Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра Вычислительных систем
Допустить к защите

зав. каф. ____________________Мамойленко С.Н.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Оптимизация решения разреженных систем линейных уравнений на системах с общей памятью

Пояснительная записка

Студент: ___________________/Крюкова Л.П./

Факультет ИВТ ___________________ Группа ИС-241

Руководитель ________________________/Пудов С.Г./

Новосибирск - 2016

Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Форма утверждена
научно-методическим
советом ФГОБУ ВПО «СибГУТИ»
Протокол №2 от 04.03.2014 г.

КАФЕДРА

Вычислительных систем

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

СТУДЕНТУ Крюковой Л.П. ГРУППЫ ИС-241

«УТВЕРЖДАЮ»

«_____» ___________ 20__г.

Зав. кафедрой

___________ /Мамойленко С.Н./

Новосибирск, 2016 г.

1. Тема выпускной квалификационной работы бакалавра
Оптимизация решения разреженных систем линейных уравнений на системах с общей памятью

утверждена указом СибГУТИ от «20» Февраля 2016 г. № 4/159о-16

2. Срок сдачи студентом законченной работы: «11» июня 2016 г.

3. Исходные данные к работе
1 Тьюарсон, Р. Разреженные матрицы : Науч. литература. – М.: Изд-во МИР, 1977. - с. ,
2 Iwashita T., Nakashima H., Takahashi Y. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. – 2012. – С. 474-483

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов)	сроки выполнения по разделам
Введение	15.05 – 17.05
Описание предметной области	17.05 – 1.06
Основная часть	1.06 – 10.06
Заключение	10.06

Дата выдачи задания «_____» ______________ 20__г.

Руководитель ____
подпись

Задание принял к исполнению «_____» ______________ 20__г.

Студент ____

АННОТАЦИЯ

бакалаврской работы студента Крюковой Л.П.
по теме «Оптимизация решения разреженных систем линейных уравнений на системах с общей памятью»

Объём бакалаврской работы 59 страниц, на которых размещены 23 рисунков и 2 таблица. При написании работы использовалось 11 источников.

Ключевые слова: разреженные матрицы, метод Гаусса.

Работа выполнена на Кафедре ВС СибГУТИ.
Руководитель –к.ф.-м.н. доцент Пудов С.Г.

Целью бакалаврской работы была разработка и анализ параллельного алгоритма для решения разреженных систем линейных алгебраических уравнений треугольного вида на машинах с общей памятью.

Разреженные матрицы имеют очевидное практическое применение, занимая важный раздел математического анализа. Их используют для инженерных и научных задач.

Особое внимание сейчас уделяют проблеме решения систем линейных уравнений $Ax = b$, где A разреженная матрица.

В рамках дипломного проекта был реализован параллельный алгоритм. Проведено эксперементальное исследование алгоритмов, базового и параллельного.

По результатам проведённых эксперементов выработаны рекомендации по выбору алгоритма в зависимости от компонент связности.

Содержание

1. ВВЕДЕНИЕ.......... 10
2. РАЗРЕЖЕННЫЕ МАТРИЦЫ.......... 12
2.1 Координатный формат хранения.......... 14
2.2 Разреженный строчный формат хранения.......... 15
2.3 Разреженный столбцовый формат.......... 15
3. РЕШЕНИЕ РАЗРЕЖЕННЫХ СЛАУ. 18
4. РАЗРАБОТКА ПАРАЛЛЕЛЬНОГО АЛГОРИТМА 22
4.1 Структуры данных 22
4.2 Анализ.......... 23
4.3 Решение 30
5. ЧИСЛЕННЫЕ ЭКСПЕРИМЕНТЫ 32
5.1 Чтение и конвертация матриц.......... 32
5.2 Валидация результатов 33
5.3 Тестирование производительности 34
5.4 Анализ результатов 36
6. ЗАКЛЮЧЕНИЕ 42
ПРИЛОЖЕНИЕ А 43
ПРИЛОЖЕНИЕ Б.......... 44
ПРИЛОЖЕНИЕ В 45

1. ВВЕДЕНИЕ

Разреженные матрицы имеют очевидное практическое применение, поскольку позволяют решать задачи больших размерностей на большинстве существующих вычислительных систем. Их используют для инженерных и научных задач, таких как:

- Оптимизационные задачи
- Численное решение дифференциальных уравнений
- Линейного и нелинейного математического программирования
- Теория графов
- Генетики
- Социологии
- И др.

