зависимостей и команд, использование целей с шаблонами и условных операторов.

Синтаксис:

```
#комментарий
переменная1 = значение1 #параметр команды 1
цель: зависимость
    команда1 $(переменная1)
цель1 цель2 ...: зависимость1 code.o ...
    команда2
%.o: %.cpp  #шаблон для всех целей, с суффиксом «.o»
    g++ -c $< -o $@  #компиляция
# $< - переменная, хранящая зависимости
# $@ - переменная, хранящая цели
```

Пример использования:

```
BUILD = target
all: $(BUILD) # стандартная цель
# basic build
target : $(OBJ_DIR_PATHS) $(OBJECTS)
  $(MPICPP) $(CPPFLAGS) $(OBJECTS) -o $(addprefix $(BIN_DIR), $@)
# подготовка каталогов
$(OBJ_DIR_PATHS):
  mkdir -p $@
# компиляция исходных кодов
$(OBJ_DIR)%.o: $(SRC_DIR)%.cpp
  $(MPICPP) $(CPPFLAGS) -c $< -o $@
```

В примере указана цель для сборки по умолчанию – «target». В ней указана линковка списка объектов $(OBJECTS) в исполняемый файл. Зависимости – каталоги объектных файлов $(OBJ_DIR_PATHS), и все необходимые объектные файлы.

Необходимые каталоги создаются командой mkdir, а объектные файлы получаются компиляцией из исходных аналогично примеру.

4.3.2 Описание параметров сборки DGraphMark

Для того чтобы была возможность управлять сборкой, было принято решение ввести набор стандартизированных параметров.

- OPENMP. Возможные значения: true, false. Включает и выключает компиляцию проекта с поддержкой директив OpenMP.
- BUILD_GRAPH500_BFS. Возможные значения: true, false. Включает и выключает сборку исполняемых файлов с алгоритмами поиска, взятыми из graph500.
- IS_GRAPH_ORIENTED. Возможные значения: true, false. Если установлено значения false, граф деориентируется после генерации.
- GRAPH_GENERATOR_TYPE. Возможные значения: KRONECKER, UNIFORM. Тип используемого генератора графов.
- VALIDATOR_DEPTH_BUILDER_TYPE. Возможные значения: BUFFERED, P2PNOCLOCK. Тим используемого DepthBuilder, используемого при валидации.
- OPENMP_FLAG. Значение флага включения директив OpenMP для выбранного компилятора.
- MPICPP. Выбранный компилятор MPI-программ.
- BUILD. Список целей для сборки по умолчанию.

4.3.3 Описание процесса сборки DGraphMark

В связи с тем, что некоторые параметры сборки влияют на формирование алгоритмов, было принято решение передавать их в исходный код через параметры компиляции как define-выражения.

Сборка проекта делится на три этапа:

- формирование флагов компиляции на основании введённых параметров;
- формирование списка файлов исходных кодов, которые необходимо скомпилировать;
- компиляция файлов исходных кодов и линковка в соответствии с заданными целями сборки.

Формирование списка файлов исходных кодов выглядит следующим образом

```
BENCHMARK = Benchmark search/SearchBenchmark
CONTROLLER = Controller search/SearchController
GENERATOR = RandomGenerator UniformGenerator KroneckerGenerator
GRAPH = Graph CSRGraph GraphDistributor
MPI = Communicable RMAWindow BufferedDataDistributor
TASK = ParentTree ParentTreeValidator SearchTask
BFS = BFSdgmark BFSGraph500P2P BFSGraph500RMA BFSTaskRMAFetch
BFSTaskP2P BFSTaskP2PNoBlock
VALIDATOR = DepthBuilder DepthBuilderBuffered
DepthBuilderP2PNoBlock
UTIL = Statistics Random
```

В переменные, соответствующие именам определённых в подразделе 4.2 каталогов, записывается список имён файлов исходных кодов, лежащих в нём. В дальнейшем формируется список файлов исходных и объектных кодов через добавление в качестве префикса соответствующего каталога, а в качестве суффикса файлового расширения

```
FILES_LIST = $(addprefix $(BENCHMARK_DIR), $(BENCHMARK))
FILES_LIST += $(addprefix $(CONTROLLER_DIR), $(CONTROLLER))
SOURCES = $(addprefix $(SRC_DIR), $(addsuffix .cpp,
$(FILES_LIST)))
OBJECTS = $(addprefix $(OBJ_DIR), $(addsuffix .o, $(FILES_LIST)))
```

В дальнейшем данные переменные используются в качестве зависимостей и целей при сборке

```
# Цель – создание каталогов.
$(OBJ_DIR_PATHS):
	mkdir -p $@
# Цель – основная сборка
dgmark dgmark_% graph500% : $(OBJ_DIR_PATHS) $(OBJECTS)
	rm -f $(OBJ_DIR)main_dgmark.o;
	$(MPICPP) $(CPPFLAGS) -DTASK_TYPE_$@ -c
$(SRC_DIR)main_dgmark.cpp -o $(OBJ_DIR)main_dgmark.o
	$(MPICPP) $(CPPFLAGS) $(OBJECTS) -o $(BIN_DIR)$@;
# Цель – компиляция исходных кодов
$(OBJ_DIR)%.o: $(SRC_DIR)%.cpp
	$(MPICPP) $(CPPFLAGS) -c $< -o $@
# Цель – очистка объектных файлов
.PHONY : clean
clean:
	rm -rf $(BIN_DIR)*
```

Для того, чтобы выбрать тип задачи, файл «./main_dgmark.cpp» перекомпилируется для каждой конечной цели сборки с соответствующим параметром препроцессора «-DTASK_TYPE_$(цель)».

4.4 Описание общих алгоритмов

Представленные в данном и следующих подразделах алгоритмы классифицированы по следующим категориям:

- RMA_FETCH – использует операции MPI RMA, блокирующий в связи с синхронизацией процессов через операцию fetch;
- P2P – использует операции MPI P2P, блокирующий (единовременно активен только один процесс, остальные находятся в пассивном по отношению к нему режиме);
- P2P_NOBLOCK – использует операции MPI P2P, неблокирующий, может использоваться буферизация запроса на получение данных;
- P2P_NOBLOCK_BUFFERED – P2P_NOBLOCK, в котором происходит рассылка данных без получения ответа, используется буферизация запросов отправки и получения.

Категории упорядочены по теоретическому возрастанию производительности решения задач рассылки. Это такие задачи, в которых каждый процесс рассылает остальным некоторый набор данных, и при этом не требуется получение ответного сообщения на запрос.

4.4.1 Общий подход к реализации алгоритмов класса RMA_FETCH

Операции MPI RMA позволяют производить операции произвольного доступа к данным, хранящимся в локальной памяти. Для осуществления доступа к памяти используются специальные объекты – RMA Окна (RMA Windows, в дальнейшем окна).

Стандартом MPI 3.0 описано два пути к созданию окон:

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
MPI_Info info, MPI_Comm comm, MPI_Win *win);

int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, void *baseptr, MPI_Win *win);
```

Функция Mpi_Win_create создаёт окно для существующего массива данных. Параметры:

- base – указатель на начало массива;
- size – количество элементов в массиве;
- disp_unit – размер элемента массива;
- info – дополнительная информация для создания (можно использовать MPI_INFO_NULL);
- comm – коммуникатор параллельной системы;
- win – указатель для созданного окна.

Функция Mpi_Win_allocate создаёт окно и выделяет память под массив. Параметры:

- size – количество элементов в массиве;
- disp_unit – размер элемента массива;
- info – дополнительная информация для создания (можно использовать MPI_INFO_NULL);
- comm – коммуникатор параллельной системы;
- base_ptr – указатель на начало созданного массива;
- win – указатель для созданного окна.

После завершения использования, окно необходимо закрыть. Это действие производится операцией MPI_Win_free. При этом память, привязанная к окну, освобождается только при использовании второго метода создания окна.

Для считывания данных из окон предусмотрено несколько операций. В DGraphMark используются следующие:

- MPI_Put – помещение буфера данных в окно выбранного узла с выбранным смещением;
- MPI_Get – считывание буфера данных из окна выбранного узла с выбранным смещением;
- MPI_Accumulate – аккумуляция буфера данных в окне выбранного узла с выбранным смещением и операцией (сумма, разность и другие).

При использовании fetch-синхронизации каждый из процессов должен вызвать метод MPI_Win_fence до и после выполнения RMA-операций. Заполнение данными запрашиваемых буферов не гарантировано внутри блока fence, но обязательно осуществляется при выходе за его пределы.

Пример.

```
MPI_Win_fence(assert, win);
//здесь выполняются операции с окнами
//полученные данные нельзя использовать
MPI_Win_fence(assert, win);
//полученные данные можно использовать
```

Параметр assert при использовании fence-синхронизации позволяет оптимизировать выполнение операций. Несколько параметров можно ввести через операцию OR. Можно указать, что внутри блока fence не происходит отсылки данных (MPI_MODE_NOPUT, в открывающем), а также что не происходит запроса данных (MPI_MODE_NOSTORE, в закрывающем). Также по возможности необходимо помечать открывающий fence как MPI_MODE_NOPRECEDE, а закрывающий как MPI_MODE_NOSUCCEED.

4.4.2 Общий подход к реализации алгоритмов класса P2P

При реализации алгоритмов данного класса коммуникации между процессами выполняются с помощью блокирующих операций MPI P2P: MPI_Send, MPI_Recv, а также их буферизированных и стандартных аналогов.

Также допускается использование пробных операций: MPI_Iprobe, и аналогов.

4.4.3 Общий подход к реализации алгоритмов класса P2P_NOBLOCK

При реализации алгоритмов данного класса коммуникации между процессами приоритетно выполняются с помощью неблокирующих операций MPI P2P: MPI_Isend, MPI_Irecv, а также их буферизированных и стандартных аналогов.

4.4.4 Общий подход к реализации алгоритмов класса P2P_NOBLOCK_BUFFERED

При реализации алгоритмов данного класса используются инструменты класса P2P_NOBLOCK, а также буферизация отправляемых сообщений.

Для каждого процесса создаётся буфер отправки, который отправляется только после заполнения или в конце коммуникации. При этом происходит наполнение сетевых пакетов данными, размером близкими к MTU (maximum transmission unit, максимальный размер полезного блока данных), что приводит к уменьшению нагрузки на сеть и повышению производительности. Рост ускорения заканчивается при превышении буфером размера сетевого пакета.

Использование данного алгоритма существенно повышает производительность, когда временные затраты на обработку единицы пересылаемых данных существенно ниже затрат на пересылку данных.

4.5 Описание реализации задачи распределения

Задачей распределения назовём такую задачу, в которой каждый узел отправляет остальным узлам некоторые наборы данных, возможно, отличающиеся. В таких задачах отсутствуют зависимости между данными, рассылка возможна в любом количестве. Вследствие этого, возможна реализация наиболее эффективного класса решения P2P_NOBLOCK_BUFFERED.

Данная задача встречается часто, и поэтому было принято решение разделить реализацию на две части: общую и частную. В общей будут описаны основные алгоритмы, а в частных протоколы рассылки и получения данных.

Краткая схема общего алгоритма распределения данных представлена на рисунке 4.1.

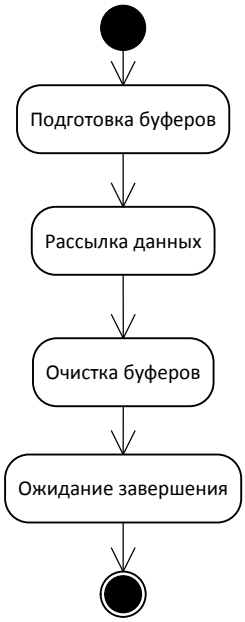

Рисунок 4.1 – Краткая схема алгоритма распределения данных

Общая реализация должна представлять собой абстрактный класс BufferedDataDistributor, который должен описывать общие методы: подготовки буферов prepareBuffers(), очистки буферов flushBuffers() и ожидания завершения waitEnd().

Также должны быть определены методы отправки буфера sendData(target) и синхронизации данных probeSynchData().

У классов наследников должны быть определены методы рассылки данных и обработки полученных сообщений processRecvData(count).

В классе BufferedDataDistributor необходимо определить следующие объекты:

- sendPackageSize – количество объектов в отправляемом пакете данных;
- objSize – размер одного объекта;
- elementSize – объектов в одном отправляемом элементе;
- objDataType – MPI тип данных отправляемого объекта;
- void **sendBuffer – буферы отправляемых данных для каждого узла;
- void *countToSend – количество данных, для отправки в узлы;
- void *recvBuffer – буфер входных данных;
- Request *sendRequest –запросы отправки;
- Request recvRequest – запрос получения данных;
- bool *isSendRequestActive – флаги активности запросов отправки;
- bool isRecvRequestActive – флаг активности запроса на получение.

Перед тем, как начать рассылку данных, необходимо подготовить буферы данных и вспомогательные объекты: установить флаги активности всех запросов в состояние «неактивен», и количество готовых к отправке элементов для каждого узла в ноль.

```
void BufferedDataDistributor::prepareBuffers()
{
	for (int reqIndex = 0; reqIndex < size; ++reqIndex) {
		countToSend[reqIndex] = 0;
		isSendRequestActive[reqIndex] = false;
	}
	isRecvRequestActive = false;
	countEnded = 0;
}
```

После завершения основного этапа рассылки необходимо проверить, пусты ли буферы исходящих сообщений, и если нет, то провести отправку. Также необходимо для каждого узла отправить пустой массив данных, оповещающий о том, что данный узел завершил работу.

```
for (int reqIndex = 0; reqIndex < size; ++reqIndex) {
	if (reqIndex == rank) {
		continue;
	}
//Отправка оставшихся неотправленных элементов в буферов
	if (!isSendRequestActive[reqIndex]
			&& countToSend[reqIndex] > 0) {
		sendData(reqIndex);
	}
//Отправка пустого буфера для индикации завершения работы
	sendData(reqIndex);
		while (isSendRequestActive[reqIndex]) {
		probeSynchData();
	}
}
```

В конце алгоритма рассылки данных должно быть выполнено ожидание завершения отправки сообщений всеми процессами. Пока существуют процессы, не завершившие рассылку, необходимо выполнять синхронизацию данных.

```
void BufferedDataDistributor::waitForOthersToEnd()
{
	++countEnded; //Текущий поток закончил
	while (countEnded < size) { //size – количество процессов
		probeSynchData();
	}
	probeSynchData();
}
```

В методе синхронизации данных должен создаваться запрос на получение данных, а также передача полученных данных в метод обработки.

```
void BufferedDataDistributor::probeSynchData()
{
      Status status;
// Активация запроса на считывание данных, если он уже отработал
      if (!isRecvRequestActive) {
            isRecvRequestActive = true;
            recvRequest = comm->Irecv(recvBuffer,sendPackageSize,
                  VERTEX_TYPE, ANY_SOURCE, DISTRIBUTION_TAG);
      }
// Проверка активности запроса на получение данных и считывание
      if (isRecvRequestActive && recvRequest.Test(status)) {
            isRecvRequestActive = false;
            const size_t dataCount =
status.Get_count(VERTEX_TYPE);
            if (dataCount > 0) {
                  processRecvData(dataCount);
            } else {
                  ++countEnded;
            }
      }
}
```

Перед отправкой данных классы-наследники должны проверять два условия: запрос к выбранному процессу не должен быть активен (в противном случае должна выполняться синхронизация данных во время ожидания выполнения), а также отправной буфер должен быть заполнен полностью. Отправку данных можно осуществить общим методом sendData.

```
void BufferedDataDistributor::sendData(int toRank)
{
      while (isSendRequestActive[toRank]) {
            probeSynchData();
      } // Если запрос ещё не отработал, необходимо ожидать
      // Отправка данных буфера
      sendRequest[toRank] = comm->Isend(&sendBuffer[toRank][0],
      countToSend[toRank], VERTEX_TYPE, toRank, DISTRIBUTION_TAG);
      countToSend[toRank] = 0;
      isSendRequestActive[toRank] = true;
}
```

4.6 Описание реализаций алгоритмов BFS

Детали реализаций алгоритмов BFS разнесены по отдельным классам-наследникам, а основные детали сконцентрированы в классе BFSdgmrak.

В нём определяется порядок обработки запроса и формирования результата работы задания, общая схема обхода графа, определяются очереди (текущего и следующего шага).

Общий алгоритм BFS показан в рисунках 4.2 и 4.3.

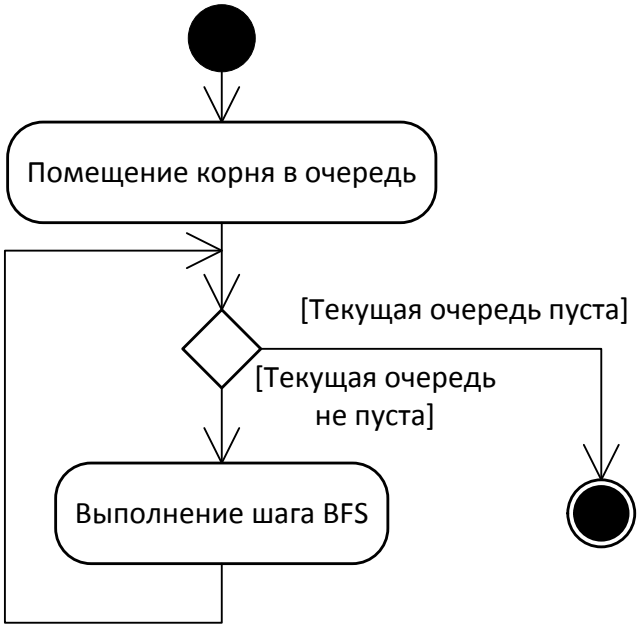

Рисунок 4.2 – Общий алгоритм BFS

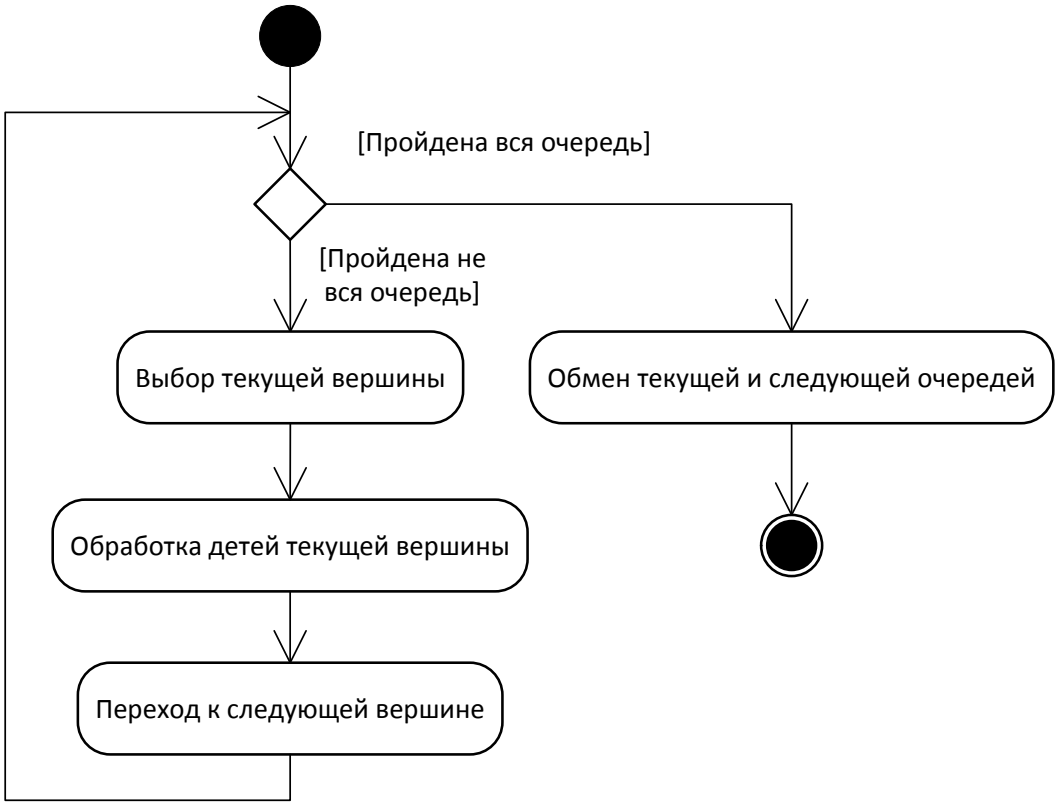

Рисунок 4.3 – Алгоритм выполнения шага BFS

Как показано на рисунке 4.3, BFS выполняется итеративно: как только текущие очереди всех процессов оказываются пусты, выполнение прекращается, и возвращается результат.

```
while (isNextStepNeeded()) { //Проверка наличия в очереди вершин
      performBFS();          //Выполнение одного шага BFS
      comm->Barrier();
      swapQueues();          //Обмен текущей и следующей очередей
}
```

Также можно выделить метод обхода текущей очереди, алгоритм которого приведён на рисунке 4.4. Для каждой из вершин текущей очереди последовательно просматривается список вершин, с ней связных. Данные вершины отправляется в метод локальной обработки соответствующего узла. При локальной обработке вершина добавляется в очередь и у неё определяется предок, если ещё не был определён. В противном случае ничего не происходит.

```
void BFSdgmark::performBFSActualStep()
{
      vector<Edge*> *edges = graph->edges;
      size_t queueEnd = queue[0]; // Обход всех вершин очереди
      for (size_t qIndex = 1; qIndex <= queueEnd;++qIndex) {
            Vertex currVertex = queue[qIndex];
            //Обход всех рёбер, ведущих из текущей
            size_t childStart = graph->getStartIndex(currVertex);
            size_t childEnd = graph->getEndIndex(currVertex);
            for (size_t childIndex = childStart;
                        childIndex < childEnd; ++childIndex) {
                  Vertex child = edges->at(childIndex)->to;
                  Vertex childLocal = graph->vertexToLocal(child);
                  int childRank = graph->vertexRank(child);
                  if (childRank == rank) { // Локальная обработка
                        processLocalChild(currVertex, childLocal);
                  } else {                  // Глобальная обработка
                        processGlobalChild(currVertex, child);
                  }
            }
      }
}
```

Реализация метода performBFS, выполняющего обход всех вершин, лежащих в текущей очереди, производится в классах-наследниках, и формирует основу алгоритма решения задачи. Также в классах-наследниках должен быть определён метод глобальной обработки вершины.

4.6.1 Реализация BFS класса RMA_FETCH

Порядок выполнения в данной реализации блокированный, в связи с ограничениями fetch-синхронизации. Этапы невозможно объединить, так как наборы данных могут существенно отличаться, а при fetch-синхронизации

необходимо, чтобы каждый из узлов выполнил вход и выход из блока команд fetch. Одновременно лишь один процесс может быть активным и выполнять обход вершин, все остальные должны выполнять fetch-синхронизацию по запросу текущего активного

```
for (int node = 0; node < size; ++node) {
      if (rank == node) {
            performBFSActualStep();
            endSynch(BFS_SYNCH_TAG);
      } else
            performBFSSynchRMA();
      comm->Barrier();
}
```

При обработке глобальной вершины последовательно запрашиваются параметры массива родителей соответствующего вершине узла. Если у указанной вершины ещё нет предка, то выполняется запись в очередь.

```
Vertex parentOfChild;
pWin->fenceOpen(MODE_NOPUT); //Получение значения родителя
вершины
pWin->get(&parentOfChild, 1, childRank, childLocal);
pWin->fenceClose(0);
//Условие непосещённости
if (parentOfChild == graph->numGlobalVertex) {
      pWin->fenceOpen(0); //Обновление массива родителей
      pWin->put(&currVertex, 1, childRank, childLocal);
      pWin->fenceClose(MODE_NOSTORE);
      Vertex queueLastIndex;
      nextQWin->fenceOpen(MODE_NOPUT); //Получение длины очереди
      nextQWin->get(&queueLastIndex, 1, childRank, 0);
      nextQWin->fenceClose(0);
      nextQWin->fenceOpen(0); // Обновление очереди
      nextQWin->put(&childLocal, 1, childRank, ++queueLastIndex);
      nextQWin->put(&queueLastIndex, 1, childRank, 0);
      nextQWin->fenceClose(MODE_NOSTORE);
}
```

При этом метод синхронизации выглядит следующим образом

```
while (waitSynch(BFS_SYNCH_TAG)) {
      pWin->fenceOpen(MODE_NOPUT); // Доступ к массиву родителей
      pWin->fenceClose(MODE_NOSTORE);
      if (waitSynch(BFS_SYNCH_TAG)) {
            pWin->fenceOpen(MODE_NOPUT);    //Доступ к
            pWin->fenceClose(MODE_NOSTORE); // массиву родителей
            nextQWin->fenceOpen(MODE_NOPUT); //Доступ к очереди
            nextQWin->fenceClose(MODE_NOSTORE);
            nextQWin->fenceOpen(MODE_NOPUT); //Доступ к очереди
            nextQWin->fenceClose(MODE_NOSTORE);
      }
}
```

4.6.2 Реализация BFS класса P2P

Данная реализация является прямым переносом RMA_FETCH реализации с заменой операций на блокирующие операции MPI P2P. При применении данных операций появляется возможность для оптимизации: можно избавиться от формата запрос/запись между узлами. Вместо них можно отправлять набор данных предок-потомок в конечный процесс, в котором они будут обработаны в соответствии с правилами локальной обработки.