Особое внимание сейчас уделяют проблеме решения систем линейных уравнений Ax = b, где A разреженная матрица. [1]

Большинство методов решения систем алгебраических линейных уравнений(СЛАУ) основаны на преобразовании матрицы с целью приведения её к более простой форме – диаганальной, треугольной и т д.

Выбор алгоритма зависит от самой матрицы и решаемой задачи. В некоторых приложениях систему с треугольной матрицей необходимо решать однократно, для этого достаточно использовать любую подходящую реализацию метода решения, учитывающую особенности формата хранения данных. Как правило такие реализации являются последовательными, поскольку форматы хранения разреженных матриц не позволяют делать эффективное распределение работы по процессорам «на лету», без предварительного анализа матрицы.

Наибольший интерес среди исследователей вызывают приложения, в которых решение систем треугольных уравнений с одной и той же матрицей происходит многократно. Например, к таким приложениям можно отнести итерационные методы решения СЛАУ с предобуславливанием, например метод Гаусса-Зейделя. В таких приложениях последовательная реализация решения треугольных систем линейных уравнений становится узким местом, в значительной степени определяющим время решения задачи. Для таких приложений имеет смысл провести анализ входной матрицы с целью поиска возможностей по распараллеливанию: поскольку каждая строка имеет лишь небольшое количество ненулевых элементов, то стоит поискать наборы строк, которые можно будет обрабатывать одновременно.

Время потраченное на анализ матрицы как правило превышает время на одно решение системы уравнений. Но это время может быть компенсировано за счет ускорения процесса решения путем распараллеливания и при большом числе таких вызовов. Погрешность в таких вычислениях не играет большой роли, главное возможность получения самих результатов при больших объемах данных. [1]

Целью дипломного проекта являлась разработка и анализ параллельного алгоритма для решения разреженных систем линейных алгебраических уравнений треугольного вида на машинах с общей памятью.

Не уменьшая общности для матрицы находится решение только для нижнего треугольника, при условии диаганального преобладания.

Решение задачи разделилось на подпункты:

- Реализация последовательного алгоритма решения разреженных СЛАУ на системах с общей памятью, как базового.
- Реализация параллельного алгоритма.
- Исследование времени работы параллельного алгоритма в сравнении с последовательным.
- Исследование времени выполнения работы алгоритма от структуры разреженной матрицы.

В следующих параграфах будут разобраны основные определения, методы хранения матриц, последовательная и параллельная реализация метода Гаусса на системах с общей памятью и результаты работы алгоритмов.

2. РАЗРЕЖЕННЫЕ МАТРИЦЫ

Матрицей называется прямоугольная таблица чисел, содержащая n строк одинаковой длинны. Матрица записывается в виде

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \quad (1)$$

Или сокращенно, $A=(a_{ij})$, где i = (1,n) - номер строки, j= (1,m) - номер столбца. [2]

На практике матрицы могут быть или плотными или разреженными. Разреженной матрицей называют матрицу в которой “много” нулевых элементов. Другое более распространенное определение, матрица разрежена, если она содержит большой процент нулевых элементов. На рисунке 1 изображена разреженная матрица.

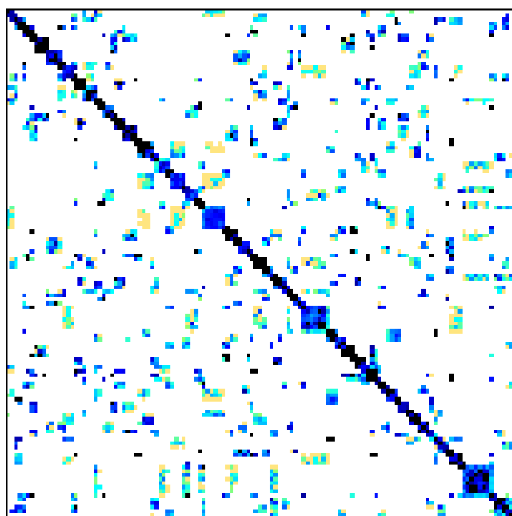

Рисунок 1- Пример портрета разреженной матрицы

Практически матрица является разряженной если она имеет O(N) нулевых элементов, где N*N – размерность матрицы. Разреженные матрицы отвечают за большой класс задач в реальном мире. Они встречаются в анализе предприятий, взаимосвязи сотрудников, генетике, теории графов, программирования, численном дифференцировании и во многих других. [8] На рисунке 2 приведен пример более плотной разреженой матрицы.