```
//Алгоритм отправки
Vertex memory[2] = {childLocal, currVertex};
requestSynch(true, childRank, BFS_SYNCH_TAG);
comm->Send(&memory[0], 2, VERTEX_TYPE, childRank, BFS_DATA_TAG);

//Алгоритм синхронизации
while (waitSynch(BFS_SYNCH_TAG, status)) {
     comm->Recv(memory,2,VERTEX_TYPE,status.Get_source(),
BFS_DATA_TAG);
     const Vertex currLocal = memory[0];
     const Vertex parentGlobal = memory[1];
     BFSdgmark::processLocalChild(parentGlobal, currLocal);
}
```

4.6.3 Реализация BFS класса P2P_NOBLOCK_BUFFERED

В связи с тем, что BFS можно представить в виде задачи распределения, было принято решение о реализации данного алгоритма. Каждый шаг BFS представляется в виде отдельной задачи распределения, в котором роль рассылки выполняет метод обхода текущей очереди.

При обработке глобальной вершины наполняется буфер, а при заполнении вызывается его отправка

```
void BFSTaskP2PNoBlock::processGlobalChild(Vertex currVertex,
Vertex child)
{
     const Vertex childLocal = graph->vertexToLocal(child);
     const int childRank = graph->vertexRank(child);
     size_t &currCount = countToSend[childRank];
     Vertex *&currBuffer = sendBuffer[childRank];
     // Заполнение буфера
     sendBuffer[childRank] [currCount] = childLocal;
     sendBuffer[childRank] [currCount + 1] = currVertex;
     //Отправка при заполнении
     if ( (currCount += 2) == sendPackageSize) {
          sendData(childRank);
     }
     probeSynchData();
}
```

При обработке полученного массива вершин последовательно для каждой вызывается метод локальной обработки вершин. В нём вершина добавляется в следующую очередь, если не была до этого посещена.

4.7 Описание алгоритмов генерации графов

Алгоритмы генерации графов сосредоточены в наследниках случайного генератора (RandomGenerator): равномерном генераторе и генераторе, использующем алгоритм Кронекера (описание в пункте 2.1.2).

4.7.1 Равномерный генератор

При равномерной генерации графа, в каждом из процессов для каждой из локальных вершин создаётся по *depth* рёбер. Ранг и локальное значение вершины назначения ребра генерируются с помощью 64-разрядного равномерного генератора случайных чисел, вследствие чего вершина назначения распределена равномерно.

4.7.2 Алгоритм Кронекера (2x2, R-MAT)

При генерации графа алгоритмом Кронекера в каждом узле создаётся по $numVertexLocal \cdot density$ рёбер, которые распределяются с помощью алгоритма распределения графа.

4.7.3 Алгоритм распределения и деориентации графа

Задача распределения рёбер графа является задачей распределения, в связи с чем её можно реализовать на основании класса BufferedDataDistributor. В отличие от общей задачи распределения, в данной задаче набор рассылаемых данных может быть неизвестен заранее. В связи с этим необходимо сформировать методы открытия и закрытия задачи, а также метод распределения ребра.

Метод открытия задачи должен выполнять методы, предшествующие методу рассылки данных алгоритма, представленного на рисунке 4.1. Повторное открытие задачи до закрытия не должно приводить к инициализации буферов.

Метод закрытия задачи должен выполнять методы, следующие после метода рассылки данных. Повторное закрытие задачи не должно приводить к очистке буферов и ожиданию завершения.

```
void GraphDistributor::open()
{
    if (!isOpen) {
        isOpen = true;
        prepareBuffers();
    }
}
```

```
void GraphDistributor::close()
{
    if (!isOpen) {
        isOpen = false;
        flushBuffers();
        waitForOthersToEnd();
    }
}
```

Метод отправки ребра является методом рассылки, и может быть повторён неоднократно. Должна быть предусмотрена локальная и глобальная обработка. При обработке полученных сообщений должна вызываться локальная обработка данных.

```
void GraphDistributor::sendEdge(Vertex from, Vertex to)
{
      const int toRank = graph->vertexRank(from);
      if (toRank == rank) {// Локальная рассылка
            edges->push_back(new Edge(from, to));
      } else {// Глобальная рассылка
            sendEdgeExternal(from, to, toRank);
      }
      probeSynchData();
}

void GraphDistributor::sendEdgeExternal(Vertex from, Vertex to,
int toRank)
{
      size_t &currCount = countToSend[toRank];
      Vertex *&currBuffer = sendBuffer[toRank];
            currBuffer[currCount] = from;
      currBuffer[currCount + 1] = to;
      currCount += elementSize;
      if (currCount == sendPackageSize) {
            sendData(toRank);
      }
}
```

Задача деориентации – частный случай задачи распределения рёбер. В ней для каждого ребра графа генерируется обратное, и затем распределяется.

```
const size_t initialEdgesCount = edges->size();
for (size_t eIndex = 0; eIndex < initialEdgesCount; ++eIndex) {
      const Edge * const edge = edges->at(eIndex);
      sendEdge(edge->to, edge->from);
}
```

4.8 Описание алгоритмов DepthBuilder

Важной частью валидации построенного массива родителей является проверка массива на отсутствие циклов. Одним из оптимальных алгоритмов проверки этого является построение массива глубин вершин. Под глубиной понимается количество рёбер, которые необходимо пройти, чтобы попасть в корень. Глубина корня принимается равной нулю.

Существует два подхода к построению массива глубин.

- Прямой. Аналогичен BFS – глубина каждой готовой и ещё не отправленной вершины рассылается до тех пор, пока существуют неотправленные вершины.
- Обратный. Глубина вершин последовательно запрашивается до тех пор, пока хотя бы один массив глубин изменяется.

После построения массива глубин можно проверить, есть ли в массиве предков цикл. Если есть, то найдутся вершины, для которых будет существовать предок, но не будет построенной глубины. Связано это с тем,

что в данной структуре данных возможны лишь циклы, не ведущие в корень. Доказательство. Для любого связного дерева справедливо равенство

$$|V| = |E| + 1\ , \qquad (4.1)$$

где $|E|$ - количество рёбер,
$|V|$ – количество вершин.

При добавлении дополнительного ребра между любыми двумя вершинами образуется цикл, но нарушается равенство 4.1. При перемещении ребра образуется цилкл, но теряется связность дерева.

Массив предков образует структуру данных, в которой описано numVertexGlobal вершин и на одно меньше ребро, так как предком корня является он сам. В связи с этим массив предков может быть либо деревом, либо быть несвязным и иметь цикл.

Таким образом, если глубина оказалась построена для всех вершин, то цикла в массиве родителей нет.

4.8.1 Прямой, P2P_ NOBLOCK_BUFFERED

В связи с тем, что прямой подход похож на BFS, можно представить его в виде задачи распределения глубин и унаследовать от BufferedDataDistributer. Схема алгоритма его приведена на рисунке 4.4.

Для каждой вершины необходимо завести состояния: не пройдена, заполнена (глубина заполнена, но не отправлена), отправлена. На этапе рассылки должны быть пройдены все вершины и для тех, что находятся в состоянии «заполнена», должна быть отправлена глубина во все смежные с ними вершины.

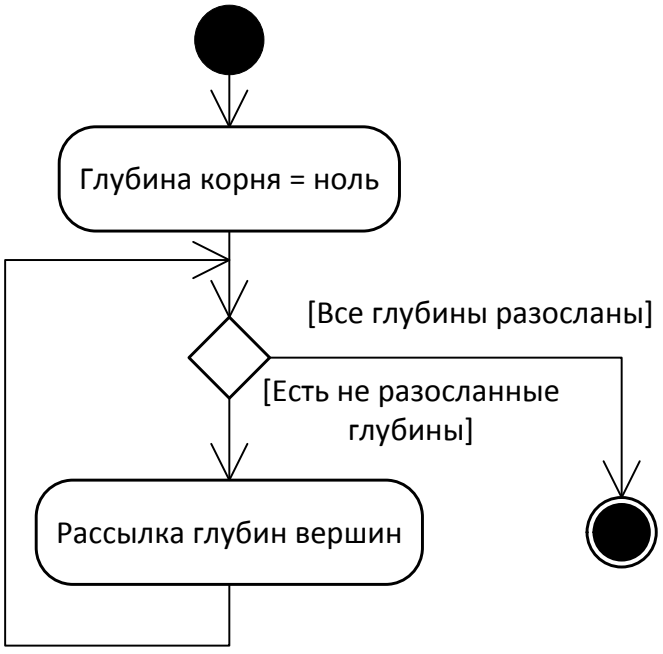

Рисунок 4.4 – Прямой алгоритм построения массива глубин

```
for (Vertex local = 0; local < graph->numLocalVertex; ++local) {
      if (vertexState[local] == stateJustFilled) {
            distributeVertexDepth(local);
            vertexState[local] = stateSent;
            buildState = min(buildState,
                              buildStateNextStepRequired);
      }
}
```

Отправка данных аналогична отправке данных в буферизованном варианте BFS за тем исключением, что вместо потомка отправляется текущая вершина, обновляемая и глубина обновляемой вершины (глубина текущей, увеличенная на единицу).

При обработке полученных данных необходимо проверить, является ли вершина-предок действительно предком. Если глубина вершины не была заполнена, то заполняется.

```
void DepthBuilderBuffered::updateDepth(Vertex parentGlobal,
Vertex localVertex, Vertex newDepth)
{     // Проверка на то, что вершина является родителем
      if (parent[localVertex] != parentGlobal) {
            return;
      }
      Vertex &currDepth = depth[localVertex];
      short &currState = vertexState[localVertex];
      if (currState == stateInitial) { Обновление состояния
            currDepth = newDepth;
            currState = stateJustFilled;
      }
}
```

4.8.2 Обратный, P2P_NOBLOCK

При реализации обратного подхода возможна буферизация только в методе, ответственном за ответы на запросы остальных процессов. Она аналогична буферизации, рассмотренной в алгоритме задачи распределения.

Алгоритмом предписано пытаться обновлять информацию о глубине предка, для каждой незаполненной вершины, пока информация о глубине хотя бы одной вершине изменяется. Основная сложность заключается в запросе глубины вершины, не являющейся локальной. При этом необходимо отправлять запрос и ожидать ответа от стороннего процесса.

Схема алгоритма приведена на рисунке 4.5.

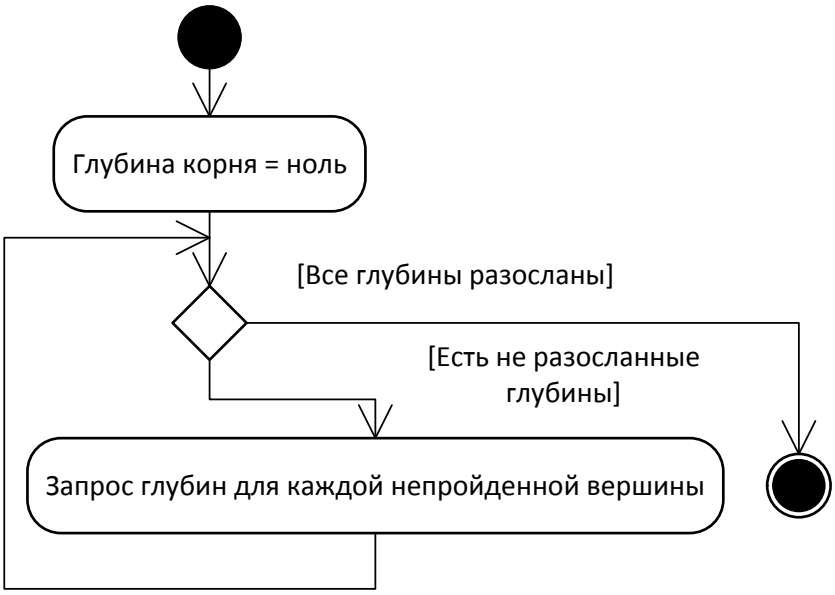

Рисунок 4.5 – Обратный алгоритм построения массива глубин

```
// Запрос глубины сторонней вершины
Vertex tgtDepth;
// Формирование запроса и ответа.
sendVertex(tgtLocal, tgtRank, LOCAL_SEND_TAG);
Request recvReq = comm->Irecv(&tgtDepth, 1, VERTEX_TYPE,
                              tgtRank, DEPTH_SEND_TAG);
while (!recvReq.Test()) {
      synchAction(); // Зжидание ответа
}
```

В данном разделе была создана реализация и рассмотрены алгоритмы DGraphMark. Далее необходимо сравнить качество реализации с Graph500 и проанализировать результаты.

5 Сравнение Graph500 и DGraphMark

Согласно требованиям к качеству бенчмарка, поставленным в пункте 4.1.2, необходимо протестировать расход памяти при работе бенчмарка DGraphMark, время выполнения BFS, валидации и итоговые оценки в сравнении с Graph500.

В связи с тем, что выделение памяти не зависит от пространственной локализации и показателей мощности узлов, а сбор информации проще произвести с одной машины, было принято решение проводить тестирование на одной физической машине.

Для тестирования остальных критериев качества было принято решение использовать параллельные системы с различными характеристиками производительности и пространственной локализации.

Таблицы, содержащие оценки качества реализации бенчмарка, приведены в приложении А.

Пустыми оставлены ячейки в таблицах для тестов, время выполнения которых оказалось существенно больше среднего времени стандартного бенчмарка Graph500 (Graph500 P2P).

Наименования проверенных алгоритмов BFS:

- Graph500 P2P – MPI P2P реализация BFS в Graph500 (исполняемый файл graph500_mpi_simple);
- Graph500 RMA – MPI RMA FETCH реализация BFS в Graph500 (исполняемый файл graph500_mpi_one_sided);
- DGrapgMark P2P – реализация алгоритма из пункта 4.6.2;
- DGrapgMark P2PNB – реализация алгоритма из пункта 4.6.3
- DGrapgMark RMA – реализация алгоритма из пункта 4.6.1.

5.1 Оценка качества использования памяти

Таблицы, представляющие оценку качества использования памяти, представлены на таблицах с А.1 по А.12. В таблицах отображено минимальное, максимальное и среднее количество памяти, выделенное на вычислительный узел, а также суммарное количество выделенной памяти.

Тестирование производилось на двух, четырёх и восьми узлах.

В таблицах с А.5 по А.12 видна существенная неравномерность распределения памяти между узлами в Graph500. Отношение максимального значения к минимальному доходит до 50 раз. Замечено, что данная неравномерность усиливается при повышении количества узлов.

Неравномерность выделения памяти отсутствует у DGraphMark. К тому же показатели в DGraphMark выделяется существенно меньше памяти, как в средних, так и в минимальных случаях.

Также необходимо отметить, что у Graph500 присутствуют утечки памяти в размере около одного процента на один прогон BFS и валидатора. В DGraphMark утечки памяти отсутствуют.

Большинство ошибок с памятью в Graph500 сконцентрировано в коде валидации результата. Было замечено, что именно после первого запуска процедуры валидации и появляется неравномерность. До неё значения памяти для каждого из процесса близки к средним.

На рисунках 5.1 и 5.2 наглядно показана неравномерность в выделении памяти.

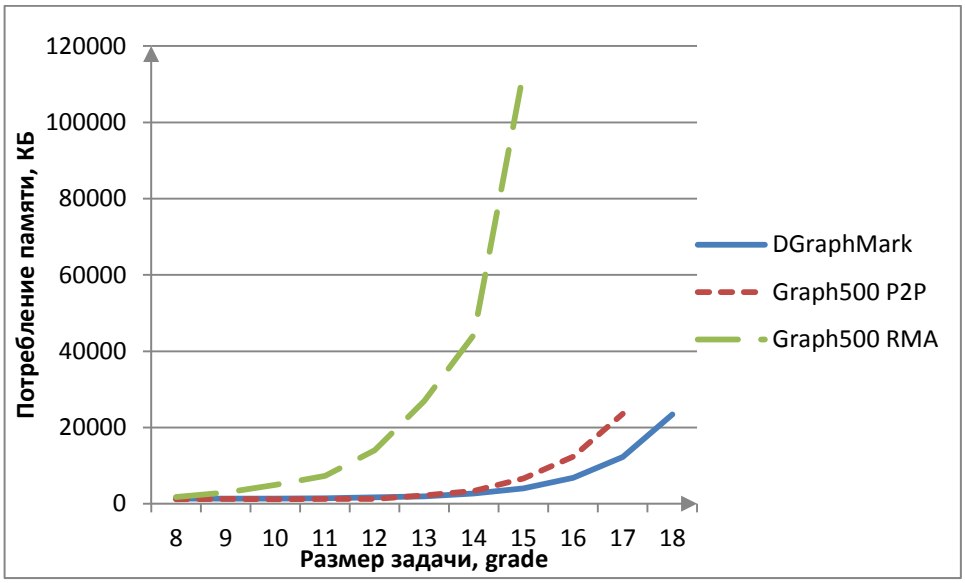

Рисунок 5.1 – Минимум количества выделенной памяти на узел при запуске бенчмарков на восьми узлах

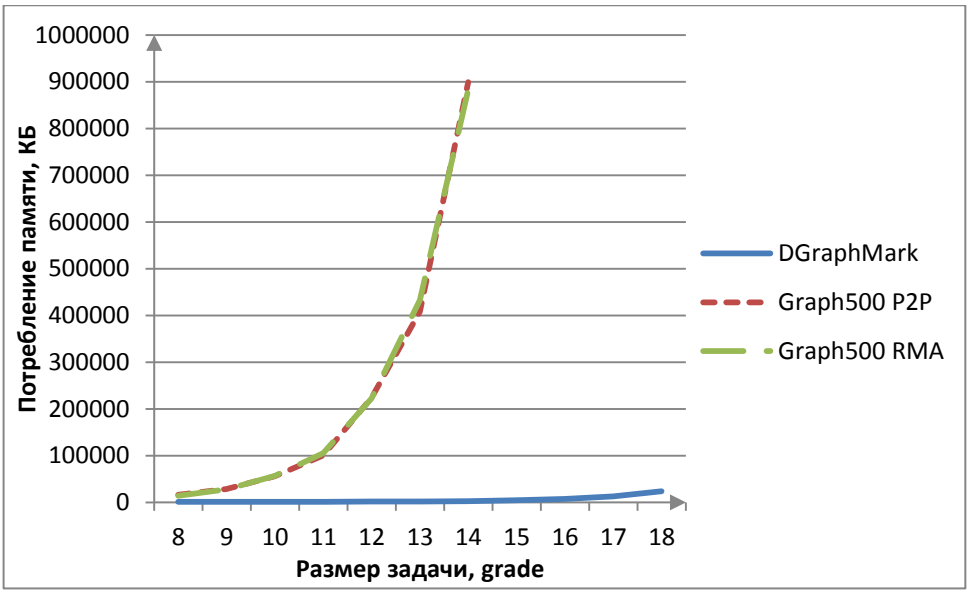

Рисунок 5.2 – Максимум количества выделенной памяти на узел при запуске бенчмарков на восьми узлах

5.2 Оценка качества процедуры валидации

Таблицы, содержащие результаты валидации: А.15, А.21, А.24, А.27, А.30, А.33, А.36, А.39. В них представлены значения времени валидации для различных конфигураций запуска.

Необходимо отметить, что алгоритм обратной валидации существенно падает в производительности при количестве узлов больше физического, а также при запуске на нескольких виртуальных машинах, как под одним гипервизором, так и под разными. Однако при остальных запусках он показывает производительность, сравнимую с алгоритмом прямой валидации, а иногда и выше (несущественно).

Падение производительности связано с наличием большого количества блокирующих операций, на которых основан алгоритм обратной валидации. Они вызваны условными зависимостями между данными. В прямом алгоритме условных зависимостей между данными нет, и поэтому он работает существенно быстрее.

Алгоритм валидации Graph500 работает существенно медленнее прямой валидации, и стабильно работает в условиях, когда отказывает алгоритм обратной валидации.

Было замечено, что общая картина времени валидации меняется в абсолютных величинах при смене конфигурации, но слабо меняется в относительных, в связи с чем для наглядного сравнения можно привести только один график.

На рисунке 5.3 наглядно показаны значения времени валидации из таблицы А.15.

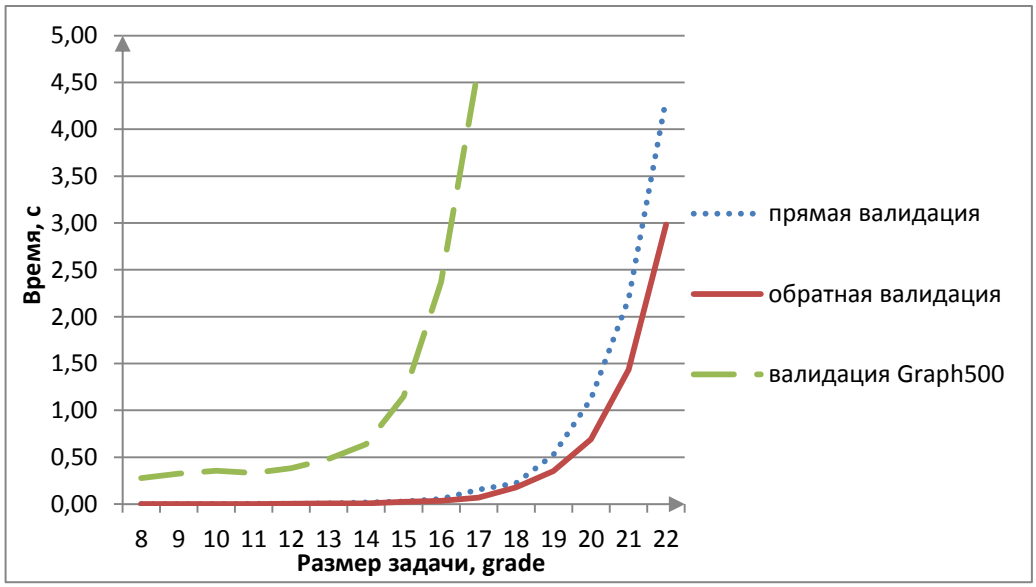

Рисунок 5.3 – Наглядное отображение значений таблицы А.15

5.3 Оценка качества процедуры BFS

Таблицы приложения А, не указанные в подразделах 5.1 и 5.2, содержат данные о времени выполнения и оценках бенчмарков.

По результатам анализа полученных данных было принято решение разделить алгоритмы на две группы: быстрые и медленные.

К быстрым алгоритмам относятся DGraphMark P2PNB и Graph500 P2P. Результаты у данных алгоритмов близки в связи с похожим подходом к решению задачи. Алгоритм Graph500 показал в среднем от двух до трёх раз меньшее время выполнения и во столько же раз большее значение оценки бенчмарка, чем DGraphMark.

Объяснение данному факту может быть следующее: в алгоритме Graph500 отсутствуют вызовы функций и обращения к объектам. Все операции записаны как макросы препроцессора, и скомпонованы внутри одной функции. В связи с этим значительно падает качество исходного кода, но растёт производительность.

Также необходимо отметить, что у данных алгоритмов при запуске на большом количестве узлов аппроксимация оценок переходит из константы в экспоненту, а затем при достижении предела памяти происходит замедление ускорения роста вплоть до константы. Данный факт отображён на рисунках 5.4, 5.5 и 5.6.

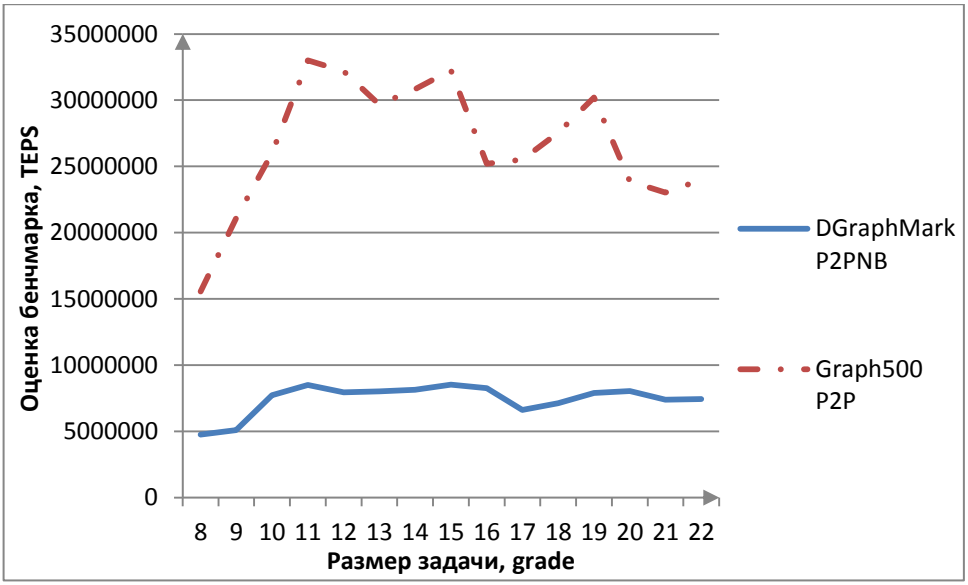

Рисунок 5.4 – Сравнение оценки быстрых алгоритмов BFS, взятое из таблицы А.16

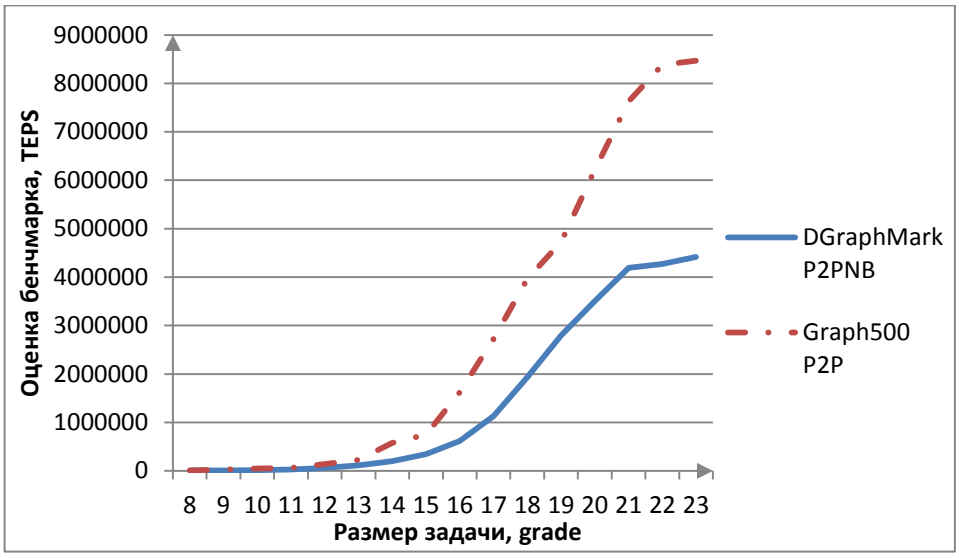

Рисунок 5.5 – Сравнение оценки быстрых алгоритмов BFS, взятое из таблицы А.19

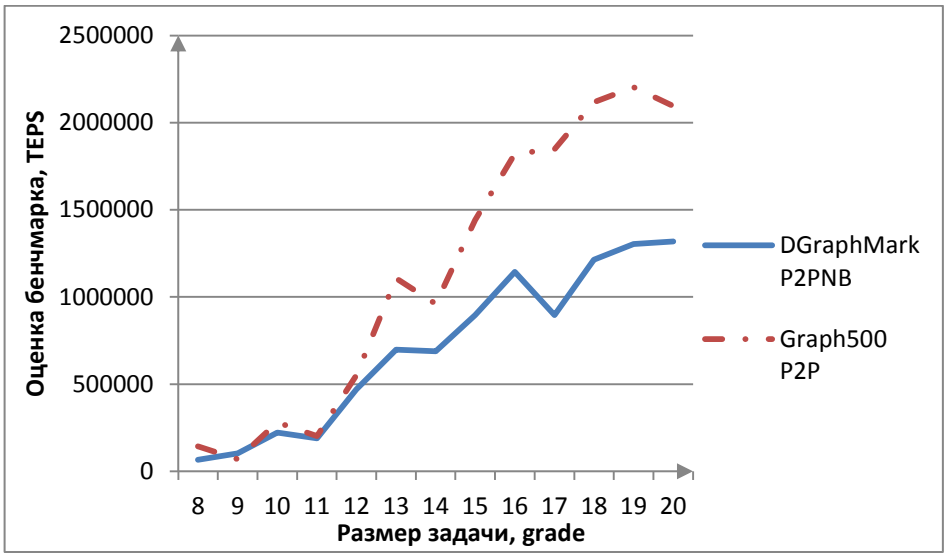

Рисунок 5.6 – Сравнение оценки быстрых алгоритмов BFS, взятое из таблицы А.31

Среди оставшихся алгоритмов наиболее медленный и стабильный – DGraphMark RMA. Он показывает самые низкие оценки и самое большое время выполнения, но при этом результаты его слабо изменяются со временем. Алгоритм DGraphMark P2P также стабилен, однако выполняется существенно быстрее, в связи с чем оценка его выше.

Алгоритм Graph500 RMA имеет в качестве графика оценок BFS горб: при малых размерах задач оценка растёт, достигает максимума, и начинает снижаться.

На рисунках 5.7 и 5.8 показаны графики оценок медленных алгоритмов при различных конфигурациях.

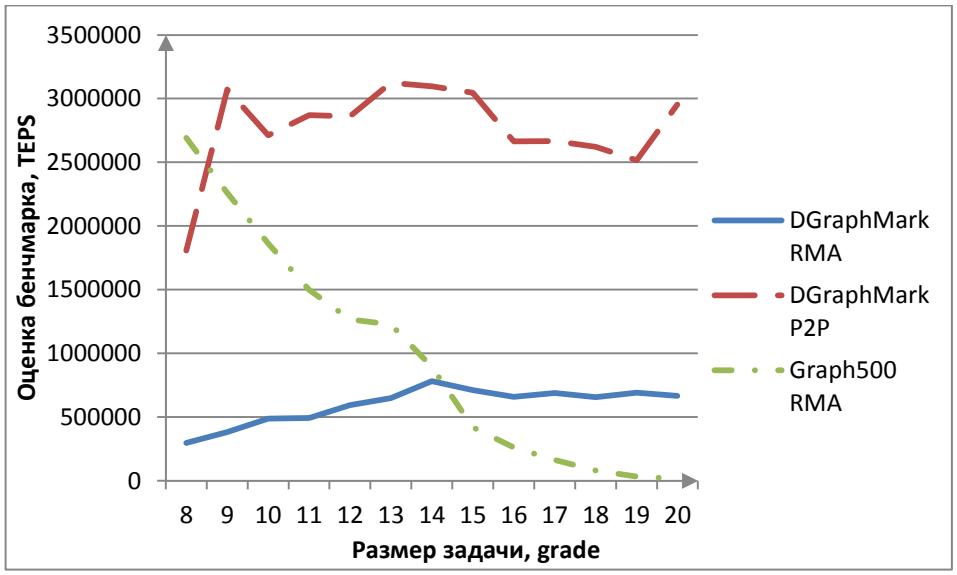

Рисунок 5.7 – Сравнение оценки медленных алгоритмов BFS, взятое из таблицы А.13

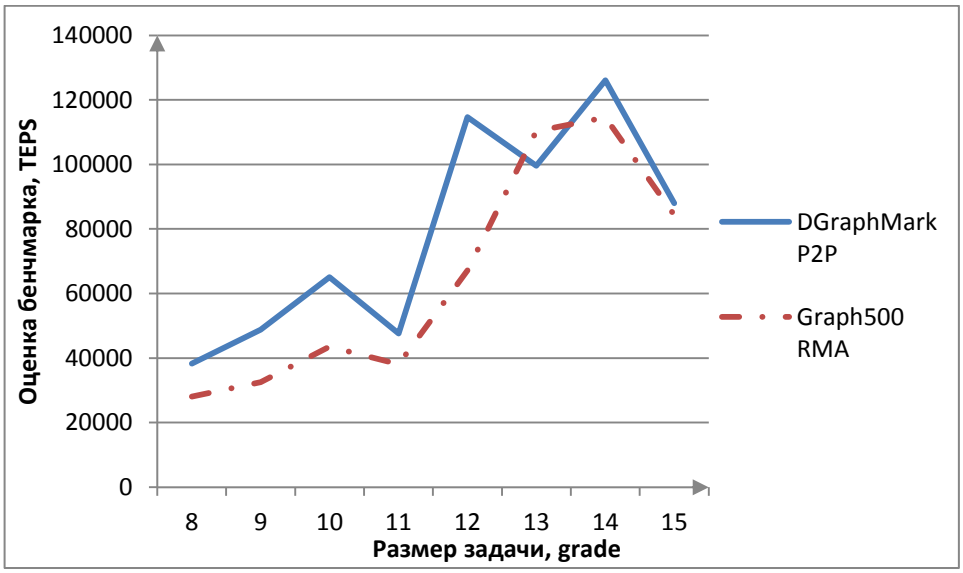

Рисунок 5.8 – Сравнение оценки медленных алгоритмов BFS, взятое из таблицы А.34

При запуске на сложных вариантах конфигураций (на нескольких физически различных узлах) реализации DGraphMark P2P и Graph500 RMA имеют схожий вид, что показано на рисунке 5.8.

Таким образом было произведено сравнение бенчмарков DGraphMark и Graph500. В ходе анализа выяснилось, что DGraphMark эффективнее использует память, а также имеет более быструю процедуру валидации результатов, а Graph500 обладает более быстрым алгоритмом BFS. Однако так как скорость валидации Graph500 оказалась существенно ниже, общая производительность его значительно ниже, чем у DGraphMark.

Заключение

В ходе данной работы было произведено исследование задач области data-intensive, а также спроектирован и разработан бенчмарк, не уступающий по совокупности характеристик аналогам, а в некоторых превосходящий их.

В первом разделе был рассмотрен класс задач data-intensive, а также основные сложности, возникающие при решении задач из этого класса. Были рассмотрены основные ограничения и пути их преодоления.

Во втором разделе был рассмотрен единственный на данный момент бенчмарк, позволяющий оценивать параллельные системы в отношении решения задач класса data-intensive – Graph500. Была рассмотрена общая структура программы, выявлены достоинства и недостатки.

В третьем разделе на основе идей Graph500, а также дополнительных требований по масштабированию был спроектирован бенчмарк DGraphMark. Были учтены все недостатки Graph500, а также большинство его внешних интерфейсов, в связи с чем возможно использование алгоритмов Graph500 внутри DGraphMark, а также сравнение результатов работы.

В четвёртом разделе были установлены основные требования к разработке и лицензированию, а также критерии качества результирующего приложения. Были описаны основные алгоритмы и подходы к решению задач.

В пятом разделе были рассмотрены и проанализированы результаты проведённой работы. Было выявлено, что DGraphMark существенно эффективнее в плане работы с памятью и обладает более быстрой процедурой валидации. Также было отмечено, что скорость работы основного алгоритма BFS в DGraphMark уступает по скорости таковой в Graph500. Однако это не является ограничением, так как ускорение, достигнутое в валидации существенно больше потерь, привнесённых BFS. К тому же за счёт грамотного проектирования возможно использование алгоритма BFS Graph500 в DGraphMark, что приведёт к совмещению достоинств.

В результате проведённой работы был разработан бенчмарк, позволяющий оценивать параллельные системы в отношении задач класса data-intensive, существенно превосходящий аналоги в скорости выполнения, требованиям к памяти и потенциальной расширяемости на задачи другого типа.

Приложение А
(справочное).
Таблицы

Таблица А.1 – Количество выделенной памяти при запуске на двух узлах

Grade	Минимум выделения памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1112	4708	5728
9	1344	7348	9240
10	1356	12568	16232
11	1916	23748	30496
12	2552	49312	62444
13	3836	96580	120564
14	6656	220944	244654
15	12036	457444	540121
16	23092	885176	1048768
17	45384	1753596	2165424
18	89572	1805204	2559356
19	177892	1842660	
20	351708	4732687	
21	701996		
22	1401060		

Таблица А.2 – Количество выделенной памяти при запуске на двух узлах

Grade	Максимум выделения памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1128	9084	7668
9	1372	8668	9688
10	1364	15856	17380
11	1932	27516	32812
12	2560	55246	63516
13	3844	116864	150972
14	6660	276184	350664
15	12064	698636	587628
16	23096	1061632	1080676
17	45388	2528148	2215852
18	89584	2631256	2817496
19	177912	2746932	
20	351720	4861846	
21	702001		
22	1401080		

Таблица А.3 – Количество выделенной памяти при запуске на двух узлах

Grade	Среднее выделение памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1122	6596	6698
9	1356	8008	9464
10	1360	14212	16806
11	1926	25632	31654
12	2556	52279	62980
13	3839	106722	135768
14	6658	248564	297659
15	12048	578040	563875
16	23094	973404	1064722
17	45386	2140872	2190638
18	89578	2218230	
19	177904	2294796	
20	351714	4797267	
21	701998		
22	140172		

Таблица А.4 – Количество выделенной памяти при запуске на двух узлах

Grade	Суммарное выделение памяти, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	2244	13192	13396
9	2712	16016	18928
10	2720	28424	33612
11	3852	51264	63308
12	5112	104558	125960
13	7678	213444	271536
14	13316	497128	595318
15	24096	1156080	1127749
16	46188	1946808	2129444
17	90772	4281744	4381276
18	179156	4436460	
19	355808	4589592	
20	703428	9594533	
21	1403996		
22	280344		

Таблица А.5 – Количество выделенной памяти при запуске на четырёх узлах

Grade	Минимум выделения памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1264	2076	3028
9	1268	2104	4412
10	1420	6432	8236
11	1592	11786	17076
12	1896	23764	32900
13	2640	46778	64452
14	4008	103928	120368
15	6796	197380	243676
16	12308	399648	492512
17	23348	705660	1109520
18	45432		
19	88924		
20	177640		
21	354396		
22	707644		

Таблица А.6 – Количество выделенной памяти при запуске на четырёх узлах

Grade	Максимум выделения памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1268	8528	13540
9	1332	15680	23244
10	1434	33676	41672
11	1616	70524	62500
12	1968	98392	110248
13	2644	213096	193776
14	4040	440016	433000
15	6800	1008440	973532
16	12320	1982160	1917768
17	23352	3972420	3325100
18	45444		
19	88948		
20	177916		
21	354560		
22	707840		

Таблица А.7 – Количество выделенной памяти при запуске на четырёх узлах

Grade	Среднее выделение памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1265	4126	6176
9	1329	5935	9878
10	1426	13261	17300
11	1610	27344	29102
12	1944	43824	54028
13	2643	91107	100702
14	4021	190746	205094
15	6798	411465	439462
16	12313	804971	859541
17	23351	1597184	1681170
18	45437		
19	88935		
20	152824		
21	354484		
22	707736		

Таблица А.8 – Количество выделенной памяти при запуске на четырёх узлах

Grade	Суммарное выделение памяти, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	5060	16504	24704
9	5316	23740	39513
10	5702	53044	69200
11	6440	109374	116408
12	7776	175296	216112
13	10572	364426	402808
14	16084	762984	820376
15	27192	1645860	1757848
16	49252	3219884	3438164
17	93404	6388734	6724680
18	181748		
19	355740		
20	611296		
21	1417936		
22	2830944		

Таблица А.9 – Количество выделенной памяти при запуске на восьми узлах

Grade	Минимум выделеня памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1324	1220	1760
9	1348	1236	2984
10	1392	1228	4960
11	1480	1244	7348
12	1656	1240	14044
13	1964	2204	26960
14	2692	3380	44365
15	4052	6628	113336
16	6796	12388	

Таблица А.10 – Количество выделенной памяти при запуске на восьми узлах

Grade	Максимум выделения памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1324	16242	14068
9	1356	28396	28108
10	1412	56152	57032
11	1516	100340	105536
12	1728	223752	222684
13	2088	408201	433004
14	2956	898876	882272
15	4412	1799576	1793824
16	7608	3531672	

Таблица А.11 – Количество выделенной памяти при запуске на восьми узлах

Grade	Среднее выделение памяти на процесс, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	1324	2996	3820
9	1352	5022	6743
10	1399	7027	11934
11	1502	15071	21890
12	1689	29211	40152
13	2012	57386	86353
14	2749	116084	155274
15	4236	231888	311457
16	6948	452540	

Таблица А.12 – Количество выделенной памяти при запуске на восьми узлах

Grade	Суммарное выделение памяти, КБ		
	DGraphMark	Graph500 P2P	Graph500 RMA
8	10592	23968	30560
9	10816	40176	53944
10	11192	56216	95472
11	12016	120568	175120
12	13512	233688	321216
13	16096	459088	690824
14	21992	928672	1242192
15	33888	1855104	2491656
16	55584	3620320	
17	99584	6780216	
18	188064		

Таблица А.13 – Оценки бенчмарков при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	298385,94525	1806505,69758	2691082,26566	1793305,76033	6179809,05899
9	383360,04669	3070712,57590	2259384,06252	8980590,26869	9722619,79853
10	489294,65229	2713503,52363	1863338,27600	12333000,13209	30326335,71756
11	493167,77115	2871357,31314	1498038,29735	13007665,05912	27537505,00035
12	593537,51516	2861496,67791	1268319,93828	14015801,90414	27296713,69851
13	650259,99939	3123896,56951	1227451,33536	12534990,59008	28443744,72870
14	782614,00119	3097621,59769	895117,02567	11191983,39847	30586169,68332
15	713254,40875	3046791,45432	420978,65409	12206352,65135	31138816,98601
16	658455,25323	2664161,14861	261334,04157	11614947,97653	30269980,25455
17	689195,91230	2666693,65535	163390,95699	11548551,74895	29086070,25491
18	657779,14111	2621130,81772	80452,54740	11141685,14585	28334003,41515
19	691010,98349	2516090,97598	33230,73505	10034071,70385	27002795,17788
20	667808,90103	2952882,29357	16662,22269	10427294,68705	25546459,36442
21				10273989,75293	24174371,95244
22				10190246,06150	22220766,05940

Таблица А.14 – Время работы BFS при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	0,00882	0,00112	0,00173	0,0014	0,00038
9	0,00744	0,00193	0,00451	0,00057	0,00051
10	0,01691	0,00486	0,00858	0,00068	0,00032
11	0,03355	0,00558	0,01503	0,00193	0,00069
12	0,08743	0,01802	0,03648	0,00247	0,00152
13	0,13503	0,03100	0,05871	0,00790	0,00283
14	0,23881	0,08367	0,21148	0,01646	0,00434
15	0,38360	0,08488	0,64640	0,02247	0,00823
16	0,80731	0,22358	2,22207	0,04613	0,01750
17	1,55186	0,35086	6,56749	0,09835	0,07582
18	3,21051	0,78812	26,65324	0,23055	0,11080
19	6,09145	1,61539		0,41918	0,19122
20	12,63844	3,25900		0,83033	0,35682
21				1,66648	0,73597
22				3,40787	1,51788

Таблица А.15 – Медиана времени валидации при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени валидации, с		
	прямая валидация	обратная валидация	валидация Graph500
8	0,00199	0,00037	0,27807
9	0,00095	0,00101	0,32484
10	0,00095	0,00196	0,35720
11	0,00292	0,00153	0,33169
12	0,00307	0,00315	0,38384
13	0,00919	0,00586	0,48393
14	0,01857	0,00860	0,64058
15	0,02642	0,02611	1,14835
16	0,05386	0,03409	2,36727
17	0,15550	0,06814	4,66660
18	0,22404	0,17831	9,23556
19	0,52667	0,35384	21,85296
20	1,11959	0,69175	
21	2,18989	1,43603	
22	4,32683	2,98123	

Таблица А.16 – Оценки бенчмарков при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	210187,30038	1823378,17703	1642434,91243	4774838,57254	15561475,71014
9	173595,40000	2047416,18210	1633377,94105	5108495,14838	21105490,39803
10	192077,24721	2091407,77698	1308493,78758	7743912,18571	26010399,97578
11	207452,51902	2024614,83519	806254,34674	8510151,91777	33006472,97598
12	218752,21392	1943829,65317	831904,56675	7959631,28928	32179572,34184
13	180673,48641	1956112,91111	810767,96706	8016417,20137	29664378,03109
14	195215,05706	2071932,73329	446165,90613	8141727,09804	30822513,45879
15	206505,25880	2099499,19652	281893,43659	8538247,52392	32260695,97746
16	200813,88652	2193616,47706	193168,19708	8272554,13755	25192929,29431
17	204485,67362	2050280,41335	87521,67859	6626276,91242	25509130,80063
18	199949,16207	1957162,34189	44691,43407	7121958,63289	27539390,37735
19	202086,37162	1944863,03071	24291,41187	7894499,59585	30197583,15195
20	202177,30027	1969679,51945	13732,73866	8060468,49995	23865490,22637
21				7404108,47382	23031652,56419
22				7456789,38260	23985270,78524

Таблица А.17 – Время работы BFS при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	0,02398	0,00326	0,00610	0,00050	0,00015
9	0,02567	0,00341	0,00419	0,00167	0,00024
10	0,04457	0,00424	0,00905	0,00239	0,00032
11	0,12834	0,00852	0,02062	0,00313	0,00054
12	0,15909	0,02815	0,06571	0,00472	0,00104
13	0,36903	0,03576	0,11432	0,00824	0,00223
14	0,67557	0,09285	0,34176	0,02133	0,00488
15	1,41357	0,13013	1,06056	0,03807	0,00821
16	2,64922	0,29737	2,93019	0,06642	0,02099
17	5,27131	0,52359	12,11209	0,16702	0,04118
18	10,87428	1,11573	49,32592	0,29908	0,07627
19	20,88929	2,15884		0,53788	0,16176
20	41,87485	4,29118		1,08639	0,35321
21				2,28473	0,76136
22				4,68786	1,39961

Таблица А.18 – Медиана времени валидации при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени валидации, с		
	прямая валидация	обратная валидация	валидация Graph500
8	0,00120	0,00163	0,85362
9	0,00096	0,00179	0,86227
10	0,00165	0,00242	0,91085
11	0,00245	0,00214	1,01488
12	0,00329	0,00362	0,96437
13	0,00651	0,00658	1,31273
14	0,01699	0,01721	1,75016
15	0,03855	0,03486	2,89543
16	0,05333	0,06304	4,58613
17	0,17509	0,16421	
18	0,27017	0,38814	
19	0,50151	0,59782	
20	1,10190	1,22975	
21	2,62633	2,49245	
22	4,38683	5,19397	

Таблица А.19 – Оценки бенчмарков при запуске на восьми узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Оценка бенчмарка, TEPS			
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	867,62450	2503,47972	5288,56939	8559,88085
9	1497,82320	3958,71028	6414,88355	24392,09248
10	2064,07109	5982,30175	16077,58065	45589,10931
11	2865,21651	9039,32030	29503,43376	62849,50215
12	3348,40304	12161,71221	58677,11508	138255,83146
13	3516,48358	14940,24052	113220,86803	226952,03003
14	4463,46732	17853,50395	197650,38693	573799,56552
15		17326,14587	345438,98651	750935,72861
16		16973,06947	615133,25013	1619026,85294
17			1131788,40056	2721620,63626
18			1932716,79426	3959851,86185
19			2797150,26528	4724891,57512
20			3503109,50418	6214159,02940
21			4189226,42162	7631539,19144
22			4268279,27392	8379578,57156

Таблица А.20 – Время работы BFS при запуске на восьми узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени работы BFS, с			
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	2,43601	0,85134	0,42000	0,23960
9	2,94001	1,08794	0,64800	0,23926
10	4,03618	1,39126	0,51025	0,23939
11	6,25202	1,93064	0,59579	0,26330
12	10,36006	2,69935	0,67600	0,27133
13	18,87600	4,43952	0,61198	0,32742
14	32,99998	7,40360	0,72400	0,27958
15		15,54340	0,78000	0,35661
16		31,94987	0,85201	0,38242
17			1,04400	0,44443
18			1,08000	0,53932
19			1,60048	0,90280
20			2,39329	1,35502
21			4,20000	2,37636
22			8,03196	4,05684
23			14,15771	7,88672

Таблица А.21 – Медиана времени валидации при запуске на восьми узлах (GCC 4.7, MPICH 3.1, ОС Linux, ЦП Intel Core i3 2310M, ОЗУ 8 ГБ)

Grade	Медиана времени валидации, с	
	прямая валидация	валидация Graph500
8	0,80762	3,34112
9	0,80815	4,59551
10	0,74086	7,23590
11	0,98778	11,38390
12	0,95574	
13	0,86424	
14	1,26781	
15	1,12779	
16	1,05579	
17	1,19980	
18	1,67961	
19	2,05177	
20	3,29559	
21	5,35148	
22	9,41978	

Таблица А.22 – Оценки бенчмарков при запуске на двух узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	658282,97893	2441015,79767	1500425,25624	4655791,10678	22025473,31282
9	606077,37190	2824707,19895	1199795,31978	5753472,60013	27140393,65561
10	641854,18755	2760040,03277	934297,86730	5041781,12517	13256071,90123
11	528071,10215	2779801,65592	648511,08136	6199321,31132	23342213,56522
12	582210,55931	3112224,66593	586871,94305	7154552,49724	24167215,31071
13	620894,13897	2463770,13968	498593,17394	7117317,18350	22906492,24533
14	647889,72720	2790298,71431	365725,36852	5405024,12585	22466522,83972
15	645622,33799	2796081,78300	193712,37387	6032952,50932	18053785,22505
16	650808,47242	3038504,24758	106323,72168	6431077,23222	17664681,89893
17	643593,89554	2987139,75299	78164,54164	6591341,89055	19055660,79334
18	642302,81835	2913807,08661		6517164,44271	19319334,55350
19	482805,63877	2823226,67459		6453529,55559	18914661,33568
20		2854973,37276		6429431,97213	18133890,69418

Таблица А.23 – Время работы BFS при запуске на двух узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	0,00352	0,00097	0,00383	0,00044	0,00016
9	0,01288	0,00148	0,00349	0,00072	0,00023
10	0,01874	0,00309	0,01230	0,00176	0,00066
11	0,03120	0,00991	0,02556	0,00269	0,00074
12	0,05662	0,01612	0,05650	0,00460	0,00146
13	0,10604	0,02671	0,13357	0,01493	0,00304
14	0,20575	0,04729	0,37603	0,02428	0,00589
15	0,41361	0,09389	1,45710	0,04369	0,01453
16	0,80585	0,17317	5,02661	0,08177	0,03000
17	1,63955	0,35335	13,57140	0,15914	0,05512
18	3,27037	0,72050		0,32210	0,11005
19	9,67197	1,48652		0,65104	0,22245
20		2,93847		1,30502	0,46429

Таблица А.24 – Медиана времени валидации при запуске на двух узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Медиана времени валидации, с		
	прямая валидация	обратная валидация	валидация Graph500
8	0,00034	0,00016	0,27248
9	0,00095	0,00026	0,32947
10	0,00107	0,00065	0,35099
11	0,00218	0,00098	0,34098
12	0,00538	0,00347	0,39453
13	0,01057	0,00463	0,47450
14	0,02696	0,00988	0,68257
15	0,05508	0,02613	1,21462
16	0,10768	0,05035	2,25673
17	0,21290	0,10062	4,35773
18	0,42332	0,20223	
19	0,84487	0,41953	
20	1,91935	0,88026	

Таблица А.25 – Оценки бенчмарков при запуске на четырёх узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Оценка бенчмарка, TEPS			
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	1595,24939	3109,56976	5224,24884	21477,62857
9	2174,09539	3840,41312	13331,12118	26495,21630
10		6098,11083	22603,73778	62729,21325
11		7547,32288	48183,55095	125444,91432
12		8229,18167	95264,10516	210848,65307
13		9454,99482	157820,34648	457117,33002
14			330909,96433	799590,23980
15			524271,74951	1145259,47911
16			956438,89718	1823857,30362
17			1120095,11458	2503508,98667
18			1560178,45408	3016943,82079
19			2092645,51554	4209545,09364
20			2257093,53311	4087541,11747

Таблица А.26 – Время работы BFS при запуске на четырёх узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Медиана времени работы BFS, с			
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	1,29582	0,65862	0,40639	0,09936
9	1,90773	1,07869	0,32796	0,15460
10		1,39932	0,36687	0,13059
11		2,25059	0,41598	0,13061
12		4,00819	0,34835	0,16735
13		7,02317	0,48000	0,14337
14			0,56401	0,17444
15			0,50400	0,23959
16			0,58800	0,29943
17			0,96678	0,44330
18			1,36692	0,74204
19			2,02800	1,00742
20			3,72797	2,05411

Таблица А.27 – Медиана времени валидации при запуске на четырёх узлах (GCC 4.6, MPICH 3.0, ОС Linux, ЦП Intel Core 2 Duo E4400, ОЗУ 3 ГБ)

Grade	Медиана времени валидации, с	
	прямая валидация	валидация Graph500
8	0,25184	1,17262
9	0,21180	1,23405
10	0,24425	1,70205
11	0,26372	2,84214
12	0,18748	5,68731
13	0,27571	
14	0,29944	
15	0,31240	
16	0,37987	
17	0,65973	
18	1,09172	
19	1,87128	
20	3,38768	

Таблица А.28 – Оценки бенчмарков при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	151845,44159	1929238,53835	310711,66143	5834562,46697	20648881,23077
9	147089,18042	972151,94568	328562,36965	797950,26400	17244536,19473
10	275743,74229	928371,84786	433688,60098	1336109,98369	6050048,57472
11	211096,39713	691755,16327	408599,42287	1325779,81770	19879793,66052
12	367198,12518	1562179,08219	382078,67637	3505335,66630	6314242,22875
13	421165,54859	1597818,48535	320274,86773	3763156,80091	12531332,30503
14	534434,77147	1928416,30863	186714,35337	4729428,07765	10586172,43654
15	542238,29002	1896463,38377	117993,32846	5162038,53894	16034410,95164
16	554911,25155	1825410,54643	61890,21760	5496510,36659	12216151,12287
17	523966,10401	2620374,57382	1204928,40398	6083916,50557	14472018,79271
18	111928,26363	3112295,14203	805885,59827	2754950,15779	17077404,75547
19	90753,28539	425908,45090		634435,14103	17766152,20683

Таблица А.29 – Время работы BFS при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	0,10213	0,00497	0,03201	0,00145	0,00066
9	0,20355	0,03176	0,05976	0,02215	0,00101
10	0,14232	0,03921	0,09351	0,03193	0,01408
11	0,33289	0,09962	0,16425	0,05535	0,00342
12	0,38481	0,08415	0,37459	0,05053	0,02173
13	0,64141	0,20770	0,95650	0,07037	0,03089
14	0,99113	0,27344	3,06993	0,12119	0,04955
15	1,93762	0,63258	8,95508	0,20596	0,09093
16	4,17009	1,16246	34,87415	0,39438	0,17730
17	8,15353	1,67996	0,00180	0,70347	0,29270
18	0,01952	0,00079	0,02507	0,00114	0,00012
19	0,06146	0,00969		0,00740	0,00061

Таблица А.30 – Медиана времени валидации при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ)

Grade	Медиана времени валидации, с		
	прямая валидация	обратная валидация	валидация Graph500
8	0,00043	0,00016	0,36521
9	0,00106	0,00025	0,39584
10	0,00141	0,00060	0,42975
11	0,00446	0,00289	0,47419
12	0,01621	0,00742	0,52173
13	0,04954	0,01667	0,64585
14	0,03627	0,06564	0,81086
15	0,08001	0,08460	1,48731
16	0,14835	0,10118	38,80832
17	0,30207	0,18939	
18	0,42318	0,30597	
19	0,78604	0,54603	

Таблица А.31 – Оценки бенчмарков при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на одном гипервизоре)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	479,48980	6825,27914	12116,55480	67107,29117	143632,38177
9	440,58343	8355,96750	11434,33373	102359,83022	70519,70373
10		9977,11529	16995,45995	223222,44694	281431,89287
11		11800,46504	24383,16950	188632,35372	201623,90851
12		13639,49293	23704,47168	470645,88789	550917,14723
13		15183,01516	35154,75025	697411,45172	1106565,79824
14		15663,13222	60177,31203	688569,45743	960306,10504
15		19911,90747	46571,04270	895957,29745	1440776,63840
16			45738,99378	1144024,29653	1821352,62405
17				896253,21197	1846739,73083
18				1214401,90909	2116445,65346
19				1304503,44817	2203128,72992
20				1317835,25419	2094779,33403

Таблица А.32 – Время работы BFS при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на одном гипервизоре)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	4,43336	0,33243	0,22417	0,03395	0,01573
9	9,43208	0,53289	0,45023	0,06484	0,05905
10		0,90667	0,54214	0,03788	0,03837
11		1,73522	0,95912	0,08896	0,08467
12		2,37901	1,42026	0,09185	0,05975
13		4,36357	2,48515	0,11639	0,06016
14		8,64898	3,53027	0,19717	0,14380
15		15,89986	7,56780	0,35489	0,19016
16			17,89855	0,48808	0,33770
17				1,19022	0,59569
18				1,73211	1,03713
19				3,46501	2,10664
20				6,48876	4,10627

Таблица А.33– Медиана времени валидации при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на одном гипервизоре)

Grade	Медиана времени валидации, с	
	прямая валидация	валидация Graph500
8	0,01838	0,81254
9	0,03750	0,94896
10	0,05863	1,23975
11	0,05260	2,70345
12	0,07936	5,43861
13	0,09849	
14	0,18712	
15	0,25823	
16	0,52041	
17	1,13531	
18	2,07301	
19	3,55119	
20	7,52380	

Таблица А.34 – Оценки бенчмарков при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade				
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	38271,55004	28112,19631	94187,87930	38767,61635
9	48790,09078	32531,53137	138358,76253	260810,81484
10	65025,13852	43524,24743	234649,58252	344323,90712
11	47582,18530	38107,81329	268560,29457	564356,85443
12	114621,48391	67110,30582	426623,68572	697302,16220
13	99540,71682	110315,07013	363353,84037	980484,70636
14	126115,74582	114569,48149	808830,78496	223050,55024
15	88029,81179	84346,78519	774544,50838	580374,02600

Таблица А.35 – Время работы BFS при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade				
	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	0,06139	0,07484	0,02399	0,05461
9	0,09224	0,14297	0,03648	0,02229
10	0,13451	0,19173	0,03847	0,02422
11	0,40036	0,48011	0,06333	0,03348
12	0,36757	0,50809	0,08520	0,04767
13	1,11969	1,47857	0,31840	0,09020
14	1,21861	1,26417	0,18632	0,79815
15	3,82853	3,14560	0,45753	2,06022

Таблица А.36 – Медиана времени валидации при запуске на двух узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade	Медиана времени валидации, с	
	прямая валидация	валидация Graph500
8	0,01990	0,27070
9	0,02948	0,40406
10	0,03722	0,53960
11	0,08050	0,83657
12	0,12120	1,59436
13	0,47956	3,18642
14	0,24320	7,73373

Таблица А.37 – Оценки бенчмарков при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade	Оценка бенчмарка, TEPS				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	188,42703	12135,33272	7594,79606	34340,78226	77595,11655
9	154,49634	13801,17575	6205,34472	45930,19836	128334,39794
10	128,57528	27572,75891	3309,39988	73672,47739	108373,93192
11		30153,45796	19908,46346	38280,50417	201549,98969
12		46363,97396	10333,44952	317980,11099	218476,80808
13		55812,79120	80066,26420	99632,08567	284150,88699
14		80838,45448	42511,20311	233002,86333	540329,01771
15		41824,15712	45795,28721	796393,87585	75303,01986

Таблица А.38 – Время работы BFS при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade	Медиана времени работы BFS, с				
	DGraphMark RMA	DGraphMark P2P	Graph500 RMA	DGraphMark P2PNB	Graph500 P2P
8	11,01405	0,22688	0,34150	0,07285	0,02678
9	33,56977	0,33039	0,67459	0,12579	0,03310
10	65,20280	0,32724	2,81994	0,12766	0,08331
11		1,03843	1,58526	0,51477	0,10079
12		0,80111	6,05419	0,11238	0,15959
13		1,58620	1,12785	0,88395	0,23819
14		2,37219	6,67000	2,17829	0,29322
15		10,75659	5,73988	2,56500	4,02720

Таблица А.39 – Медиана времени валидации при запуске на четырёх узлах (GCC 4.7, MPICH 3.1, ОС Linux, VirtualBox 2 ядра, ОЗУ 2 ГБ, 2 виртуальные машины на гипервизорах, соединённых 100Мбит/с ЛВС)

Grade	Медиана времени валидации, с	
	прямая валидация	валидация Graph500
8	0,06057	0,47717
9	0,08461	0,57063
10	0,13187	0,68288
11	0,24887	1,76827
12	0,14028	3,41096
13	0,88351	6,36990

Приложение Б
(обязательное).
Фрагменты листинга программы

Файл манифеста сборки приложения (makefile)