Очень часто многие задачи или не могут быть решены или решаются крайне долго из-за своих объемов. Если посмотреть на данные задачи в них встречаются разреженные матрицы, поэтому необходимо знать форматы хранения и основные алгоритмы.

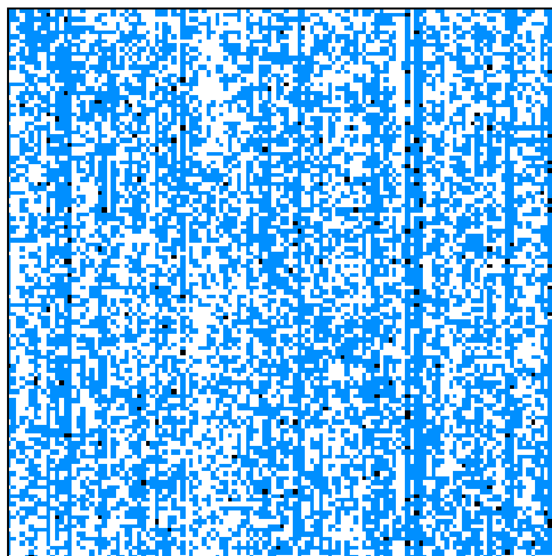

Рисунок 2 - Пример портрета разреженной матрицы

Очень важно знать структуру матриц. Для регулярных структур(ленточных и т д) матриц, существуют более быстрые форматы хранения и модицикации алгоритмов. [3]

Иногда структура матрицы, разреженность и задача не имеет смысла в применении особых форм и может быть посчитана стандартными методами для плотных матриц.

Поэтому перед использованием особых форм реализации, следует провести комплекстный анализ. Затрата по времени на анализ и специальный формат хранения будет оправдан для набора задач с однотипным расположением нулей.

Специальный формат хранения, это упакованные матрицы. Упакованный формат хранения подразумевает хранение только не нулевых элементов и их местоположение. [4]

Псюсы упакованного формата хранения очевидны.

- Матрицы больших размеров смогут быть вмещенны в память ЭВМ, если не занимать место нулевыми элементами.
- Матрицы могут быть на столько большие, что даже при этом весьма сжатом формате не будут помещаться в память, придется использовать внешнюю память и подгружать новые элементы. Подгрузка значений из внешней памяти в операционную весьма медленный процесс, при размещении в упакованном формате, элементы будут просто записываться в конец, что весьма сокращает работу по поиску места в двумерном формате хранения.
- Исключается работа в алгоритмах с нулевыми элементами, многие циклы упраздняются.
- Можно оптимизировать формат хранения с целью экономии по памяти.

Почти для каждого особого вида матрицы существуют свои форматы хранения. Так же существуют обобщенные форматы, их рекомендации по хранению действительны для любого типа матрицы. Некоторые из этих форматов будут описаны ниже.

Предположим у нас есть разреженная матрица типа double, не нулевых элементов 20%, размером $n > 10^{10}$, хранить еще в двумерном формате не имеет смысла размер для хранения элементов составит n*n*sizeof(double), где sizeof(double) = 8байт.

Данный объем элементов будет занимать много место в памяти не давая ни какой полезной информации. Из условий задачи значащих элементов 20%, тоесть из 8*n2 полезный объем составит: $8*0.2*n^2 = 1.6*n^2$.

2.1 Координатный формат хранения

Самый очевидный формат для хранения только не нулевых элементов, это коррдинатный. При такой упаковке, в случае нашего примера вы создаете 3 массива длинной 0.2*n.

Первый массив values будет хранить только не нулевые элементы, второй массив cols хранит позицию в колонке данного элемента и третий массив rows будет хранить позицию в строке.

Плюсом данного метода является не упорядоченный формат хранения элементов, удобная и понятная запись всем. Чаще всего данным методом матрицы запаковывают в случае быстрого и удобного хранения в файле.

Основным из минусов этого метода является долгий поиск элемента, в следствии не упорядоченных по строкам или столбцам элементов. Избыточное хранение одинаковых элементов (индекса строки или столбца).