```
# true of false to use or not of OpenMP in compilation
OPENMP = false

#true to enable building of graph500 bfs task runners
BUILD_GRAPH500_BFS = true

#false to make graph unoriented
IS_GRAPH_ORIENTED = true

#Type of used generator. Use KRONECKER or UNIFORM.
#Illegal type will produce failure on start.
GRAPH_GENERATOR_TYPE = UNIFORM

#Type of depth builder used in validator. Use BUFFERED of P2PNOBLOCK.
VALIDATOR_DEPTH_BUILDER_TYPE = BUFFERED

#compile flags
OPENMP_FLAG = -fopenmp
MPICPP = mpic++
CPPFLAGS = -Ofast -std=c++98 #-std=c++11
CPPFLAGS += -DGENERATOR_TYPE_$(GRAPH_GENERATOR_TYPE)
CPPFLAGS += -DDEPTH_BUILDER_TYPE_$(VALIDATOR_DEPTH_BUILDER_TYPE)

ifeq ($(OPENMP), true)
      CPPFLAGS += $(OPENMP_FLAG)
endif

ifeq ($(IS_GRAPH_ORIENTED), false)
      CPPFLAGS += -DGRAPH_IS_UNORIENTED
endif

#directories definition
SRC_DIR = src/
BIN_DIR = bin/
OBJ_DIR = $(BIN_DIR)obj/
BENCHMARK_DIR = benchmark/
CONTROLLER_DIR = controller/
GENERATOR_DIR = generator/
GRAPH_DIR = graph/
MPI_DIR = mpi/
TASK_DIR = task/search/
BFS_DIR = $(TASK_DIR)bfs/
VALIDATOR_DIR = $(TASK_DIR)validator/
UTIL_DIR = util/

#definition of path to object directories
OBJ_DIR_PATHS = $(addprefix $(OBJ_DIR), \
            $(BENCHMARK_DIR) $(BENCHMARK_DIR)search/ \
            $(CONTROLLER_DIR) $(CONTROLLER_DIR)search/ \
            $(GENERATOR_DIR) $(GRAPH_DIR) $(MPI_DIR) \
            $(TASK_DIR) $(BFS_DIR) $(VALIDATOR_DIR) $(UTIL_DIR) )
```

```
#Definitions of sources to compile
BENCHMARK = Benchmark search/SearchBenchmark
CONTROLLER = Controller search/SearchController
GENERATOR = RandomGenerator UniformGenerator KroneckerGenerator
GRAPH = Graph CSRGraph GraphDistributor
MPI = Communicable RMAWindow BufferedDataDistributor
TASK = ParentTree ParentTreeValidator SearchTask
BFS = BFSdgmark BFSGraph500P2P BFSGraph500RMA BFSTaskRMAFetch BFSTaskP2P
BFSTaskP2PNoBlock
VALIDATOR = DepthBuilder DepthBuilderBuffered DepthBuilderP2PNoBlock
UTIL = Statistics Random

#separated, because if use all, it is too long, error occurred in build.
FILES_LIST = $(addprefix $(BENCHMARK_DIR), $(BENCHMARK)) \
          $(addprefix $(CONTROLLER_DIR), $(CONTROLLER)) \
          $(addprefix $(GENERATOR_DIR), $(GENERATOR)) \
          $(addprefix $(GRAPH_DIR), $(GRAPH)) \
          $(addprefix $(MPI_DIR), $(MPI))
FILES_LIST += $(addprefix $(TASK_DIR), $(TASK)) \
          $(addprefix $(BFS_DIR), $(BFS)) \
          $(addprefix $(VALIDATOR_DIR), $(VALIDATOR)) \
          $(addprefix $(UTIL_DIR), $(UTIL)) \
          main_dgmark

#full sources and objects paths
SOURCES = $(addprefix $(SRC_DIR), $(addsuffix .cpp, $(FILES_LIST)))
OBJECTS = $(addprefix $(OBJ_DIR), $(addsuffix .o, $(FILES_LIST)))

#build targets
BUILD = dgmark dgmark_p2p dgmark_p2p_noblock dgmark_rma

ifeq ($(BUILD_GRAPH500_BFS), true)
      BUILD += graph500_p2p graph500_rma
endif

#build rules
.PHONY : all
all: $(BUILD)

# prepairing directories.
$(OBJ_DIR_PATHS):
      mkdir -p $@

# extended builds
dgmark dgmark_% graph500% : $(OBJ_DIR_PATHS) $(OBJECTS)
      rm -f $(OBJ_DIR)main_dgmark.o;
      $(MPICPP) $(CPPFLAGS) -DTASK_TYPE_$@ -c $(SRC_DIR)main_dgmark.cpp -o
$(OBJ_DIR)main_dgmark.o
      $(MPICPP) $(CPPFLAGS) $(OBJECTS) -o $(BIN_DIR)$@;

#building of sources
$(OBJ_DIR)%.o: $(SRC_DIR)%.cpp
      $(MPICPP) $(CPPFLAGS) -c $< -o $@

#cleaning binaries
.PHONY : clean
clean:
      rm -rf $(BIN_DIR)*
```

```
// Точка входа в приложение

int main(int argc, char** argv)
{
	Init();
	Intracomm *comm = &COMM_WORLD;

	vector<Task*> *tasks = new vector<Task*>();

	#ifdef TASK_TYPE_dgmark_p2p
		tasks->push_back(new BFSTaskP2P(comm));
	#elif TASK_TYPE_dgmark_p2p_noblock
		tasks->push_back(new BFSTaskP2PNoBlock(comm));
	#elif TASK_TYPE_dgmark_rma
		tasks->push_back(new BFSTaskRMAFetch(comm));
	#elif TASK_TYPE_graph500_p2p
		tasks->push_back(new BFSGraph500P2P(comm));
	#elif TASK_TYPE_graph500_rma
		tasks->push_back(new BFSGraph500Optimized(comm));
	#else
		tasks->push_back(new BFSTaskP2PNoBlock(comm));
	#endif

	SearchController *controller = new SearchController(comm, argc, argv);
	controller->run(tasks);
	controller->clean(tasks);

	Finalize();
	return 0;
}

#ifndef RANDOM_H
#define	RANDOM_H

namespace dgmark {
	class Random {
	public:
		Random(const Random& orig);
		virtual ~Random();
		static Random* getInstance(Intracomm *comm);

		/**
		 * Generates 64-bit random number.
		 * @return random number.
		 */
		uint64_t next();

		/**
		 * Generates 64-bit random number.
		 * @param min Min bourder.
		 * @param max Max bourder.
		 * @return random number.
		 */
		uint64_t next(uint64_t min, uint64_t max);

		/**
		 * Generates dandom double [0..1)
		 * @return random double.
		 */
		double nextDouble();
```

```
	private:
		Random(uint64_t seed);
		uint64_t seed;
		int randBitSize; //bit length of number generated by rand()
		void fillRandBitSize();
		static Random *instance;
	};

	Random::Random(uint64_t newSeed) : typeBitSize(64)
	{
		seed = time(0) + newSeed;
		srand(time(0) + newSeed);
		srand(rand() + newSeed);
		fillRandBitSize();
	}

	Random* Random::getInstance(Intracomm *comm)
	{
		if (!instance) {
			instance = new Random(comm->Get_rank());
		}
		return instance;
	}

	uint64_t Random::next()
	{
		uint64_t randomNumber = 0;
		int typeBitSize = 64;

		while (typeBitSize / randBitSize > 0) {
			randomNumber = randomNumber << randBitSize | rand();
			typeBitSize -= randBitSize;
		}

		randomNumber = randomNumber << typeBitSize
				| (rand() & (1 << typeBitSize - 1));
		return randomNumber;
	}

	uint64_t Random::next(uint64_t min, uint64_t max)
	{
		if (min > max) {
			uint64_t temp = max;
			max = min;
			min = temp;
		}

		uint64_t randomNumber = next();
		randomNumber = min + randomNumber % (max - min);
		return randomNumber;
	}

	double Random::nextDouble()
	{
		return((double) rand()) / RAND_MAX;
	}
}

#endif	/* RANDOM_H */
```

```
#ifndef     BENCHMARK_H
#define     BENCHMARK_H
namespace dgmark {
      class Benchmark : public Communicable {
      public:
            Benchmark(Intracomm *comm, Task *task, Validator *validator,
                  Graph *graph, int numStarts);
            Benchmark(const Benchmark& orig);
            virtual ~Benchmark();

            void run();
            double getTaskOpeningTime();
            virtual string getStatistics();
            string getName();

      protected:
            static const int statisticsPrecision = 5;

            Task *task;
            Validator *validator;
            Graph *graph;

            int numStarts;
            bool isSuccessfullyFinished;

            vector<double> *taskRunningTimes;
            vector<double> *validationTimes;
            vector<double> *marks;

            Log log;

            /**
             * Runs single benchmark task with index "startIndex".
             * @param startIndex Index of task to start.
             * @return true, if finished successfully.
             */
            virtual bool runSingleTask(int startIndex) = 0;

            /**
             * Prints statistics to output stream.
             * @param out Output stream.
             */
            virtual void printBenchmarkStatistics(stringstream &out);

            /**
             * Prints statistics of data to string.
             * @param data Data to get statistics from.
             * @param statName Name of ststistics field. Example: ".statName".
             * @param floatfieldFlag ios::scientific or ios::fixed.
             *        Format of printing.
             * @param precision Precision if ios:fixed format.
             * @return Statistics string.
             */
            string getStatistics(vector<double> *data,
                        string statName,
                        const ios::fmtflags floatfieldFlag = ios::scientific,
                        int precision = statisticsPrecision);

      };
```

```
	void Benchmark::run()
	{
		log << "Running " << getName() << " benchmark\n";
		isSuccessfullyFinished = true;
		task->open(graph);
		log << "\n";
		for (int startIndex = 0; startIndex < numStarts; ++startIndex) {
			bool isValid = runSingleTask(startIndex);
			if (!isValid) {
				isSuccessfullyFinished = false;
				break;
			}
		}

		comm->Barrier(); // wait, while all processes are finished task.
		task->close();
		comm->Barrier(); // wait for closing task for all processes.
	}

	string Benchmark::getStatistics()
	{
		stringstream out;
		out.precision(statisticsPrecision);
		out.setf(ios::fixed, ios::floatfield);
		string name = getName();

		out << "#\n";
		out << "#" << name << " benchmark\n";
		out << "#\n";

		if (isSuccessfullyFinished) {
			out << name << ".time.taskOpening = "
			    << task->getTaskOpeningTime() << "\n";
			printBenchmarkStatistics(out);
		} else {
			out << "\n#There were errors while running benchmark,
				no statistics available.\n";
		}
		return out.str();
	}
}

#ifndef SEARCHBENCHMARK_H
#define	SEARCHBENCHMARK_H

namespace dgmark {

	class SearchBenchmark : public Benchmark {
	public:
		SearchBenchmark(Intracomm *comm, SearchTask *task,
				Graph *graph, int numStarts);
		SearchBenchmark(const SearchBenchmark& orig);
		virtual ~SearchBenchmark();
		virtual bool runSingleTask(int startIndex);

	private:
		Vertex *startRoots;
		vector<double> *traversedEdges;
		Vertex* generateStartRoots(Vertex maxStartRoot);
		Vertex generateOutstandingRoot();
		virtual void printBenchmarkStatistics(stringstream &out);
	};
```

```
Vertex* SearchBenchmark::generateStartRoots(Vertex maxStartRoot)
{
      log << "\nGenerating roots... ";
      double startTime = Wtime();
      Vertex* startRoots = new Vertex[numStarts];
      Random *random = Random::getInstance(comm);

      if (rank == 0) {
            for (int i = 0; i < numStarts; ++i) {
                  startRoots[i] = random->next(0, maxStartRoot);
            }
      }

      comm->Bcast(startRoots, numStarts, VERTEX_TYPE, 0);
      double rootsGenerationTime = Wtime() - startTime;
      log << rootsGenerationTime << " s\n";
      return startRoots;
}

bool SearchBenchmark::runSingleTask(int startIndex)
{
      assert(task->getTaskType() == SEARCH);

      SearchTask *task = (SearchTask*) this->task;
      task->setRoot(startRoots[startIndex]);

      log << "Running search task (" << (startIndex + 1) << "/"
          << numStarts << ")\n";
      ParentTree *result = task->run();

      int restartCount = 0;
      const int maxRestarts = 5;
      while (result->getTraversedEdges() < 1
             && ++restartCount <= maxRestarts) {
            log << "Error running BFS: no edges going from root.\n";
            log << "Restart " << restartCount << "\n";
            delete result;
            task->setRoot(generateOutstandingRoot());
            result = task->run();
      }

      if (result->getTraversedEdges() < 1) {
            log << "Error running BFS: graph is too sparse\n";
      }

      bool isValid = validator->validate(result);

      if (isValid) {
            log << "Traversed edges: " <<result->getTraversedEdges();
            log << "Task mark: " << result->getMark() << " TEPS";
      }

      traversedEdges->push_back(result->getTraversedEdges());
      taskRunningTimes->push_back(result->getTaskRunTime());
      validationTimes->push_back(validator->getValidationTime());
      marks->push_back(result->getMark());

      delete result;
      return isValid;
}
}
#endif      /* SEARCHBENCHMARK_H */
```

```
#ifndef CONTROLLER_H
#define     CONTROLLER_H

namespace dgmark {

      class Controller : public Communicable {
      public:
            Controller(Intracomm *comm, int argc, char **argv);
            Controller(const Controller& orig);
            virtual ~Controller();

            /**
             * Creates benchmarks for tasks and runs them.
             * @param tasks Tasks.
             */
            virtual void run(vector<Task*> *tasks) = 0;

            /**
             * Cleans tasks array.
             * @param tasks Tasks.
             */
            virtual void clean(vector<Task*> *tasks) = 0;

            /**
             * Runs benchmarks.
             * @param benchmarks Bencharks.
             */
            void run(vector<Benchmark*> *benchmarks);

            /**
             * Cleans benchmarks array.
             * @param benchmarks Bencharks.
             */
            void clean(vector<Benchmark*> *benchmarks);

      protected:
            int grade;
            int density;
            int numStarts;

            Log log;
            static const int CONTROLLER_PRECISION = 5;

            /**
             * Returns task-specific statistics.
             * @return Statistics string.
             */
            virtual string getSpecificStatistics() = 0;

      private:
            string getInitialStatistics();
            void parseArguments(int argc, char** argv);
            void printResult(string stat);
      };
```

```
	void Controller::run(vector<Benchmark*> *benchmarks)
	{
		string statistics = getInitialStatistics()
					+ getSpecificStatistics();

		for (size_t bmark = 0; bmark < benchmarks->size(); ++bmark) {
			Benchmark *benchmark = benchmarks->at(bmark);
			benchmark->run();
			statistics += benchmark->getStatistics();
		}
		printResult(statistics);
	}

	void Controller::printResult(string stat)
	{
		log << stat;
		int mkdirResult = system("mkdir -p dgmarkStatistics");
		if (mkdirResult) { //if can't create file
			log << "\nCan't create statistics file. "
				<< "Mkdir error code " << mkdirResult << "\n";
			return;
		}
		time_t datetime = time(0);
		tm *date = localtime(&datetime);
		stringstream fileName;
		fileName << "dgmarkStatistics/dgmark_stat_"
			<< "g" << grade << "_d" << density << "_"
			<< (date->tm_year + 1900) << "-"
			<< (date->tm_mon + 1) << "-"
			<< date->tm_mday << "_"
			<< date->tm_hour << "-"
			<< date->tm_min << "-"
			<< date->tm_sec
			<< ".properties";
		ofstream fileOut;
		fileOut.open(fileName.str().c_str());
		fileOut << stat;
		fileOut.close();
	}

	void Controller::parseArguments(int argc, char** argv)
	{
		if (rank == 0) {
			grade = 8;
			density = 16;
			numStarts = 32;
			if (argc >= 2) {
				grade = atoi(argv[1]);
			}
			if (argc >= 3) {
				density = atoi(argv[2]);
			}
			if (argc >= 4) {
				numStarts = atoi(argv[3]);
			}
		}
		comm->Bcast(&grade, 1, INT, 0);
		comm->Bcast(&density, 1, INT, 0);
		comm->Bcast(&numStarts, 1, INT, 0);
	}
}
#endif	/* CONTROLLER_H */
```

```
#ifndef SEARCHCONTROLLER_H
#define     SEARCHCONTROLLER_H

namespace dgmark {

      class SearchController : public Controller {
      public:
            SearchController(Intracomm *comm, int argc, char **argv);
            SearchController(const SearchController& orig);
            virtual ~SearchController();

            virtual void run(vector<Task*> *tasks);
            virtual void clean(vector<Task*> *tasks);
      protected:
            virtual string getSpecificStatistics();
      private:
            GraphGenerator *generator;
            GraphGenerator* createGenerator();
      };

      GraphGenerator* SearchController::createGenerator()
      {
            Log log(comm);
            #ifdef GENERATOR_TYPE_UNIFORM
            {
                  log << "Using uniform gerenator\n";
                  return new UniformGenerator(comm);
            }
            #elif GENERATOR_TYPE_KRONECKER
            {
                  log << "Using Kronecker gerenator\n";
                  return new KroneckerGenerator(comm);
            }
            #else
            {
                  log << "\n\nCan't determine graph generator no create\n";
                  log << "Generator was not defined in makefile";
                  log << " (variable GRAPH_GENERATOR_TYPE)\n\n";
                  assert(false);
                  return 0;
            }
            #endif
      }

      void SearchController::run(vector<Task*> *tasks)
      {
            Graph *graph = generator->generate(grade, density);
            vector<Benchmark*> *benchmarks = new vector<Benchmark*>(0);
            for (auto taskIt = tasks->begin();taskIt < tasks->end();++taskIt){
                  Task *task = *taskIt;
                  assert(task->getTaskType() == SEARCH);
                  Benchmark* benchmark = new SearchBenchmark(comm,
                              (SearchTask*) task, graph, numStarts);
                  benchmarks->push_back(benchmark);
            }
            Controller::run(benchmarks);
            Controller::clean(benchmarks);
            graph->clear();
            delete graph;
      }
}
#endif      /* SEARCHCONTROLLER_H */
```

```
#ifndef GRAPHGENERATOR_H
#define     GRAPHGENERATOR_H

namespace dgmark {

      /**
       * Generates Graphs.<br />
       * Parameters for generation passes with constructor.
       */
      class GraphGenerator : public Communicable {
      public:
            GraphGenerator(Intracomm *comm);
            virtual ~GraphGenerator();

            /**
             * Generates graph with specified parameters.
             * @param grade Log[2] of total vertices count
             * @param density Average edges count, connected to vertex.
             * @return Graph.
             */
            virtual Graph* generate(int grade, int density) = 0;
            virtual double getGenerationTime() = 0;
            virtual double getDistributionTime() = 0;
      };
}
#endif      /* GRAPHGENERATOR_H */

#ifndef RANDOMGENERATOR_H
#define     RANDOMGENERATOR_H

namespace dgmark {

      class RandomGenerator : public GraphGenerator {
      public:
            RandomGenerator(Intracomm *comm);
            virtual ~RandomGenerator();

            Graph* generate(int grade, int density);
            double getGenerationTime();
            double getDistributionTime();

      protected:
            Random *random;
            Log log;

            /**
             * For implementing generators.
             * Here generated graph accorfing to algorithm of generator.
             * @param graph Graph to generate edges in.
             */
            virtual void generateInternal(Graph *graph) = 0;

      private:
            double generationTime;
            double distributionTime;

            void doGenerate(Graph* graph);
            void doDistribute(Graph* graph);
      };
```

```
	Graph* RandomGenerator::generate(int grade, int density)
	{
		Graph* graph = new Graph(comm, grade, density);
		doGenerate(graph);

		#ifdef GRAPH_IS_UNORIENTED
		doDistribute(graph);
		#endif

		return graph;
	}

	void RandomGenerator::doGenerate(Graph* graph)
	{
		log << "Generating graph... ";
		comm->Barrier();
		double startTime = Wtime();

		generateInternal(graph);
		graph->edges->resize(graph->edges->size(), 0);

		comm->Barrier();
		generationTime = Wtime() - startTime;
		log << generationTime << " s\n";
	}

	void RandomGenerator::doDistribute(Graph* graph)
	{
		log << "Distributing graph... ";
		comm->Barrier();
		double startTime = Wtime();

		//Here we make graph unoriended, by transfering edges between
nodes.
		GraphDistributor *distributor = new GraphDistributor(comm, graph);
		distributor->distribute();
		delete distributor;

		comm->Barrier();
		distributionTime = Wtime() - startTime;
		log << distributionTime << " s\n";
	}

	double RandomGenerator::getGenerationTime()
	{
		return generationTime;
	}

	double RandomGenerator::getDistributionTime()
	{
		return distributionTime;
	}
}

#endif	/* RANDOMGENERATOR_H */
```

```
#ifndef KRONECKERGENERATOR_H
#define     KRONECKERGENERATOR_H

namespace dgmark {

      class KroneckerGenerator : public RandomGenerator {
      public:
            KroneckerGenerator(Intracomm *comm);
            virtual ~KroneckerGenerator();

      protected:
            /**
             * Adds edges with kronecker algorithm 2x2. Equals to R-MAT.
             * @param graph Graph to add in.
             * @param localVertex local vertex to start from.
             * @param numEdges num edges to add from vertex.
             */
            virtual void generateInternal(Graph *graph);

      private:
            Edge* generateEdge(Graph *graph);

            void moveBorder(Vertex &left, Vertex &right, int dest);
            Vertex revert(Vertex source, int grade);
            void getKroneckerDest(int &fromDest, int &toDest);
      };

      /**
       * This map represents probability to connect vertices with.
       * Definition: vertice is strong, if it is in first half of table.
       * Definition: vertice is weak, if it is in second half of table.
       * Algorithm to connect is binary search.
       * Each time we have to decide, if our vertice is weak or strong,
       * and with which kind to connect.
       */
      static const double kronecker2x2Probability[2][2] = {
            {0.57, 0.19},
            {0.19, 0.05}
      };

      static const double KPStrongToStrong = kronecker2x2Probability[0][0];
      static const double KPStrongToWeak = KPStrongToStrong
                                          + kronecker2x2Probability[0][1];
      static const double KPWeakToStrong = KPStrongToWeak
                                          + kronecker2x2Probability[1][0];
      static const double KPWeakToWeak = KPStrongToWeak
                                          + kronecker2x2Probability[1][1];
      Vertex KroneckerGenerator::revert(Vertex source, int grade)
      {
            Vertex result = 0;
            for (int i = 0; i < grade; ++i) {
                  //log << result << " | " << source << " reverting\n";
                  result = result << 1 | source & 1;
                  source = source >> 1;
            }
            return result;
      }
```

```
	void KroneckerGenerator::generateInternal(Graph* graph)
	{
		GraphDistributor *distributor = new GraphDistributor(comm, graph);
		distributor->open();
		// create numLocalVertex * grade edges
		for (Vertex local = 0; local < graph->numLocalVertex; ++local) {
			for (int i = 0; i < graph->density; ++i) {
				Edge * edge = generateEdge(graph);
				while (edge->from == edge->to) {
					delete edge;
					edge = generateEdge(graph);
				}
				distributor->sendEdge(edge->from, edge->to);
				delete edge;
			}
		}
		distributor->close();
		delete distributor;
	}

	Edge* KroneckerGenerator::generateEdge(Graph *graph)
	{
		Vertex fromL = 0;
		Vertex fromR = graph->numGlobalVertex - 1;
		Vertex toL = 0;
		Vertex toR = graph->numGlobalVertex - 1;
		int toDest, fromDest;

		while (fromR - fromL != 1 && toR - toL != 1) {
			getKroneckerDest(fromDest, toDest);
			moveBorder(fromL, fromR, fromDest);
			moveBorder(toL, toR, toDest);
		}

		getKroneckerDest(fromDest, toDest);
		fromL += fromDest;
		toL += toDest;
		//reverting is important in balancing reasons
		return new Edge(revert(fromL, graph->grade),
				revert(toL, graph->grade));
	}

	void KroneckerGenerator::getKroneckerDest(int &fromDest, int &toDest)
	{
		double prob = random->nextDouble();
		if (prob < KPStrongToStrong) {
			fromDest = 0;
			toDest = 0;
		} else if (prob < KPStrongToWeak) {
			fromDest = 0;
			toDest = 1;
		} else if (prob < KPWeakToStrong) {
			fromDest = 1;
			toDest = 0;
		} else {
			fromDest = 1;
			toDest = 1;
		}
	}
}

#endif	/* KRONECKERGENERATOR_H */
```

```
#ifndef UNIFORMGENERATOR_H
#define     UNIFORMGENERATOR_H

namespace dgmark {

      class UniformGenerator : public RandomGenerator {
      public:
            UniformGenerator(Intracomm *comm);
            virtual ~UniformGenerator();

      protected:
            void generateInternal(Graph *graph);

            /**
             * Adds edges evenly with equal probabilities.
             * @param graph Graph to add in.
             * @param localVertex local vertex to start from.
             * @param numEdges num edges to add from vertex.
             */
            virtual void addEdgeFromVertex(Graph *graph, Vertex localVertex,
                                           size_t numEdges);
      };

      void UniformGenerator::generateInternal(Graph *graph)
      {
            for (Vertex vertex = 0; vertex < graph->numLocalVertex; ++vertex){
                  addEdgeFromVertex(graph, vertex, graph->density);
            }
      }

      void UniformGenerator::addEdgeFromVertex(Graph *graph,
                              Vertex localVertex, size_t numEdges)
      {
            vector<Edge *> * const edges = graph->edges;
            Vertex globalVertexFrom = graph->vertexToGlobal(localVertex);
            for (int edgeIndex = 0; edgeIndex < numEdges; ++edgeIndex) {
                  uint64_t rankTo = random->next(0, size);
                  Vertex localVertexTo =
                         random->next(0, graph->numGlobalVertex);
                  Vertex globalVertexTo =
                         graph->vertexToGlobal(rankTo, localVertexTo);

                  //prevent self-loops
                  if (globalVertexFrom != globalVertexTo) {
                        edges->push_back(
                               new Edge(globalVertexFrom, globalVertexTo));
                  } else {
                        --newEdgeIndex;
                        continue;
                  }
            }
      }
}

#endif      /* UNIFORMGENERATOR_H */
```

```
#ifndef CSRGRAPH_H
#define     CSRGRAPH_H

namespace dgmark {

      /*
       * Compressed Sparse Row Graph.
       * Contains vector of edges, sorted by v.form, and then by v.to.
       * Provides start and end indices of an output edges interval
       * for any local vertex.
       */
      class CSRGraph : public Graph {
      public:
            CSRGraph(Graph *graph);
            CSRGraph(const CSRGraph& orig);
            virtual ~CSRGraph();

            size_t getStartIndex(Vertex v) const;
            size_t getEndIndex(Vertex v) const;

      private:
            /**
             * Builds CSR graph from existing.
             * First, sorts edges by source,
             * then builds an array of start index for each vertex.
             */
            void buildCSR();
            Vertex *startIndex;
      };

      void CSRGraph::buildCSR()
      {
            //This kind of sort gives better performance.
            std::stable_sort(edges->begin(), edges->end(), compareEdge);
            Vertex prev = -1;
            for (size_t edgeIndex = 0;edgeIndex < edges->size(); ++edgeIndex){
                  const Edge * const currEdge = edges->at(edgeIndex);
                  const Vertex localVertex = vertexToLocal(currEdge->from);

                  if (previousVertex == localVertex)  continue;

                  //filling skipped edges with index of current.
                  //This makes they endIndex - startIndex == 0
                  for (Vertex skip = prev + 1; skip < localVertex; ++skip) {
                        startIndex[skip] = currEdgeIndex;
                  }

                  startIndex[localVertex] = currEdgeIndex;
                  prev = localVertex;
            }

            for (Vertex skip = prev + 1; skip < numLocalVertex; ++skip) {
                  startIndex[skip] = edges->size();
            }

            startIndex[numLocalVertex] = edges->size();
      }
}

#endif      /* CSRGRAPH_H */
```

```
#ifndef GRAPHDISTRIBUTOR_H
#define     GRAPHDISTRIBUTOR_H

namespace dgmark {

      /**
       * Universal class to distribute graph edges.
       * 2 usages:
       * 1) clone all edges and distribute them
       * 2) open -> send required edges to graph -> close.
       */
      class GraphDistributor : public BufferedDataDistributor {
      public:
            GraphDistributor(Intracomm* comm, Graph *graph);
            virtual ~GraphDistributor();

            /**
             * Distributes graph.
             * Each edge (myRank|localNode -> hisRank|hisLocal)
             * sends to hisRank as (hisRank|hisLocal -> myRank|localNode).
             *
             * @param graph Graph to distribute.
             */
            void distribute();

            /**
             * Transfers edge to "edge->to" referenced node.
             * Swaps "to" and "from" before sending.
             * @param edge Edge
             */
            void sendEdge(Vertex from, Vertex to);

            /**
             * Starts communication.
             */
            void open();

            /**
             * Ends communication.
             */
            void close();

      protected:
            virtual void processRecvData(size_t countToRead);

      private:
            static const size_t ELEMENT_SIZE = 2;
            static const size_t BUFFERED_ELEMENTS = 256;

            const Graph * const graph;
            vector<Edge*> * const edges;

            /**
             * Distributes edges to nodes.
             */
            void distributeEdges();

            void sendEdgeExternal(Vertex from, Vertex to, int toRank);

            bool isOpen;
            Log log;
      };
```

```
	void GraphDistributor::distribute()
	{
		if (isOpen) {
			log << "\nGraph distribution usage mixed up!\n";
			assert(false);
		}

		open();
		distributeEdges();
		close();

		size_t numEdges = edges->size();
		edges->resize(numEdges, 0); //Shrink size.
	}

	void GraphDistributor::distributeEdges()
	{
		const size_t initialCount = edges->size();
		for (size_t edgeIndex = 0; edgeIndex < initialCount; ++edgeIndex){
			const Edge * const edge = edges->at(edgeIndex);
			sendEdge(edge->to, edge->from);
		}
	}

	void GraphDistributor::sendEdge(Vertex from, Vertex to)
	{
		const int toRank = graph->vertexRank(from);
		if (toRank == rank) {
			edges->push_back(new Edge(from, to));
		} else {
			sendEdgeExternal(from, to, toRank);
		}
		probeSynchData();
	}

void GraphDistributor::sendEdgeExternal(Vertex from, Vertex to, int toRank)
	{
		while (isSendRequestActive[toRank]) {
			probeSynchData();
		}

		size_t &currCount = countToSend[toRank];
		Vertex *&currBuffer = sendBuffer[toRank];
		currBuffer[currCount] = from;
		currBuffer[currCount + 1] = to;
		currCount += elementSize;

		if (currCount == sendPackageSize) {
			sendData(toRank);
		}
	}

	void GraphDistributor::processRecvData(size_t countToRead)
	{
		for (size_t index = 0; index < countToRead; index += elementSize){
			const Vertex from = recvBuffer[index];
			const Vertex to = recvBuffer[index + 1];
			edges->push_back(new Edge(from, to));
		}
	}
}
#endif	/* GRAPHDISTRIBUTOR_H */
```

```
#ifndef     BUFFEREDDATADISTRIBUTOR_H
#define     BUFFEREDDATADISTRIBUTOR_H

namespace dgmark {

      class BufferedDataDistributor : public Communicable {
      public:
            BufferedDataDistributor(Intracomm *comm,
                  size_t elementSize, size_t bufferedElementsCount);
            virtual ~BufferedDataDistributor();

      protected:
            virtual void processRecvData(size_t countToRead) = 0;

            static const int DISTRIBUTION_TAG = 246;
            static const int END_TAG = 643;

            const size_t sendPackageSize;
            const size_t elementSize;

            Vertex **sendBuffer;
            Vertex *countToSend;
            Vertex *recvBuffer;

            Request *sendRequest;
            Request recvRequest;

            bool *isSendRequestActive;
            bool isRecvRequestActive;

            void sendData(int toRank);
            void probeSynchData();
            void probeRecv();

            void prepareBuffers();
            void flushBuffers();
            void updateRequestsActivity();
            void waitForOthersToEnd();
      private:
            int countEnded;
      };

      void BufferedDataDistributor::sendData(int toRank)
      {
            while (isSendRequestActive[toRank]) {
                  probeSynchData();
            }

            sendRequest[toRank] = comm->Isend(
                  &sendBuffer[toRank][0], countToSend[toRank],
                  VERTEX_TYPE, toRank, DISTRIBUTION_TAG);

            countToSend[toRank] = 0;
            isSendRequestActive[toRank] = true;
      }
```

```
	void BufferedDataDistributor::probeSynchData()
	{
		Status status;
		if (isRecvRequestActive && recvRequest.Test(status)) {
			isRecvRequestActive = false;
			const size_t dataCount = status.Get_count(VERTEX_TYPE);
			if (dataCount > 0) {
				processRecvData(dataCount);
			} else {
				++countEnded;
			}
		}

		probeRecv();
		updateRequestsActivity();
	}

	void BufferedDataDistributor::probeRecv()
	{
		if (!isRecvRequestActive) {
			isRecvRequestActive = true;
			recvRequest = comm->Irecv(
				recvBuffer, sendPackageSize,
				VERTEX_TYPE, ANY_SOURCE, DISTRIBUTION_TAG);
		}
	}

	void BufferedDataDistributor::flushBuffers()
	{
		for (int node = 0; node < size; ++ node) {
			if (node == rank) {
				continue;
			}
			//send all unsent data
			if (!isSendRequestActive[node] && countToSend[node] > 0) {
				sendData(reqIndex);
			}
			//sending empty array in end of communication.
			sendData(node);
			while (isSendRequestActive[node]) {
				probeSynchData();
			}
		}
	}

	void BufferedDataDistributor::waitForOthersToEnd()
	{
		// tell all processes, that you had been stopped;
		++countEnded;
		while (countEnded < size) {
			probeSynchData();
		}
		probeSynchData();

		if (isRecvRequestActive) {
			recvRequest.Cancel();
		}
	}
}

#endif	/* BUFFEREDDATADISTRIBUTOR_H */
```

```
#ifndef     RMAWINDOW_H
#define     RMAWINDOW_H

namespace dgmark {

      template<class T>
      class RMAWindow : public Communicable {
      public:
            RMAWindow(Intracomm *comm, size_t size, Datatype dataType);
            virtual ~RMAWindow();
            T* getData();
            size_t getDataSize();

            /** Cleans data, stored in window. */
            void clean();

            /**
             * Opens fence synchronization
             * @param assertType: MODE_NOPUT if no data putted.
             */
            void fenceOpen(int assertType = 0);

            /**
             * Opens fence synchronization
             * @param assertType: MODE_NOSTORE if no data was stored.
             */
            void fenceClose(int assertType = 0);

            /**
             * Retrieves data from window.
             * @param dataToGet pointer to allocated place to store data.
             * @param length length of data.
             * @param tgtRank rank of node, where data lies.
             * @param shift shift in target's data.
             */
            void get(T* dataToGet, size_t length, int tgtRank, size_t shift);

            /**
             * Puts data in target's storage.
             * @param dataToPut pointer to allocated place of data to put.
             * @param length length of data.
             * @param tgtRank rank of node, where data lies.
             * @param shift shift in target's data.
             */
            void put(T* dataToPut, size_t length, int tgtRank, size_t shift);

            /**
             * Accumulates data in target's storege.
             * @param dataToAcc pointer to allocated data to accumulate.
             * @param length length of data.
             * @param tgtRank rank of node, where data lies.
             * @param shift shift in target's data.
             * @param operation operation to accumulate with
             */
            void accumulate(T* dataToAcc, size_t length, int tgtRank,
                            size_t shift, const Op &operation);
      private:
            T *data; //don't freed here
            const size_t dataSize;
            const Datatype dataType;
            Win *win; //freed in ~RMAWindow
      };
```

```
	template<class T>
	void RMAWindow<T>::fenceOpen(int assertType)
	{
		win->Fence(MODE_NOPRECEDE | assertType);
	}

	template<class T>
	void RMAWindow<T>::fenceClose(int assertType)
	{
		win->Fence(MODE_NOSUCCEED | assertType);
	}

	template<class T>
	void RMAWindow<T>::get(T* dataToGet, size_t length,
				int tgtRank, size_t shift)
	{
		win->Get(dataToGet, length, dataType, tgtRank,
			shift, dataLength, dataType);
	}

	template<class T>
	void RMAWindow<T>::put(T* dataToPut, size_t length,
				int tgtRank, size_t shift)
	{
		win->Put(dataToPut, length, dataType, tgtRank,
			shift, dataLength, dataType);
	}

	template<class T>
	void RMAWindow<T>::accumulate(T* dataToAcc, size_t length,
				int targetRank, size_t shift, const Op &operation)
	{
		win->Accumulate(dataToAcc, length, dataType, tgtRank,
				shift, dataLength, dataType, operation);
	}
}

#endif	/* RMAWINDOW_H */
```

```
#ifndef     STATISTICS_H
#define     STATISTICS_H

namespace dgmark {
      using namespace std;

      class Statistics {
      public:
            Statistics(vector<double>* data);
            Statistics(const Statistics& orig);
            virtual ~Statistics();

      private:
            double mean;
            double stdDeviation;
            double relStdDeviation;

            double minimum;
            double firstQuartile;
            double median;
            double thirdQuartile;
            double maximum;
      };

      Statistics::Statistics(vector<double>* data)
      {
            size_t dataSize = data->size();

            mean = 0;
            for (vector<double>::iterator it = data->begin();
                 it < data->end(); ++it) {
                  mean += *it;
            }
            mean /= dataSize;

            stdDeviation = 0;
            for (vector<double>::iterator it = data->begin();
                 it < data->end(); ++it) {
                  stdDeviation += pow(*it - mean, 2.);
            }
            stdDeviation = sqrt(stdDeviation / (dataSize - 1));

            relStdDeviation = stdDeviation / mean;

            vector<double> *sortedData = new vector<double>();
            sortedData->insert(sortedData->begin(), data->begin(),
                               data->end());

            sort(sortedData->begin(), sortedData->end());

            size_t quartile = dataSize / 4 < 1 ? 1 : dataSize / 4;
            minimum = sortedData->front();
            firstQuartile = sortedData->at(dataSize / 4);
            median = sortedData->at(dataSize / 2);
            thirdQuartile = sortedData->at(dataSize - quartile);
            maximum = sortedData->back();

            delete sortedData;
      }

}
#endif      /* STATISTICS_H */
```

```
#ifndef RESULT_H
#define     RESULT_H

namespace dgmark {

      enum TaskType {
            STUB, SEARCH
      };

      /**
       * Shows, that object can be classified to some task type.
       */
      class Classifieble {
      public:

            virtual ~Classifieble()
            {
            }

            /**
             * @return Task type, in which  this object is classified.
             */
            virtual TaskType getTaskType() = 0;
      };

      /**
       * Represents result of the task.
       */
      class Result : public Classifieble, public Communicable {
      public:

            Result(Intracomm *comm) : Communicable(comm)
            {
            }

            virtual ~Result()
            {
            }

            /**
             * @return performance mark of the result.
             */
            virtual double getMark() = 0;

            /**
             * @return Time of running task for this result.
             */
            virtual double getTaskRunTime() = 0;
      };
}

#endif      /* RESULT_H */
```

```
#ifndef     TASK_H
#define     TASK_H

#include <string>
#include "../graph/Graph.h"
#include "Result.h"

namespace dgmark {

      /**
       * Represents the task.
       */
      class Task : public Classifieble, public Communicable {
      public:

            Task(Intracomm *comm) : Communicable(comm)
            {
            }

            virtual ~Task()
            {
            }

            /**
             * Opens task for graph.
             * @param graph Graph to run task on.
             */
            virtual void open(Graph *graph) = 0;