Пример упаковки данным форматом:

Данную матрицу (рисунок 3) легко можно представить в координатном формате. Для более наглядного примера я упорядочу элементы по строкам.

Values = {5, 1, 1, 2, 3, 4, 1, 4, 4, 1, 3, 7, 2, 7, 1, 2, 2}

Cols = {0, 1, 0, 2, 3, 4, 0, 2, 5, 0, 2, 6, 0, 1, 7, 3, 8}

Rows = {0, 1, 2, 2, 3, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 8, 8}

Если данные заранее не упорядочить, то для того что бы найти элемент в 0 строке и в 5 столбце, или понять, что его нет, может понадобиться пройти по всех элементам до конца, потратив O(k) времени, где k-размерность одного из массивов. Размеры всех трех массивов одинаковые.

На рисунке 3 изображена разреженная матрица.

$$
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\
1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\
1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\
2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2
\end{pmatrix}
$$

Рисунок 3 - Пример разреженной матрицы 9*9

Для уменьшения времени на поиск в программах используют другие форматы хранения, применяя конвертеры.

2.2 Разреженный строчный формат хранения

Строковые алгоритмы используют разреженный сройчный формат (Compressed Row Storage (CRS)). CRS использует так же три массива, но у него нет избыточности по элементам. В данном формате элементы считываются по строкам и в третий массив записывается интекс первого элемента. [9]

Во время поиска, можно будет пройтись от первого элемента данной строки, до первог элемента следующей, потратив на поиск O(p) времени, где p-разментность i строки.

Формат как говорилось ранее так же использует три массива values, cols и pointers. Быстрый поиск по строке удобен для таких алгоритмов как:

- Метод Гаусса
- Умножение матрицы на вектор

Пример упаковки данным форматом:

Используя данные матрицы изображенной на рисунке 2. Представим матрицу в CRS формате.

Values = {5, 1, 1, 2, 3, 4, 1, 4, 4, 1, 3, 7, 2, 7, 1, 2, 2}

Cols = {0, 1, 0, 2, 3, 4, 0, 2, 5, 0, 2, 6, 0, 1, 7, 3, 8}

Pointers = {0, 1, 2, 4, 5, 6, 9, 12, 15, 17}

2.3 Разреженный столбцовый формат

Для некоторых алгоритмов удобнее обход по столбцам, они используют разреженный стобцовый формат (Compressed Sparse Column (CSC)). CSC отличается от CRS только тем, что вместо массива позиций столбцов используются позиции строк, а массив указателей указывает на начало столбца. [9]

В таком формате быстрый поиск элемента будет осуществляться по столбцам.

Алгоритмы использующие данный метод упаковки:

- Транспонированное умножение матрицы на плотный вектор
- Умножение разреженной матрицы на разреженный вектор

Пример упаковки данным форматом:

Используя данные матрицы изображенной на рисунке 2. Представим матрицу в CSC формате.

Values = {5, 1, 1, 2, 3, 4, 1, 4, 4, 1, 3, 7, 2, 7, 1, 2, 2}

Rows = {0, 2, 5, 6, 7, 1, 7, 2, 5, 6, 3, 8, 4, 5, 6, 7, 8}

Pointers = {0, 1, 2, 4, 5, 6, 9, 12, 15, 17}

Для алгоритмов которые часто порождают ненулевые элементы, чаще всего эти методы хранения не используются. Для них удобно использовать связные списки. [9]

Для обращения к массивам тратится меньше времени из-за их последовательного расположение в памяти. Но добавлеине в произвольную позицию элемента в связно списке, занимает меньше времени, благодоря перебрасыванию указателей.

Пример упаковки данным форматом:

Данный пример будет показан не на конкретных числах. Представление данного формата лучше изображать графически. Структура элементов в программе можно посмотреть на листинге 1.

Листинг 1 – Структура элемента списка

```
struct matrix {
     int element;
     int i;
     struct matrix *next_j;
};
struct matrix values[k];
```

В данной структуре element это значение ненулевого элемента в матрице, i индекс его строки, next_j, указатель на следующий элемент в данной строке. (рисунок 4)

Массив структур values размером k, где k это количество строк в матрице.