            /**
             * @return Time of opening current or last task.
             */
            virtual double getTaskOpeningTime() = 0;

            /**
             * Run task on graph. Be sure to initialize all parameters before
run.
             * @return
             */
            virtual Result* run() = 0;

            /**
             * Close task for graph.
             */
            virtual void close() = 0;

            /**
             * Returns name of the task, used in benchmark. No spaces allowed!
             * @return name of the task.
             */
            virtual string getName() = 0;
      };

}

#endif      /* TASK_H */
```

```
#ifndef     VALIDATOR_H
#define     VALIDATOR_H

#include "Result.h"

namespace dgmark {

      /**
       * Validates result of the task.
       */
      class Validator : public Classifieble, public Communicable {
      public:

            Validator(Intracomm *comm) : Communicable(comm)
            {
            }

            virtual ~Validator()
            {
            }

            /**
             * Validates result.
             * @param result result of task.
             * @return true, if result is valid, false otherwise.
             */
            virtual bool validate(Result *result) = 0;

            /**
             * @return Validation time.
             */
            virtual double getValidationTime() = 0;
      };
}

#endif      /* VALIDATOR_H */
```

```
#ifndef PARENTTREE_H
#define     PARENTTREE_H

namespace dgmark {

      class ParentTree : public Result {
      public:
            /**
             * Creates parent tree (result for tree-makers).
             * @param comm Communacator to work with.
             * @param root global root vertex.
             * @param parent Array of parents to verticies.
             * @param graph initial graph os result.
             * @param duration Duration of tree-making task in seconds
             */
            ParentTree(Intracomm *comm, Vertex root, Vertex *parent, const
CSRGraph *graph, double duration);
            ParentTree(const ParentTree& orig);
            virtual ~ParentTree();

            /**
             * Traversed egdes per second is a mark of this kind of task.
             * TEPS.
             * @return mark for task result.
             */virtual
            double getMark();
            double getTaskRunTime();
            TaskType getTaskType();
            Vertex getRoot();
            const Vertex* getParent();
            size_t getParentSize();
            const CSRGraph* getInitialGraph();
            double getTraversedEdges();
      private:
            Vertex root;
            Vertex * const parent;
            const CSRGraph * const graph;
            double taskRunTime;
            size_t traversedEdges;

            double calculateTraversedEdges(const Vertex * const parent, const
CSRGraph * const graph);
      };

      double ParentTree::calculateTraversedEdges(const Vertex * const parent,
                                    const CSRGraph * const graph)
      {
            double realTraversedEdges = 0;
            for (Vertex local = 0; local < graph->numLocalVertex; ++ local) {
                  size_t traversedMax = graph->getEndIndex(localVertex)
                                        - graph->getStartIndex(localVertex);
                  realTraversedEdges
                        += parent[localVertex] != graph->numGlobalVertex
                        ? traversedMax : 0;
            }

            comm->Allreduce(IN_PLACE, &realTraversedEdges, 1, DOUBLE, SUM);
            return realTraversedEdges;
      }
}

#endif      /* PARENTTREE_H */
```

```
#ifndef SEARCHTASK_H
#define     SEARCHTASK_H

#include "ParentTree.h"
#include "../Task.h"
#include "../../graph/CSRGraph.h"
#include "../../util/Log.h"

namespace dgmark {

      class SearchTask : public Task {
      public:
            SearchTask(Intracomm *comm);
            SearchTask(const SearchTask& orig);
            virtual ~SearchTask();

            virtual TaskType getTaskType();

            virtual void open(Graph *newGraph);
            virtual double getTaskOpeningTime();
            virtual ParentTree *run() = 0;
            virtual void close();

            void setRoot(Vertex newRoot);

      protected:
            const CSRGraph * graph;
            Vertex root;
            Log log;
      private:
            double taskOpeningTime;
      protected:
            Vertex numLocalVertex;
      };


      void SearchTask::open(Graph *newGraph)
      {
            log << "Opening task... ";
            comm->Barrier();
            double startTime = Wtime();

            graph = new CSRGraph(newGraph);

            comm->Barrier();
            taskOpeningTime = Wtime() - startTime;
            log << taskOpeningTime << " s\n";

            numLocalVertex = graph->numLocalVertex;
      }

      void SearchTask::close()
      {
            comm->Barrier();
            log << "Closing task... \n";
            delete graph;
            comm->Barrier();
      }

}

#endif      /* SEARCHTASK_H */
```

```
#ifndef     DEPTHBUILDER_H
#define     DEPTHBUILDER_H

namespace dgmark {

      class DepthBuilder {
      public:

            DepthBuilder(const Graph *graph);
            virtual ~DepthBuilder();

            /**
             * Builds depth. Do not clean array, it is reusable.
             * Deletes with deletion of builder.
             * @param parentTree Source of parent tree.
             * @return Depths array of verticies. 0 if failed to build array.
             */
            Vertex* buildDepth(ParentTree *parentTree);

      protected:
            static const int SYNCH_END_TAG = 24067;
            static const int LOCAL_SEND_TAG = 4670;
            static const int DEPTH_SEND_TAG = 3350;

            Vertex *depth;
            const Graph * const graph;
            const Vertex *parent;
            const CSRGraph *csrGraph;

            bool isNextStepRequired;

            virtual void buildNextStep() = 0;
            virtual void prepare(Vertex root);
      };

      Vertex* DepthBuilder::buildDepth(ParentTree *parentTree)
      {
            prepare(parentTree->getRoot());
            parent = parentTree->getParent();
            csrGraph = parentTree->getInitialGraph();

            while (isNextStepRequired) {
                  buildNextStep();
            }

            return depth;
      }

      void DepthBuilder::prepare(Vertex root)
      {
            isNextStepRequired = true;

            for (Vertex local = 0; local < graph->numLocalVertex; ++ local) {
                  depth[local] = graph->numGlobalVertex;
            }
      }
}

#endif      /* DEPTHBUILDER_H */
```

```
#ifndef     DEPTHBUILDERBUFFERED_H
#define     DEPTHBUILDERBUFFERED_H

namespace dgmark {

      class DepthBuilderBuffered :  public BufferedDataDistributor,
                                    public DepthBuilder {
      public:
            DepthBuilderBuffered(Intracomm *comm, Graph *graph);
            DepthBuilderBuffered(const DepthBuilderBuffered& orig);
            virtual ~DepthBuilderBuffered();

      protected:
            virtual void processRecvData(size_t countToRead);
            virtual void buildNextStep();
            virtual void prepare(Vertex root);

      private:
            static const size_t ELEMENT_SIZE = 3;
            static const size_t BUFFERED_ELEMENTS = 256;

            short *vertexState;
            static const short stateInitial = 0;
            static const short stateJustFilled = 1;
            static const short stateSent = 2;

            void distributeVertexDepth(Vertex localVertex);
            void updateDepth(Vertex parentGlobal,
                             Vertex currParent, Vertex parentDepth);
      };

      void DepthBuilderBuffered::prepare(Vertex root)
      {
            DepthBuilder::prepare(root);
            for (Vertex local = 0; local < graph->numLocalVertex; ++ local) {
                  vertexState[local] = stateInitial;
            }
            if (graph->vertexRank(root) == rank) {
                  Vertex rootLocal = graph->vertexToLocal(root);
                  depth[rootLocal] = 0;
                  vertexState[rootLocal] = stateJustFilled;
            }
      }

      void DepthBuilderBuffered::buildNextStep()
      {
            prepareBuffers();
            isNextStepRequired = false;

            for (Vertex local = 0; local < graph->numLocalVertex; ++ local) {
                  if (vertexState[local] == stateJustFilled) {
                        distributeVertexDepth(localVertex);
                        vertexState[localVertex] = stateSent;
                  }
                  probeSynchData();
            }

            flushBuffers();
            waitForOthersToEnd();
            comm->Allreduce(IN_PLACE, &isNextStepRequired, 1, SHORT, LOR);
      }
```

```
	void DepthBuilderBuffered::distributeVertexDepth(Vertex localVertex)
	{
		const size_t startIndex = csrGraph->getStartIndex(localVertex);
		const size_t endIndex = csrGraph->getEndIndex(localVertex);
		for (size_t childIdx = startIndex; childIdx<endIndex; ++childIdx){
			Vertex child = csrGraph->edges->at(childIndex)->to;
			Vertex childLocal = csrGraph->vertexToLocal(child);
			Vertex childRank = csrGraph->vertexRank(child);
			Vertex currGlobal = csrGraph->vertexToGlobal(localVertex);
			if (childRank == rank) {
			    updateDepth(currGlobal,childLocal,depth[localVertex]+1);
			    continue;
			} else {
				while (isSendRequestActive[childRank]) {
					probeSynchData();
				}

				size_t &currCount = countToSend[childRank];
				Vertex *&currBuffer = sendBuffer[childRank];

				currBuffer[currCount] = currGlobal;
				currBuffer[currCount + 1] = childLocal;
				currBuffer[currCount + 2] = depth[localVertex] + 1;
				currCount += elementSize;

				if (currCount == sendPackageSize) {
					sendData(childRank);
				}
			}
		}
	}

	void DepthBuilderBuffered::processRecvData(size_t countToRead)
	{
		for (size_t index = 0; index < countToRead; index += elementSize){
			const Vertex parentGlobal = recvBuffer[index];
			const Vertex localChild = recvBuffer[index + 1];
			const Vertex newDepth = recvBuffer[index + 2];
			updateDepth(parentGlobal, localChild, newDepth);
		}
	}

	void DepthBuilderBuffered::updateDepth(Vertex parentGlobal,
	                            Vertex localVertex, Vertex newDepth)
	{
		if (parent[localVertex] != parentGlobal) {
			return;
		}

		Vertex &currDepth = depth[localVertex];
		short &currState = vertexState[localVertex];

		if (currState == stateInitial) {
			currDepth = newDepth;
			currState = stateJustFilled;
			isNextStepRequired = true;
		}
	}
}

#endif	/* DEPTHBUILDERBUFFERED_H */
```

```
#ifndef     DEPTHBUILDERP2PNOBLOCK_H
#define     DEPTHBUILDERP2PNOBLOCK_H

namespace dgmark {

      class DepthBuilderP2PNoBlock : public Communicable, public DepthBuilder{
      public:
            DepthBuilderP2PNoBlock(Intracomm *comm, Graph *graph);
            virtual ~DepthBuilderP2PNoBlock();

      protected:
            virtual void buildNextStep();
            virtual void prepare(Vertex root);

      private:
            Vertex getDepth(Vertex currVertex);
            void synchAction();
            void startRecv();
            void waitForOthersToEnd();

            Request recvRequest;
            bool isRecvActive;
            Vertex requestedVertex;
      };

      void DepthBuilderP2PNoBlock::buildNextStep()
      {
            isNextStepRequired = false;

            for (size_t local = 0; local < graph->numLocalVertex; ++ local) {
                  const Vertex currParent = parent[local];
                  Vertex &currDepth = depth[local];

                  //was not visited or depth is already built.
                  if (currParent == graph->numGlobalVertex
                  || currDepth < graph->numGlobalVertex) {
                        continue;
                  }

                  const Vertex parentDepth = getDepth(currParent);

                  if (parentDepth == graph->numGlobalVertex) {
                        continue;
                  }

                  if (currDepth == graph->numGlobalVertex) {
                        currDepth = parentDepth + 1;
                        isNextStepRequired = true;
                  }

                  synchAction();
            }

            waitForOthersToEnd();
            comm->Allreduce(IN_PLACE, &isNextStepRequired, 1, SHORT, LOR);

            if (isRecvActive) {
                  recvRequest.Cancel();
                  isRecvActive = false;
            }
      }
```

```
	void DepthBuilderP2PNoBlock::prepare(Vertex root)
	{
		DepthBuilder::prepare(root);

		if (graph->vertexRank(root) == rank) {
			Vertex rootLocal = graph->vertexToLocal(root);
			depth[rootLocal] = 0;
		}
		isRecvActive = false;
	}

	void DepthBuilderP2PNoBlock::waitForOthersToEnd()
	{
		requestSynch(true, SYNCH_END_TAG); // curr node work is over;
		int endedProcesses = 1;
		while (endedProcesses < size) {
			synchAction();
			while (probeSynch(SYNCH_END_TAG)) {
				++endedProcesses;
			}
		}
	}

	Vertex DepthBuilderP2PNoBlock::getDepth(Vertex tgtVertex)
	{
		const int tgtRank = graph->vertexRank(tgtVertex);
		const Vertex tgtLocal = graph->vertexToLocal(tgtVertex);
		Vertex tgtDepth;
		if (tgtRank == rank) {
			tgtDepth = depth[tgtLocal];
		} else {
			sendVertex(tgtLocal, tgtRank, LOCAL_SEND_TAG);
			Request recvReq = comm->Irecv(&tgtDepth, 1, VERTEX_TYPE,
							tgtRank, DEPTH_SEND_TAG);
			while (!recvReq.Test()) {
				synchAction();
			}
		}
		return tgtDepth;
	}

	void DepthBuilderP2PNoBlock::synchAction()
	{
		Status status;
		if (isRecvActive && recvRequest.Test(status)) {
			sendVertex(depth[requestedVertex],
					status.Get_source(), DEPTH_SEND_TAG);
			isRecvActive = false;
		}
		startRecv();
	}

	void DepthBuilderP2PNoBlock::startRecv()
	{
		if (!isRecvActive) {
			recvRequest = comm->Irecv(&requestedVertex,
				1, VERTEX_TYPE, ANY_SOURCE, LOCAL_SEND_TAG);
			isRecvActive = true;
		}
	}
}
#endif	/* DEPTHBUILDERP2PNOBLOCK_H */
```

```
#ifndef     BFSDGMARK_H
#define     BFSDGMARK_H

namespace dgmark {

      class BFSdgmark : public SearchTask {
      public:
            BFSdgmark(Intracomm *comm);
            BFSdgmark(const BFSdgmark& orig);
            virtual ~BFSdgmark();

            virtual ParentTree* run();

      protected:
            static const int BFS_SYNCH_TAG = 541;
            static const int BFS_DATA_TAG = 7353;

            /**
             * queue is a queue of vertex (local).
             * Traversed vertex adds to the end of the next queue.
             * queue[0] is a length
             * Elements are located in 1..queue[0] indecies.
             */
            Vertex *queue;

            /**
             * Queue, prepared for next step.
             */
            Vertex *nextQueue;

            /**
             * parent is an array, which associates vertex with it parent
             * (global) in tree.
             * parent[root] is always must be root.
             * parent[visited] >= 0 and \<= numGlobalVertex
             * parent[initially] == numGlobalVertex
             * Note: contains local vertex only.
             */
            Vertex *parent;

            /**
             * Performs BFS step.
             */
            virtual void performBFS() = 0;

            /**
             * Performs actual BFS step.
             */
            virtual void performBFSActualStep();

            /**
             * Processes local child.
             * Adds it to local queue and sets it's parent, if it was not set.
             */
            virtual void processLocalChild(Vertex parentVertexGlobal, Vertex
childVertexLocal);

            /**
             * Processes global child.
             * Adds it to local queue and sets it's parent, if it was not set.
             */
            virtual void processGlobalChild(Vertex currVere,Vertex child) = 0;
```

```
            /**
             * Swaps queues.
             * Swaps next and current queues.
             */
            virtual void swapQueues();

            /**
             * Cleans queues and parent.
             */
            void resetQueueParent();

            /**
             * Function to calculate queue size.
             */
            virtual Vertex getQueueSize();

            /**
             * Calculaded need of next BFS step.
             * @return true, if next step needed.
             */
            virtual bool isNextStepNeeded();
      };

      ParentTree* BFSdgmark::run()
      {
            log << "Running BFS (" << getName() << ") from " << root << "\n";
            comm->Barrier();
            double startTime = Wtime();

            resetQueueParent();

            if (graph->vertexRank(root) == rank) {
                  //root is my vertex, put it into queue.
                  Vertex rootLocal = graph->vertexToLocal(root);
                  parent[rootLocal] = root;
                  queue[1] = rootLocal;
                  queue[0] = 1;
            }

            //main loop
            stepCount = 0;
            while (isNextStepNeeded()) {
                  performBFS();
                  comm->Barrier();
                  swapQueues();
                  stepCount++;
            }

            comm->Barrier();
            double taskRunTime = Wtime() - startTime;

            Vertex *resultParent = new Vertex[numLocalVertex];
            for (int i = 0; i < numLocalVertex; ++i) {
                  resultParent[i] = parent[i];
            }

            ParentTree *parentTree = new ParentTree(comm, root, resultParent,
graph, taskRunTime);
            log << "BFS time: " << taskRunTime << " s\n";
            return parentTree;
      }
```

```
	inline void BFSdgmark::performBFSActualStep()
	{
		vector<Edge*> *edges = graph->edges;
		const size_t queueEnd = queue[0];

		for (size_t queueIndex = 1; queueIndex <= queueEnd; ++queueIndex)
{
			const Vertex currVertex = queue[queueIndex];

			size_t childStart = graph->getStartIndex(currVertex);
			size_t childEnd = graph->getEndIndex(currVertex);
			for (size_t cIdx = childStart; cIdx < childEnd; ++cIdx) {
				const Vertex child = edges->at(cIdx)->to;
				const Vertex childLocal = graph->vertexToLocal(child);
				const int childRank = graph->vertexRank(child);
				if (childRank == rank) {
		processLocalChild(graph->vertexToGlobal(currVertex), childLocal);
				} else {
		processGlobalChild(graph->vertexToGlobal(currVertex), child);
				}
			}
			++queue[0];
		}
	}

	inline void BFSdgmark::processLocalChild(Vertex parentVertexGlobal,
						Vertex childVertexLocal)
	{
		if (parent[childVertexLocal] == graph->numGlobalVertex) {
			parent[childVertexLocal] = parentVertexGlobal;
			nextQueue[++nextQueue[0]] = childVertexLocal;
		}
	}

	inline void BFSdgmark::resetQueueParent()
	{
		memset(queue, 0, getQueueSize() * sizeof(Vertex));
		queue[0] = 0;
		nextQueue[0] = 0;

		for (size_t i = 0; i < numLocalVertex; ++i) {
			parent[i] = graph->numGlobalVertex;
		}
	}

	inline Vertex BFSdgmark::getQueueSize()
	{
		return numLocalVertex;
	}

	bool BFSdgmark::isNextStepNeeded()
	{
		//finds OR for "isQueueEnlarged" in all processes.
		bool isQueueEnlarged = queue[0] > 0;
		comm->Allreduce(IN_PLACE, &isQueueEnlarged, 1, BOOL, LOR);
		return isQueueEnlarged;
	}
}

#endif	/* BFSDGMARK_H */
```

```
#ifndef     BFSTASKRMAFETCH_H
#define     BFSTASKRMAFETCH_H

namespace dgmark {

      class BFSTaskRMAFetch : public BFSdgmark {
      public:
            BFSTaskRMAFetch(Intracomm *comm);
            BFSTaskRMAFetch(const BFSTaskRMAFetch& orig);
            virtual ~BFSTaskRMAFetch();
            virtual string getName();
            virtual void open(Graph *newGraph);
            virtual void close();

      protected:
            RMAWindow<Vertex> *qWin;
            RMAWindow<Vertex> *nextQWin;
            RMAWindow<Vertex> *pWin;
            virtual void swapQueues();
            virtual void performBFS();
            virtual void processGlobalChild(Vertex currVertex, Vertex child);

      private:
            /**
             * Performs RMA synchronization.
             * Purpose to allow main process in queue to perform actual BFS.
             */
            void performBFSSynchRMA();
      };

      void BFSTaskRMAFetch::performBFS()
      {
            for (int node = 0; node < size; ++node) {
                  if (rank == node) {
                        performBFSActualStep();
                        endSynch(BFS_SYNCH_TAG);
                  } else {
                        performBFSSynchRMA();
                  }
                  comm->Barrier();
            }
      }

      void BFSTaskRMAFetch::performBFSSynchRMA()
      {
            while (true) {
               if (waitSynch(BFS_SYNCH_TAG)) {
                  pWin->fenceOpen(MODE_NOPUT); //allow read parent
                  pWin->fenceClose(MODE_NOSTORE);
                  if (waitSynch(BFS_SYNCH_TAG)) {
                        pWin->fenceOpen(MODE_NOPUT); //allow to write parent
                        pWin->fenceClose(MODE_NOSTORE);
                        nextQWin->fenceOpen(MODE_NOPUT); //allow to read queue
                        nextQWin->fenceClose(MODE_NOSTORE);
                        nextQWin->fenceOpen(MODE_NOPUT); //allow put to queue
                        nextQWin->fenceClose(MODE_NOSTORE);
                  }
               } else {
                  break; //if fence is not neaded mode
               }
            }
      }
```

```
	void BFSTaskRMAFetch::processGlobalChild(Vertex currVertex,
	                                         Vertex child)
	{
		Vertex childLocal = graph->vertexToLocal(child);
		const int childRank = graph->vertexRank(child);
		Vertex parentOfChild;

		requestSynch(true, BFS_SYNCH_TAG); //fence is needed now
		// reading parent of child
		pWin->fenceOpen(MODE_NOPUT);
		pWin->get(&parentOfChild, 1, childRank, childLocal);
		pWin->fenceClose(0);

		assert(0 <= parentOfChild
		       && parentOfChild <= graph->numGlobalVertex);

		bool isInnerFenceNeeded =
		                    (parentOfChild == graph->numGlobalVertex);
		requestSynch(isInnerFenceNeeded, BFS_SYNCH_TAG);

		// call for inner fence if it is needed
		if (isInnerFenceNeeded) {
			pWin->fenceOpen(0);
			pWin->put(&currVertex, 1, childRank, childLocal);
			pWin->fenceClose(MODE_NOSTORE);

			//Updating queue
			Vertex queueLastIndex;

			nextQWin->fenceOpen(MODE_NOPUT);
			// get queue[0] (queue length)
			nextQWin->get(&queueLastIndex, 1, childRank, 0);
			nextQWin->fenceClose(0);

			assert(0 <= queueLastIndex
			&& queueLastIndex <= getQueueSize());

			nextQWin->fenceOpen(0);

			// write to queue end
			nextQWin->put(&childLocal, 1, childRank, ++queueLastIndex);

			//update length of queue
			nextQWin->put(&queueLastIndex, 1, childRank, 0);
			nextQWin->fenceClose(MODE_NOSTORE);
		}
	}
}

#endif	/* BFSTASKRMAFETCH_H */
```

```
#ifndef BFSTASKP2PNOBLOCK_H
#define     BFSTASKP2PNOBLOCK_H

namespace dgmark {

      class BFSTaskP2PNoBlock :     public BFSTaskP2P,
                                    public BufferedDataDistributor {
      public:
            BFSTaskP2PNoBlock(Intracomm *comm);
            virtual ~BFSTaskP2PNoBlock();
            virtual string getName();
      protected:
            virtual void performBFS();
            virtual void processGlobalChild(Vertex currVertex, Vertex child);
            virtual void processRecvData(size_t countToRead);
      private:
            static const size_t ELEMENT_SIZE = 2;
            static const size_t BUFFERED_ELEMENTS = 256;

      };

      void BFSTaskP2PNoBlock::performBFS()
      {
            prepareBuffers();
            performBFSActualStep();
            flushBuffers();
            waitForOthersToEnd();
      }

      void BFSTaskP2PNoBlock::processGlobalChild(Vertex currVertex,
                                                 Vertex child)
      {
            const Vertex childLocal = graph->vertexToLocal(child);
            const int childRank = graph->vertexRank(child);

            while (isSendRequestActive[childRank]) {
                  probeSynchData();
            }

            size_t &currCount = countToSend[childRank];
            Vertex *&currBuffer = sendBuffer[childRank];
            currBuffer[currCount] = childLocal;
            currBuffer[currCount + 1] = currVertex;
            currCount += 2;

            if (currCount == sendPackageSize) {
                  sendData(childRank);
            }

            probeSynchData();
      }

      void BFSTaskP2PNoBlock::processRecvData(size_t countToRead)
      {
            for (size_t dataIndex = 0;dataIndex < countToRead;dataIndex += 2){
                  const Vertex currLocal = recvBuffer[dataIndex];
                  const Vertex parentGlobal = recvBuffer[dataIndex + 1];
                  processLocalChild(parentGlobal, currLocal);
            }
      }
}
#endif      /* BFSTASKP2PNOBLOCK_H */
```

```
#ifndef     BFSTASKP2P_H
#define     BFSTASKP2P_H

namespace dgmark {

      class BFSTaskP2P : public BFSdgmark {
      public:
            BFSTaskP2P(Intracomm *comm);
            BFSTaskP2P(const BFSTaskP2P& orig);
            virtual ~BFSTaskP2P();

            virtual string getName();
            virtual void open(Graph *newGraph);
            virtual void close();
      protected:
            virtual void performBFS();
            virtual void processGlobalChild(Vertex currVertex, Vertex child);
            virtual void performBFSSynch();
      };

      void BFSTaskP2P::performBFS()
      {
            for (int node = 0; node < size; ++node) {
                  if (rank == node) {
                        performBFSActualStep();
                        endSynch(BFS_SYNCH_TAG);
                  } else {
                        performBFSSynch();
                  }
                  comm->Barrier();
            }
      }

      void BFSTaskP2P::processGlobalChild(Vertex currVertex, Vertex child)
      {
            const Vertex childLocal = graph->vertexToLocal(child);
            const int childRank = graph->vertexRank(child);
            Vertex memory[2] = {childLocal, currVertex};
            requestSynch(true, childRank, BFS_SYNCH_TAG);
            comm->Send(&memory[0], 2, VERTEX_TYPE, childRank, BFS_DATA_TAG);
      }

      void BFSTaskP2P::performBFSSynch()
      {
            Status status;
            Vertex memory[2] = {0};
            while (true) {
                  if (waitSynch(BFS_SYNCH_TAG, status)) {
                        comm->Recv(&memory[0], 2, VERTEX_TYPE,
                                    status.Get_source(), BFS_DATA_TAG);
                        const Vertex currLocal = memory[0];
                        const Vertex parentGlobal = memory[1];
                        BFSdgmark::processLocalChild(parentGlobal, currLocal);
                  } else {
                        break; //synchronization is not neaded mode
                  }
            }
      }
}