Таким образом если необходимо вставить элемент в конкретную строку, мы находим нужную структуру и меняем в ней указатели. (рисунок 5)

Рисунок 4 - Связный список

То есть для того что бы в эту структуру добавить новый элемент просто необходимо поменять указатель на следующий.

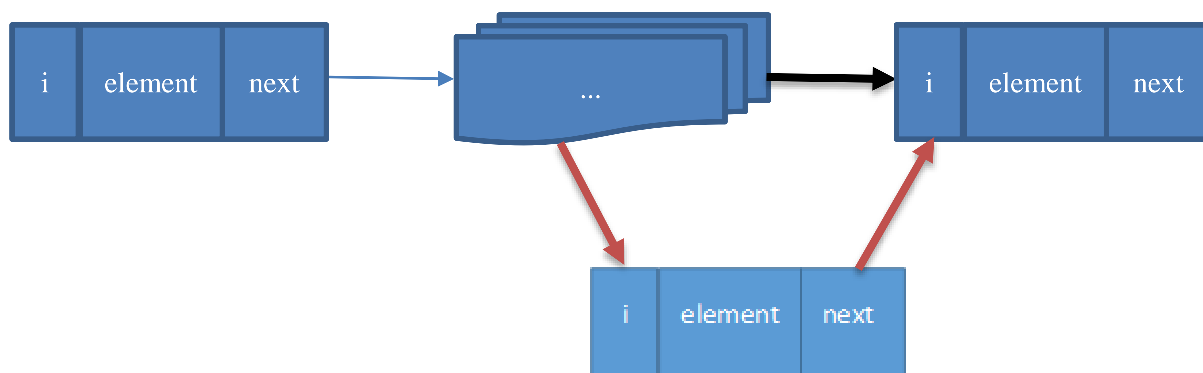

Рисунок 5 - Добавление в связный список

Перестановка указателей выполняется за O(1), бывший указатель(черный) стирается, теперь он указывает на новый элемент, а указатель нового элемента указывает на следующий старый элемент.

Вставка элемента в данную структуру аналогична вставки элемента в связный список.

Используя в работе лишь метод Гаусса, для данной работы применялся координатный формат и CRS. Базовым форматом для алгоритма Гаусса, стал формат CRS, как строковый.

3. РЕШЕНИЕ РАЗРЕЖЕННЫХ СЛАУ.

Метод Гаусса – классический метод решения систем линейных алгебраических уравнений (СЛАУ). Наиболее известный метод в своём разделе Алгебры. Этот метод последовательного исключения переменных, решение достигается элементарными преобразованиями строк и столбцов матрицы, получая на выходе едeничную матрицу и вектор-ответов справа.

В данной работе, не уменьшая общности далее рассматривается только нижний треугольник с не нулевой главной диаганалью. Решение для верхнего треугольника и плотных матриц можно получить аналогичным образом.

Решение уравнений можно представить в виде AX = Y, где Y это правая часть уравнения, X – вектор решений.

Для матрицы A существует условия диагонального приобладания, так как для плохих собственных чисел метод Гаусса будет не целесообразен к использованию. В случае диагонального преобладания, плохих собственных чисел не получится и метод Гаусса будет применим.

Наглядное представление решения и вывод формул. (рисунок 6)

$$\left(\begin{array}{ccc|c} a_{11} & 0 & 0 & y_1 \\ a_{21} & a_{22} & 0 & y_2 \\ a_{31} & a_{32} & a_{33} & y_3 \end{array}\right)$$

Рисунок 6 - Общий вид матрицы 3*3 с правой частью

Для рисунка 6 можно вывести формулы по вычеслению x_i.

$$x_1 = \frac{y_1}{a_{11}} \tag{2}$$

$$x_2 = \frac{y_2 - x_1 a_{21}}{a_{22}} \tag{3}$$

$$x_3 = \frac{y_3 - x_1 a_{31} - x_2 a_{32}}{a_{33}} = \frac{y_3 - \sum_{j=1}^{2} x_j a_{3j}}{a_{33}} \tag{4}$$

Следуя из данных формул, можно представить матрицу общего вида и записать для нее решение.

$$\begin{pmatrix} a_{11} & 0 & 0 & | & y_1 \\ \vdots & \ddots & 0 & | & \vdots \\ a_{n1} & \dots & a_{nm} & | & y_n \end{pmatrix}$$

Рисунок 7 - общий вид матрицы

Тогда общее решение для любого x_i, где i = (1, n) примет вид:

$$x_i = \frac{y_i - \sum_{j=0}^{i-1} x_j a_{ij}}{a_{ii}} \quad (5)$$

При данных условиях, на кладываемых на задачу, диаганальные элементы всегда есть, это исключает деление на ноль и каждый x зависит лишь от тех элементов которые стоят до диаганального и на предыдущих шагах просчитываются. Вычислительная сложность данного алгоритма O(n*m). На листинге 3 по формуле 5 реализуется метод Гаусса.

```
        for(unsigned int j = mtx.getCol(i);
                    j < mtx.getCol(i+1) - 2; ++j)
                sum += mtx.getValue(j)
                         * answer[mtx.getRow(j)];
        answer.push_back((mtx.getValue(mtx.getCol(i+1) - 1)
                          -             sum)              /
(mtx.getValue(mtx.getCol(i+1) - 2)));
    }
    return answer;
}
```

Листинг 3 – последовательная реализация метода Гаусса

```
Template <class T>
vector<T> algorithm_math(CRS<T> &mtx) {
    vector<T> answer;
    answer.reserve(mtx.sizeGaus());
    T sum = 0.0;
    for(unsigned int I = 0; I < mtx.getColSize() - 1; ++i)
{
        sum = 0.0;
```

Разбор примера по листингу 3 разберем по данным с рисунка 8.

Представим нашу матрицу в упакованном формате. Для упаковки используется формат CRS, о котором говорилось ранее, правую часть тоже представим как еще один столбец матрицы.

$$\left(\begin{array}{ccccccccc|c} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & -2 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 & 1 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 & 3 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 & 9 \end{array}\right)$$

Рисунок 8 - общий вид матрицы

Values = {5, 5, 1, 4, 1, 2, 2, 3, 7, 4, -2, 1, 4, 4, 1, 1, 3, 7, 3, 2, 7, 1, 7, 2, 2, 9}

Cols = {0, 9, 1, 9, 0, 2, 9, 3, 9, 4, 9, 0, 2, 5, 9, 0, 2, 6, 9, 0, 1, 7, 9, 3, 8, 9}

Pointers = {0, 2, 4, 7, 9, 11, 15, 19, 23, 26}

Начиная с i = 0 произведем поиск всех x_i по нашему алгоритму.

Для поиска первого элемента достаточно будет поделить правую часть на диаганальный.

$$x_0 = \frac{5}{5} = 1 \tag{6}$$

Из матрицы видно что X_1 не зависит от X_0 из этого следует, что он будет посчитан аналогичным образом.

$$x_1 = \frac{4}{1} = 4 \tag{7}$$

Второй элемент зависит от нулевого, необходимо посчитать сумму, которую следует вычесть из правой части.

$$sum = x_0 a_{20} = 1 * 1 = 1 \tag{8}$$

После подсчета суммы выполним действия:

$$x_2 = \frac{2 - sum}{2} = \frac{2 - 1}{1} = 0.5 \tag{9}$$

Выполняя данные преобразования последовательно для каждой из строк матрицы будут получены следующие результаты.

$$x_3 = \frac{7}{4} = 1.75 \qquad x_4 = \frac{-2}{4} = -0.5$$
$$x_5 = \frac{1 - 1 * 1 - 4 * 0.5}{4} = -0.5 \qquad x_6 = \frac{3 - 1 * 1 - 3 * 0.5}{7} = \frac{1}{14} \tag{10}$$

$$x_7 = \frac{7 - 2 * 1 - 7 * 4}{1} = -23 \qquad x_8 = \frac{9 - 2 * 1.75}{2} = 2.75$$

Из определения резреженных матриц и реальных размеров задач следует, что одна строка вероятнее всего будет независить от множества других строк, при данном условии велика вероятность обработки множества строк параллельно, что существенно уменьшает время работы.

4. РАЗРАБОТКА ПАРАЛЛЕЛЬНОГО АЛГОРИТМА

Реальные задачи представляют собой большие разреженные матрицы, для которых имеет смысл использовать параллельные алгоритмы решения, с целью ускорения получения правильного результата. [5]

Идея параллельного алгоритма заключается в предварительном анализе матрицы, с целью разбиения её на подмножество независимых строк для запуска их параллельно.