#endif      /* BFSTASKP2P_H */
```

Приложение В
(обязательное).
Демонстрационный материал

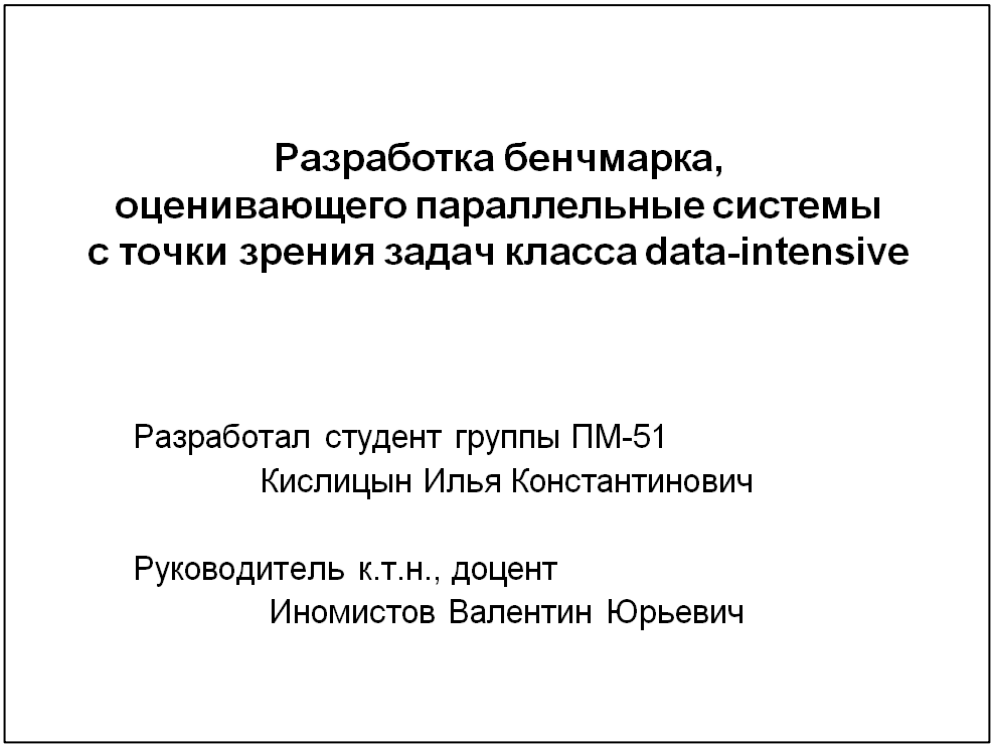

Рисунок В.1 – Титульный слайд презентации

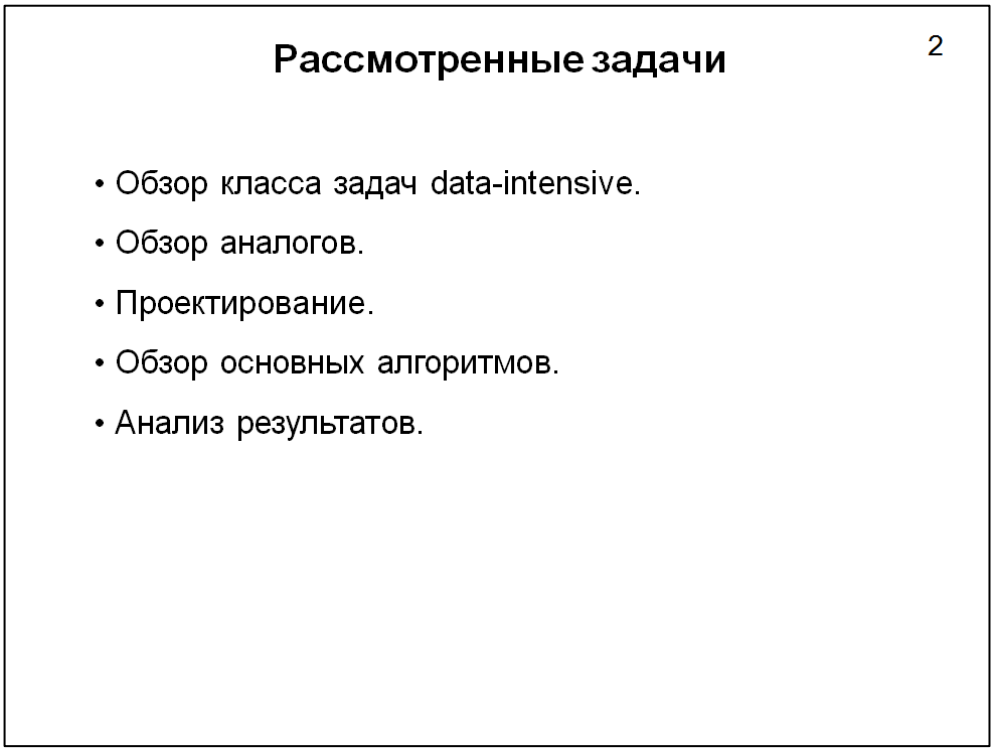

Рисунок В.2 – Рассмотренные задачи

Класс задач data-intensive

3

Задача принадлежит классу data-intensive, если она оперирует большим объёмом данных, а объём вычислительных операций существенно меньше операций по перемещению данных.

К этому классу относятся:

• задачи на графах (например поиск в ширину);

• задачи над большими данными (например MapReduce).

Рисунок В.3 – Класс задач data-intensive

Обзор аналогов

4

Единственным аналогом является бенчмарк Graph500.

Достоинства:

- создан в 2010 году, на данный момент протестирован на 160 суперкомпьютерах и кластерах.

Недостатки:

- неподдерживаемый, плохо масштабируемый код;
- неэффективное использование памяти;
- низкая скорость работы.

Рисунок В.4 – Аналоги

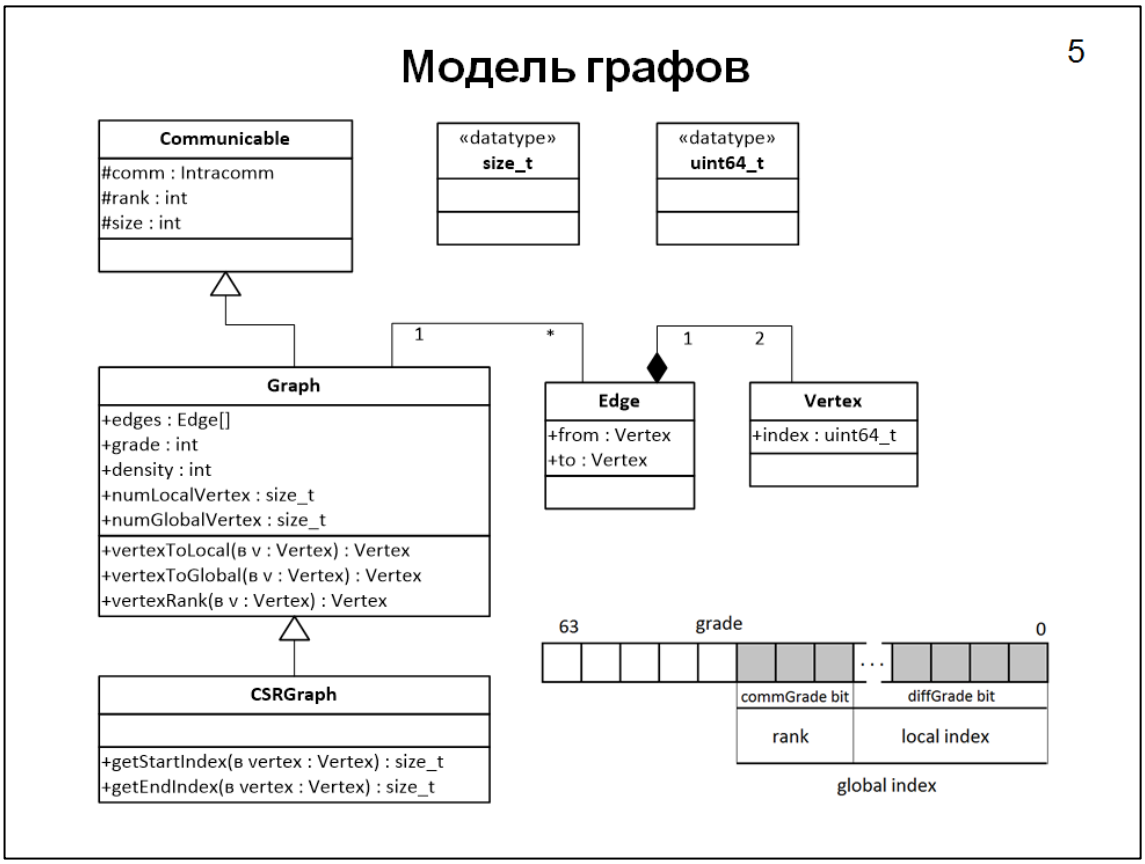

Рисунок В.5 – Модель графов

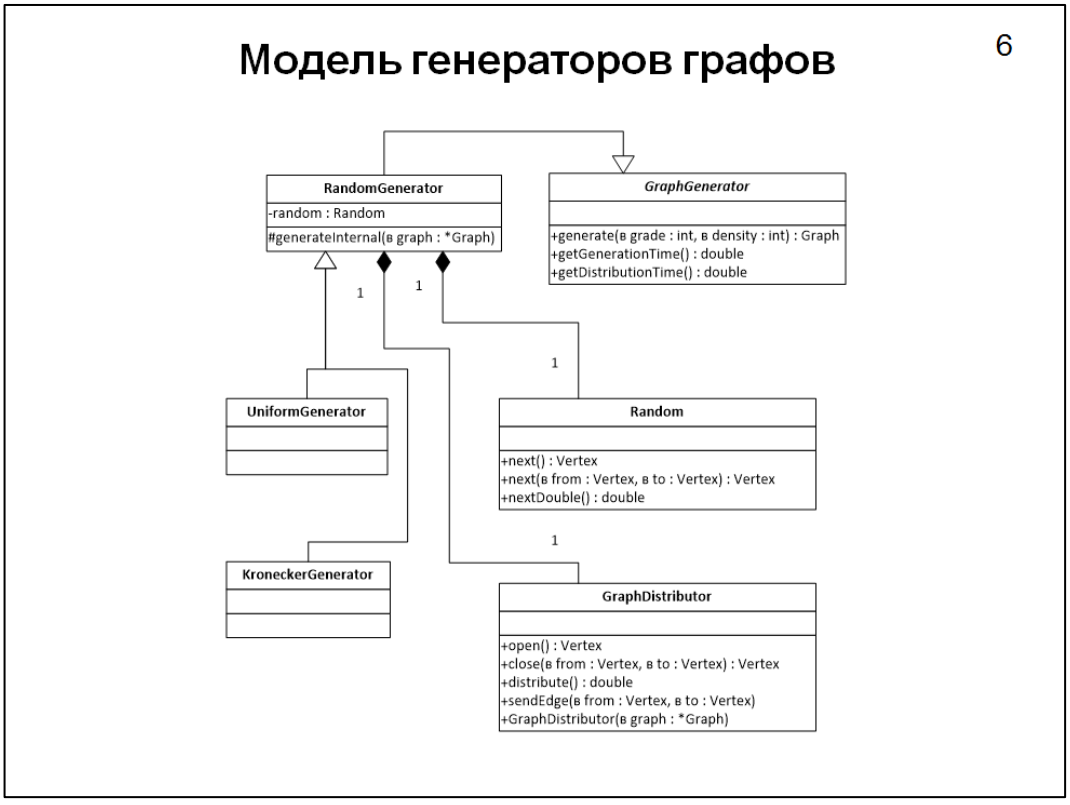

Рисунок В.6 – Модель генератора графов

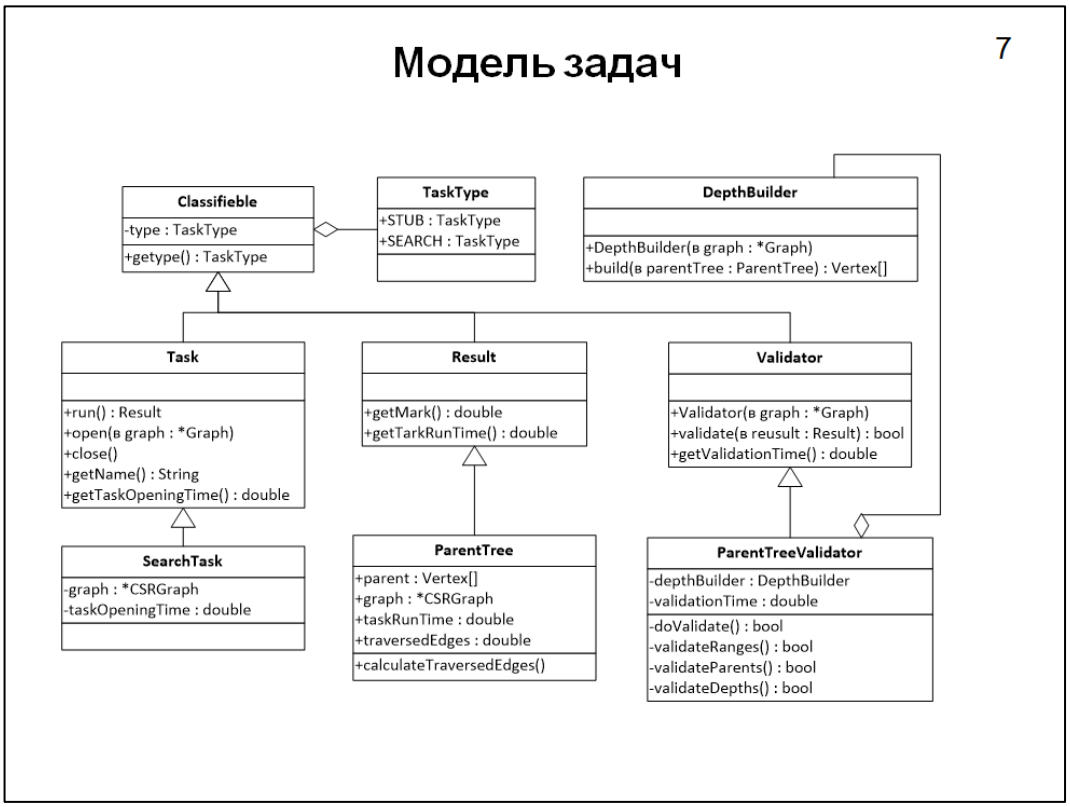

Рисунок В.7 – Модель задач

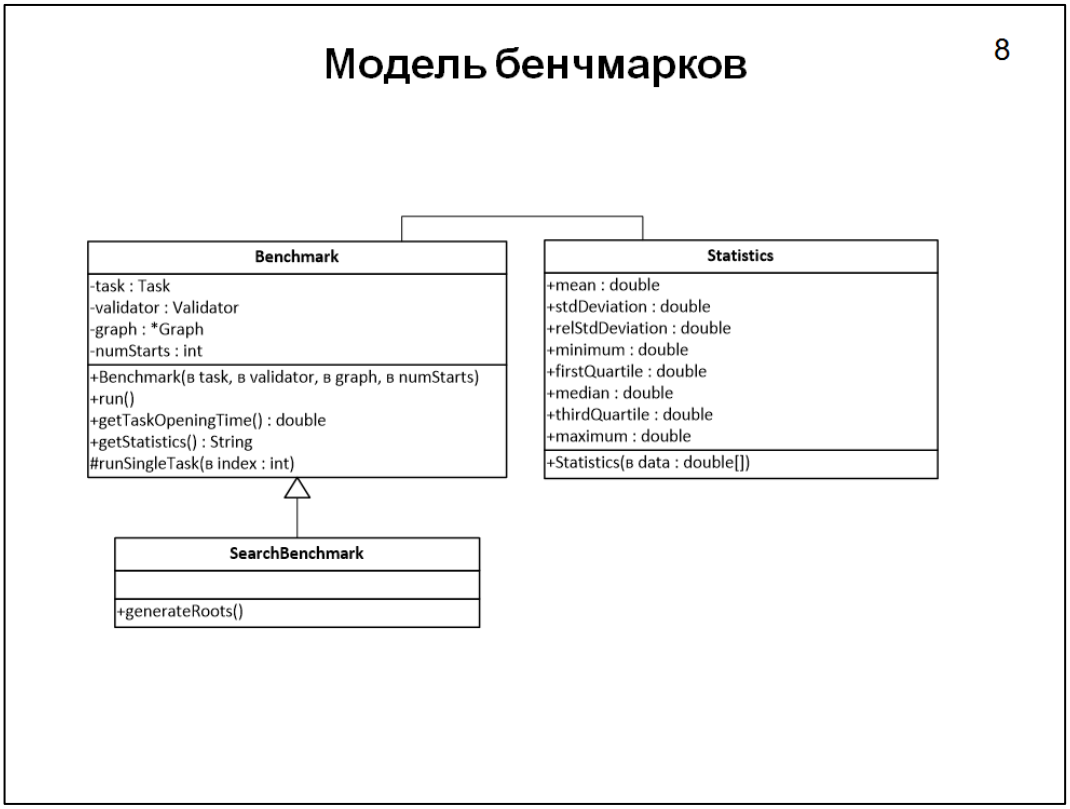

Рисунок В.8 – Модель бенчмарка

Рисунок В.9 – Модель контроллера

Рисунок В.10 – Основная деятельность

Рисунок В.11 – Деятельность контроллера

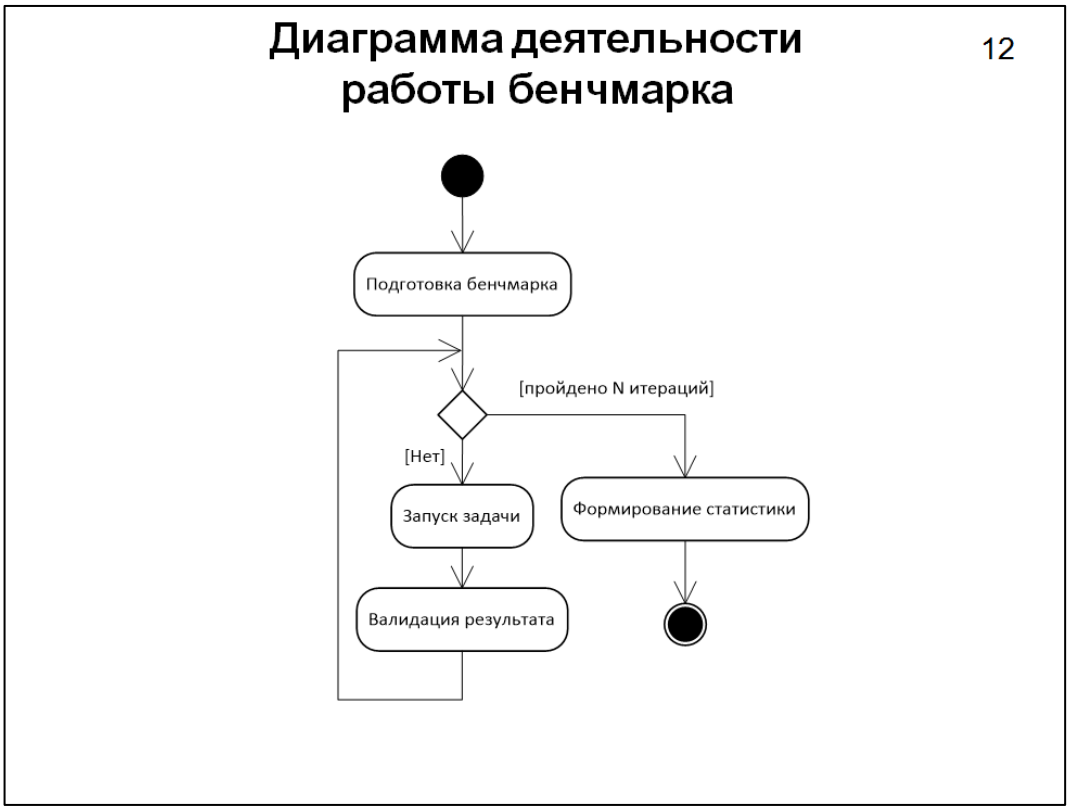

Рисунок В.12 – Деятельность бенчмарка

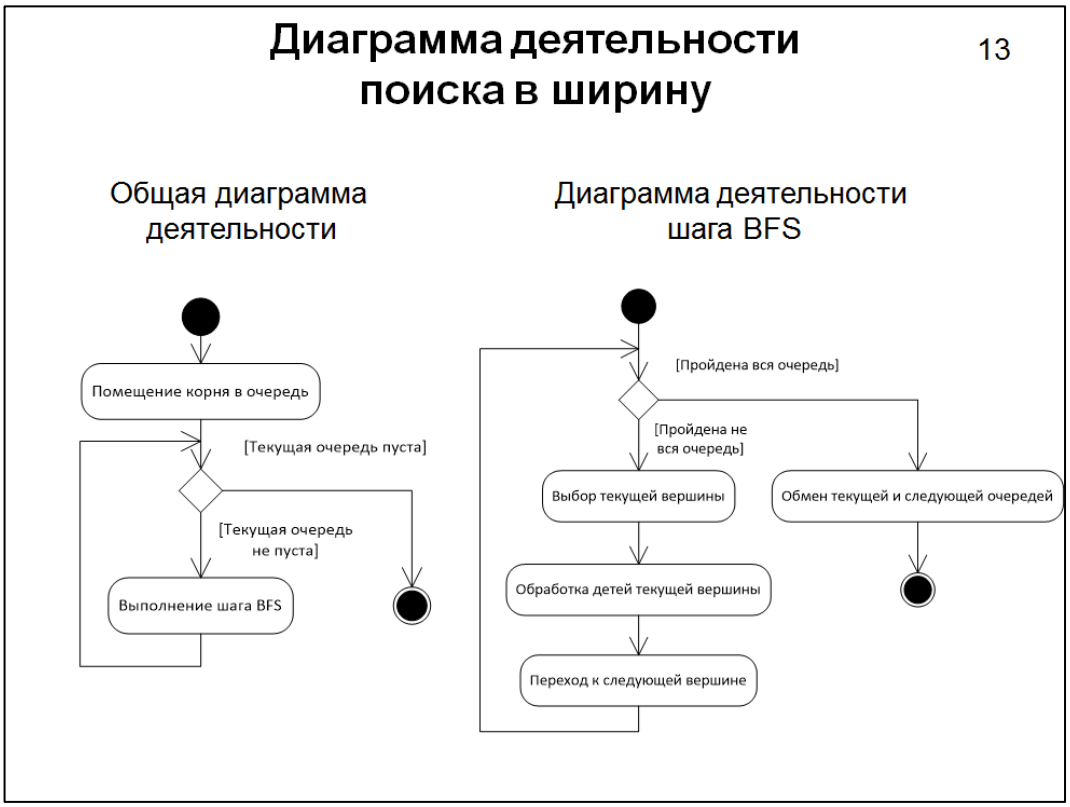

Рисунок В.13 – Алгоритм поиска в ширину

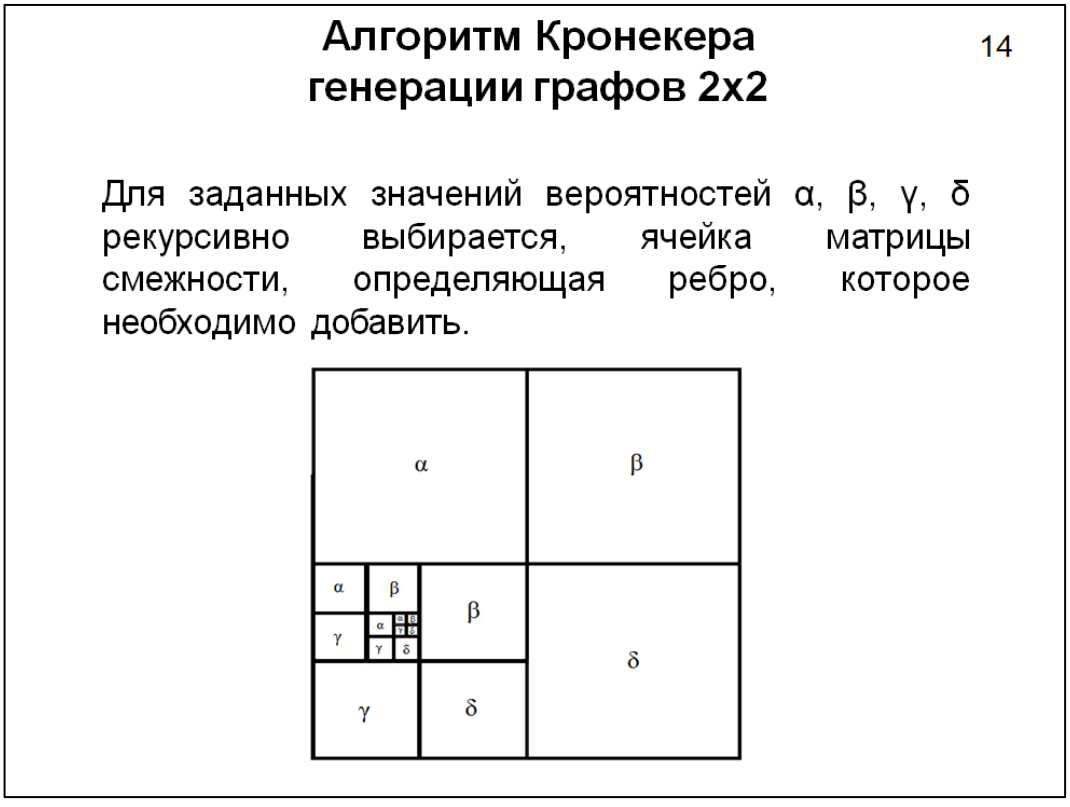

Рисунок В.14 – Алгоритм Кронекера

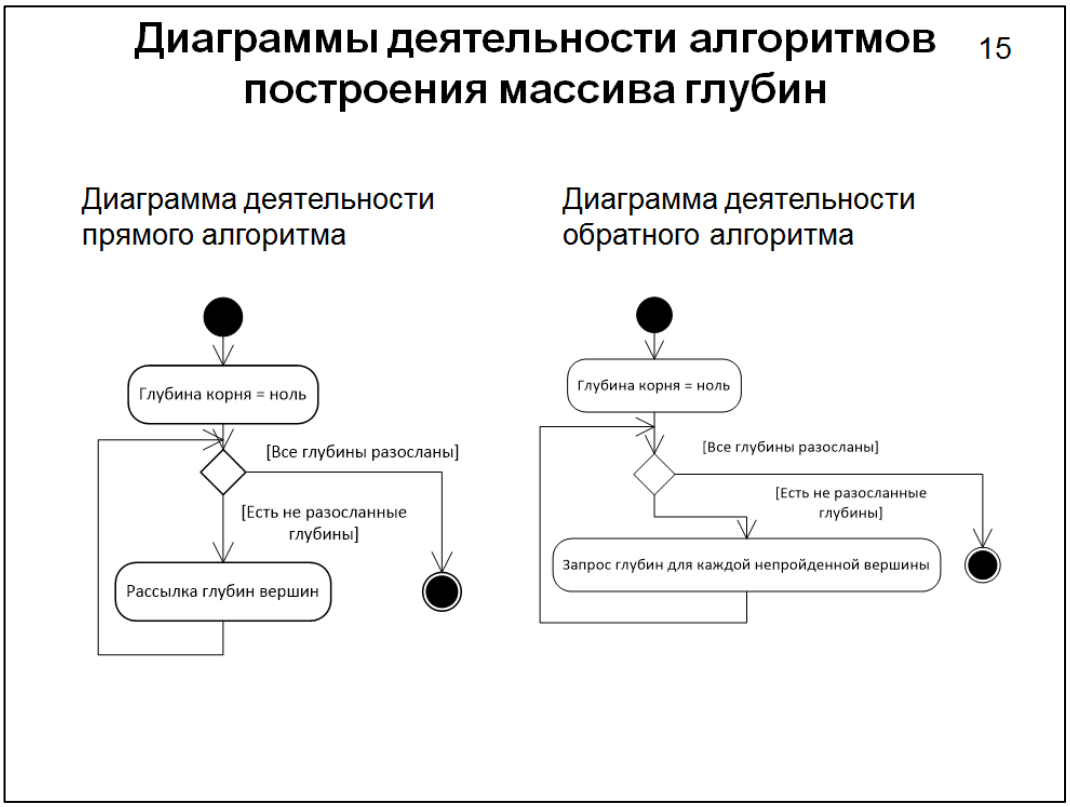

Рисунок В.15 – Алгоритм построения глубин

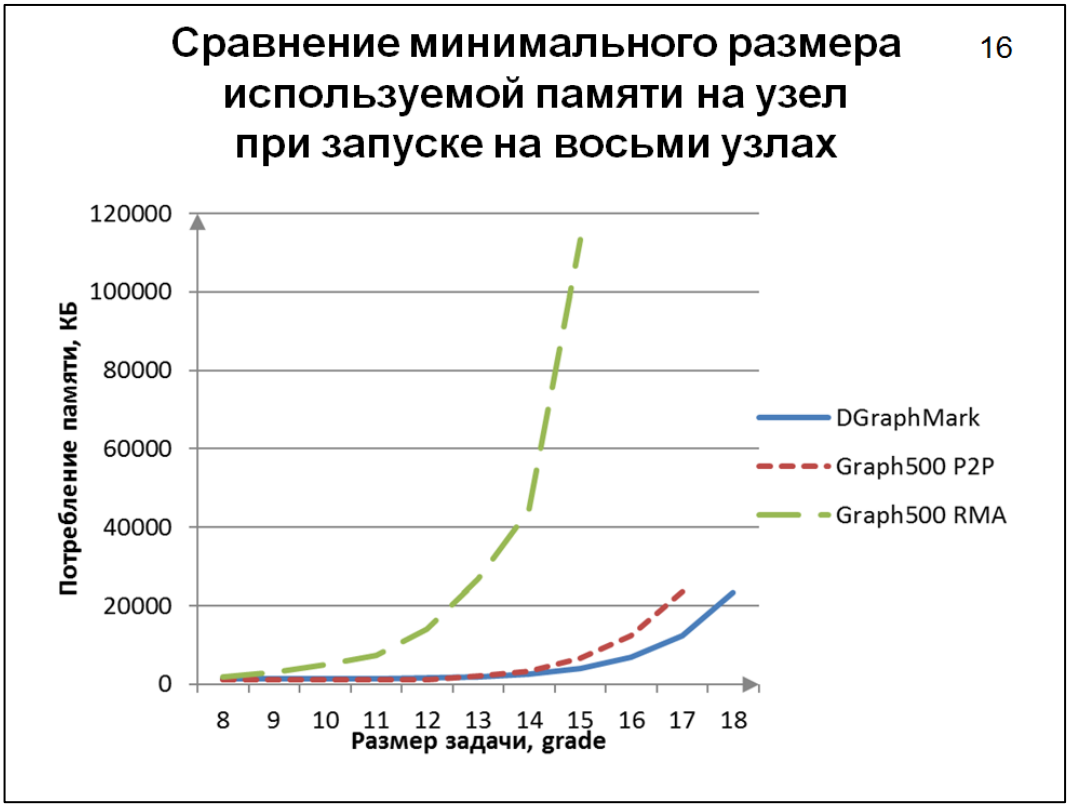

Рисунок В.16 – Сравнение минимального размера используемой памяти

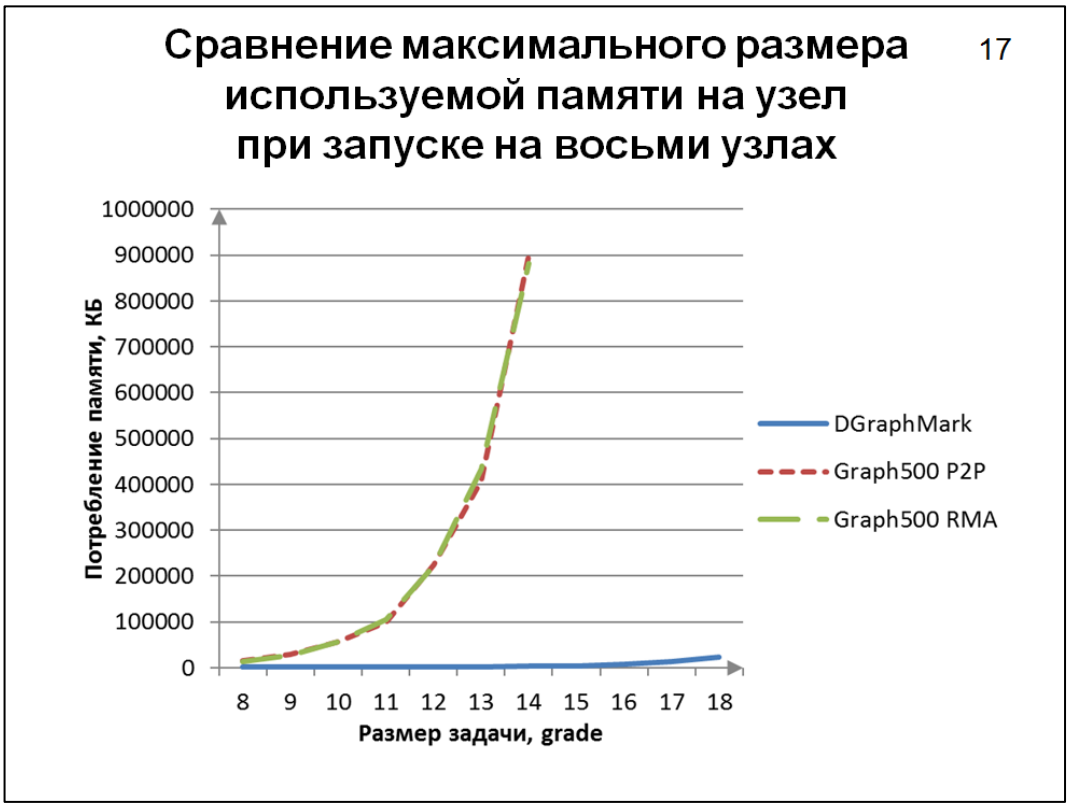

Рисунок В.17 – Сравнение максимального размера используемой памяти

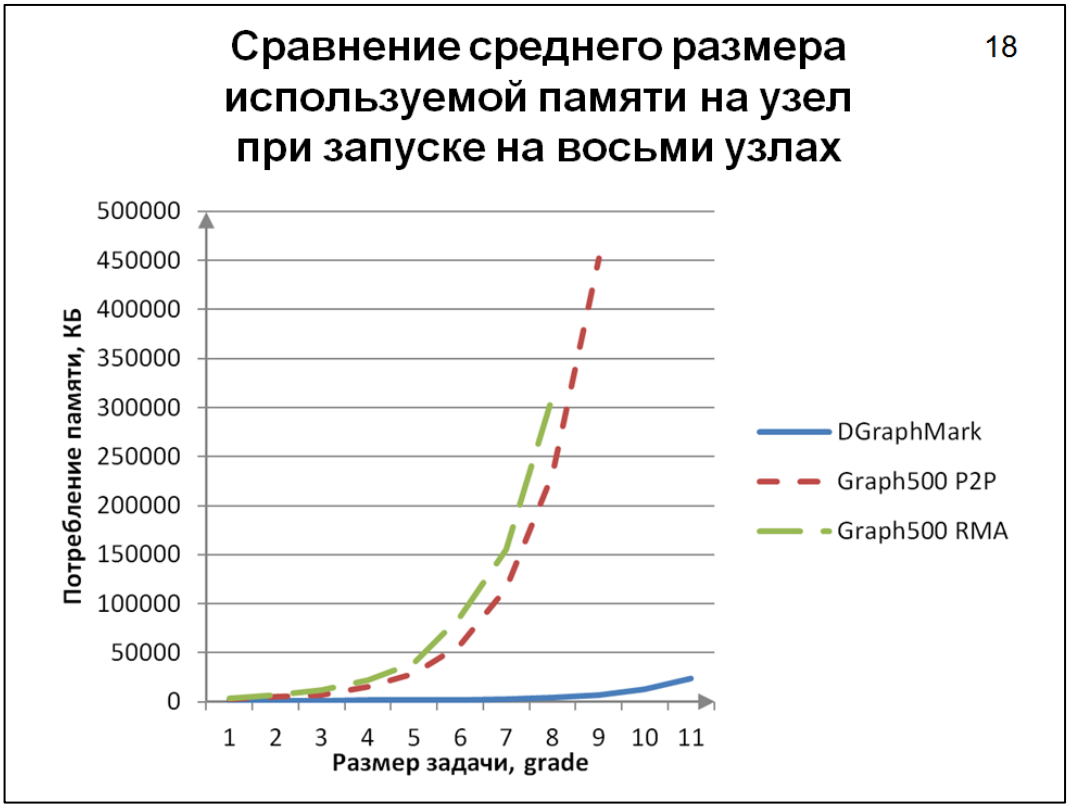

Рисунок В.18 – Сравнение среднего размера используемой памяти

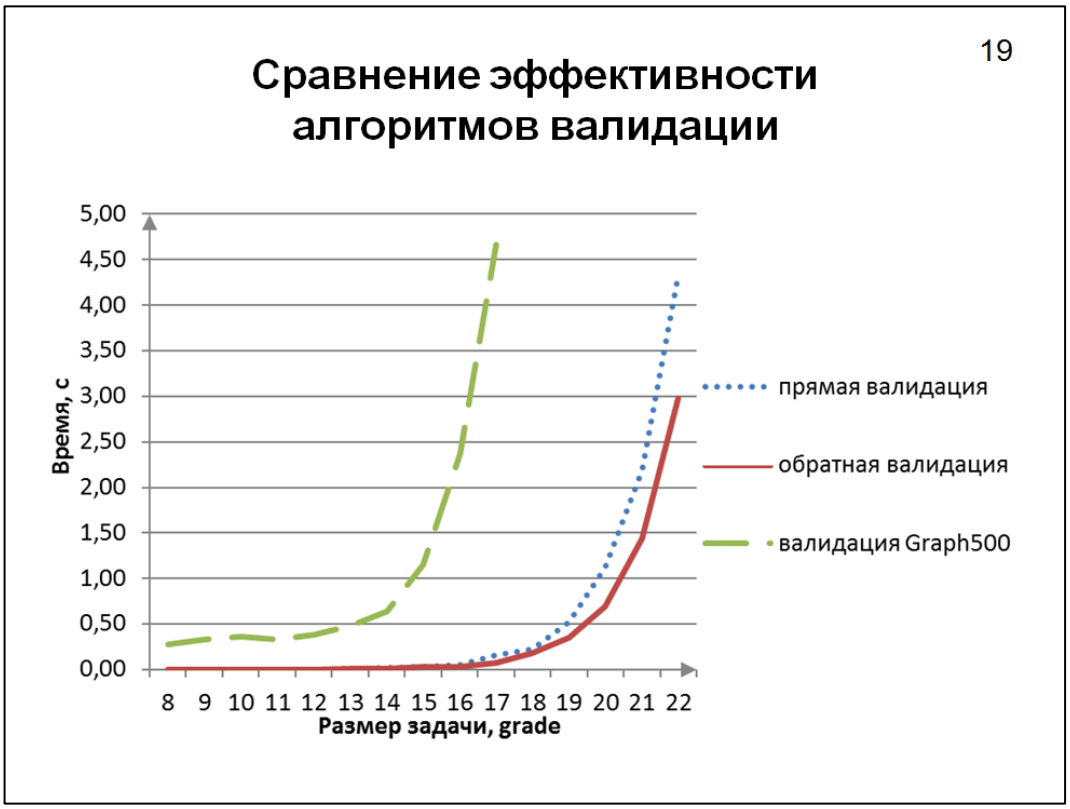

Рисунок В.19 – Сравнение алгоритмов валидации

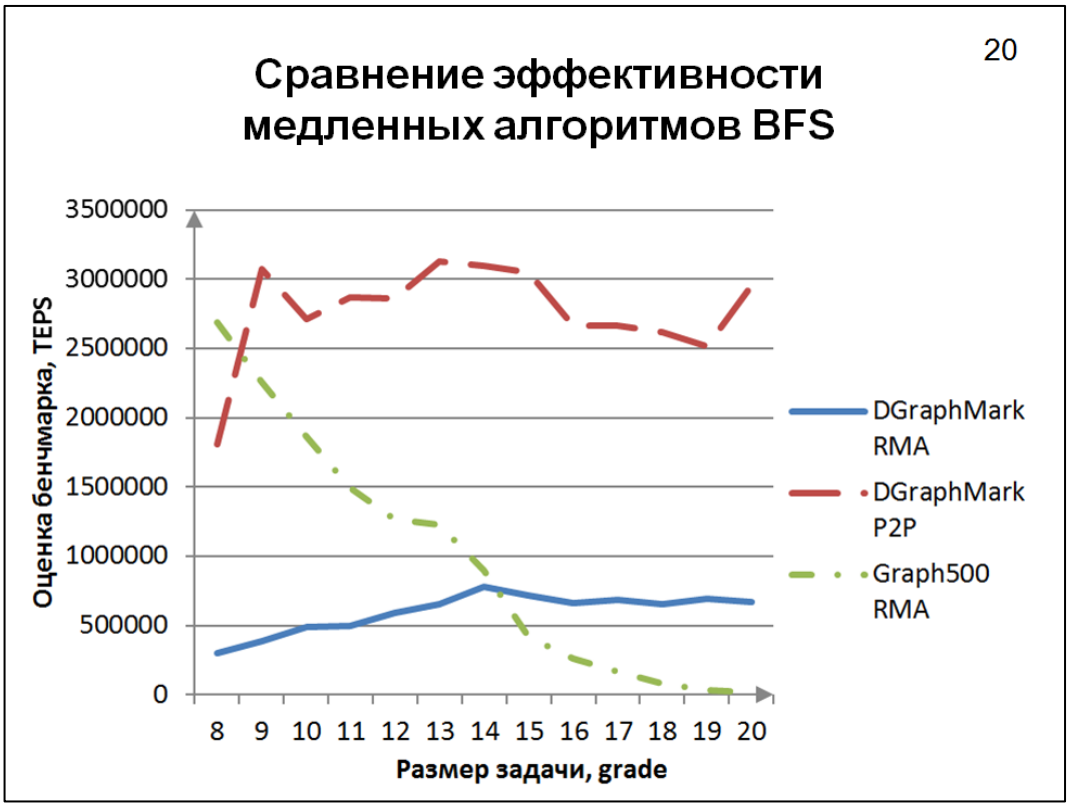

Рисунок В.20 – Сравнение медленных алгоритмов BFS

Рисунок В.21 – Сравнение быстрых алгоритмов BFS

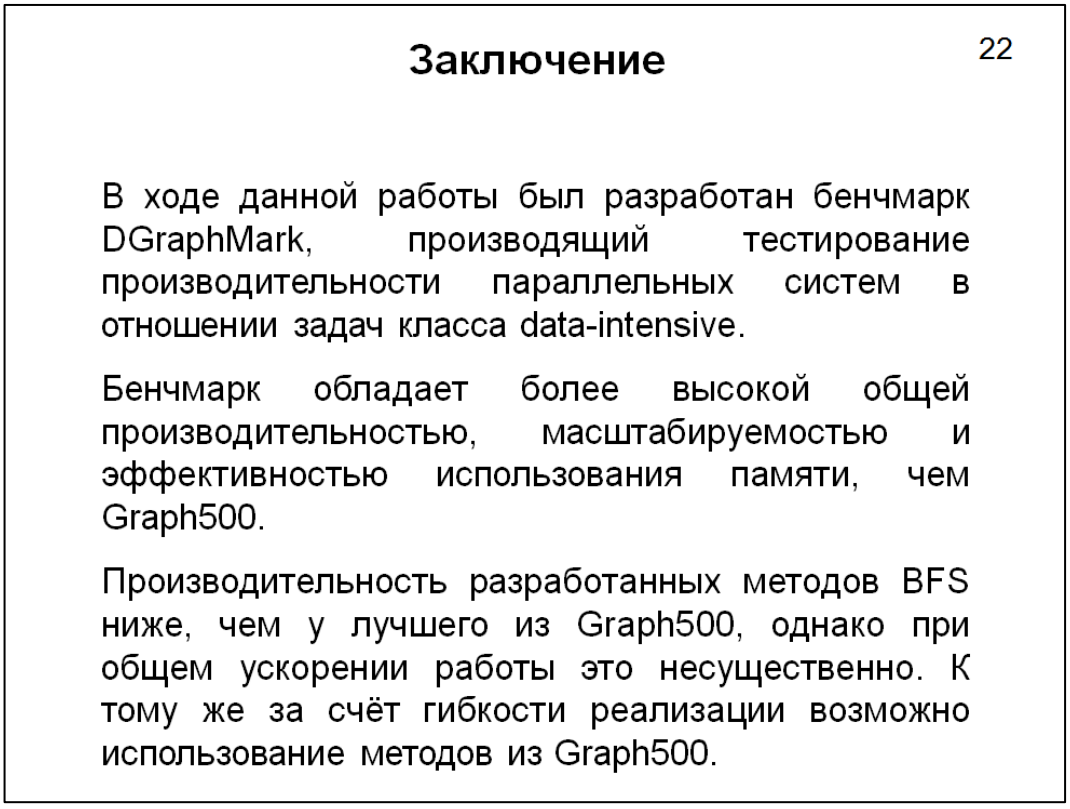

Рисунок В.22 – Слайд заключения

Приложение Г
(обязательное).
Авторская справка

Я, Кислицын Илья Константинович, автор дипломной работы сообщаю, что мне известно о персональной ответственности автора за разглашение сведений, подлежащих защите законами РФ о защите объектов интеллектуальной собственности.

Одновременно сообщаю, что

1) при подготовке к защите (опубликованию) дипломной работы не использованы источники (документы, отчеты, диссертации, литература и т.п.), имеющие гриф секретности или “Для служебного пользования“ ВятГУ или другой организации;

2) данная работа не связана с незавершенными исследованиями или уже с завершенными, но еще официально не разрешенными к опубликованию ВятГУ или другими организациями;

3) данная работа не содержит коммерческую информацию, способную нанести ущерб интеллектуальной собственности ВятГУ или другой организации;

4) данная работа не является результатом НИР или ОКР, выполняемой по договору с организацией;

5) в предлагаемом к опубликованию тексте нет данных по незащищенным объектам интеллектуальной собственности других авторов;

6) согласен на использование результатов своей работы ВятГУ для учебного процесса;

7) использование моей дипломной работы в научных исследованиях оформляется в соответствии с законодательством РФ о защите интеллектуальной собственности.

«____» _____ 2014 г. Подпись автора _____

Сведения по авторской справке подтверждаю

«____» _____ 2014 г. _____
Зав. кафедрой

Приложение Д
(обязательное).
Библиографический список

1. Эйсымонт Л., Фролов А., Семенов А. Graph500: адекватный рейтинг // Издательство "Открытые системы". 2011. URL: http://www.osp.ru/os/2011/01/13006961/ (дата обращения: 12.03.2014).
2. // MPI Documents: [сайт]. [2012]. URL: http://www.mpi-forum.org/docs/docs.html (дата обращения: 1.01.2014).
3. // nsf.gov - IIS - Funding - Data-intensive Computing : [сайт]. [2009]. URL: http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324&org=IIS (дата обращения: 06.04.2013).
4. // Graph 500 Benchmark Specification: [сайт]. [2011]. URL: http://www.graph500.org/specifications (дата обращения: 01.01.2014).
5. Gokhale M., Cohen J., Yoo A., Miller W.M., Arpith J., Ulmer C., Pearce R. Hardware Technologies for High-Performance Data-Intensive Computing // Computation. 2008. URL: http://computation.llnl.gov/casc/dcca-pub/dcca/Papers_files/data-intensive-ieee-computer-0408.pdf (дата обращения: 06.04.2014).
6. Lin J., Dyer C. Data-Intensive Text Processing with MapReduce // Data-Intensive Text Processing. 2010. URL: http://lintool.github.io/MapReduceAlgorithms/ (дата обращения: 06.04.2014).
7. Goldberg A., Mills H.P., Nyland S.L., Prins F.J. A Design Methodology for Data-Parallel Applications // Allen Goldberg personal web-site. 2000. URL: http://www.agoldberg.org/Publications/DesignMethForDP.pdf (дата обращения: 06.04.2014).
8. Волков Д., Фролов А. Оценка быстродействия нерегулярного доступа к памяти // Издательство "Открытые системы". 2008. URL: http://www.osp.ru/os/2008/01/4836914/ (дата обращения: 06.04.2014).
9. Groer C., Sullivan D.B., Poole S. A Mathematical Analysis of the R-MAT Random Graph Generator // Oak Ridge National library. 2011. URL: http://web.ornl.gov/~tcg/publications/rmat.pdf (дата обращения: 07.04.2014).
10. Leskovec J., Chakrabarti D., Kleinberg J., Faloutsos C., Ghahramani Z. Kronecker Graphs: An Approach to Modeling Networks // Standford Computer Science. 2010. URL: http://cs.stanford.edu/people/jure/pubs/kronecker-jmlr10.pdf (дата обращения: 07.04.2014).
11. McConnell S. Code Complete. Redmond, Washington: Microsoft Press, 2004.
12. Linux kernel coding style // Linux Kernel Archive. URL: https://www.kernel.org/doc/Documentation/CodingStyle (дата обращения: 01.01.2014).