Чтобы реализовать алгоритм анализа, необходимо выбрать такую структуру данных, которая была бы удобна для хранения зависимостей, имела максимально эффективные по скорости алгоритмы добавления, удаления и поиска элемента.

4.1 Структуры данных

Для реализации параллельного алгоритма была выбрана биномиальная куча. Биномиальные кучи строются на основе биномиальных деревьев. [7]

Биномиальное дерево B_k - это рекурсивно определяемое дерево высоты k, в котором:

- 2*k – узлов в дереве
- i = [0, k] – колличество узлов на одном уровне
- Предок имеет k дочерних узлов

Биномиальная куча – это множество деревьев удовлетворяющих свойстам биномиального дерева. Биномиальная куча удовлетворяет следующим свойстам: [7]

- Биномиальное дерево должно быть упрядоченно max-heap или min-heap, тоесть ключи предка в таком дереве должными быть или меньше(min-heap) или больше(max-heap) ключей своих дочерних узлов.
- для любого целого $k \geq 0$ имеется не более одного биномиального дерева, чей корень имеет степень k

Узел биномальной кучи состоит из слудеющих полей:

- key – приоритет,
- value – значение,
- degree – количество дочерних узлов,
- parent – указатель на родительский узел,
- child – указатель на левый дочерний узел,
- sibling – указатель на сестринский узел.
- flag – флаг присваивания дочернего узла.

Построения дерева зависимостей, дает существенное понимание о строении матрицы, колличестве независимых элементов, колличестве дочерних узлов и структуре связей зависимости. Для параллельного алгоритма, постоянный обход дерева и поиск независимых элементов, удаление на каждой итерации большого колличества вершин будут весьма не эффективны. Для исключения этой неэффективности, проводился второй анализ, анализ дерева.

Второй анализ заключается в анализе дерева и определении компонент связности. Для данного анализа мы используем структуру векторов.

Вектор – это коллекция объектов какого-либо типа, к каждому из которых есть доступ по индексу. Вектор является контейнеов потому что может содержать любые типы, типом может выступать другой объект. Добавление элементов в вектор производится вункцией данного шаблона push_back(element). Для данного анализа вектор был проинициализирован объектами типа вектор, который в свою очередь проинициализирован типом double, типом элементов в матрице.

В следующем подпункте, будет разобран конкретный пример и описано пошаговое выполнение анализа.

4.2 Анализ

Для наглядного представления анализа, приведем пример реализации в картинках. На рисунке 9 изображена разряженная матрица, для которой будет выполнен анализ.

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Рисунок 9 – Разреженная матрица

Шаг 1. Поиск элементов независящих от других, результат шага 1 представлен на рисунке 10.

Элементы которые подсвечены голубым на рисунке 10, на первом шаге для всех строк от них зависящих будут просчитаны. Повторяем Шаг 1 пока не будет просчитана вся матрица, результат повторорения шага 1, представлен на рисунке 11, является шагом 2.

После шага 2 видно, что просчитаны все зависимые элементы и может быть досчитана матрица, результат шага 3 представлен на рисунке 12.

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Рисунок 10 - Результат поиска независимых элементов. Шаг 1

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Рисунок 11 - Результат поиска независимых элементов. Шаг 2

После реализации шага 3, анализ матрицы заканчивается, разными цветами подсвечены строки, которые можно считать параллельно, сперва все голобые, потом красные и наконец желтые. Реализация данного анализа представлена ниже в листинге 4.

На листинге 5 преставлено добавление в биномиальную кучу.

На листинге 6 предсталена структураузла биномиального дерева.

$$
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\
1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\
1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\
2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2
\end{pmatrix}
$$

Рисунок 12 - Результат поиска независимых элементов. Шаг 3

Листинг 4 – реализация первого шага анализа

```
template <class T>
bmHeap<T> *initBmHeap(CRS<T> &mtx)
{
     bmHeap<T> *h = new bmHeap<T>();
     for(unsigned int I = 0; I < mtx.getColSize() - 1; i++)
     {
        h->bmheap_insert(I, (mtx.getValue(mtx.getCol(i+1) -
        2)), false);

        for(unsigned int j = mtx.getCol(i);
                    j < mtx.getCol(i+1) - 2; ++j)
              h->bmheap_insert(mtx.getRow(j),
              mtx.getValue(j), true);
    }
    return h;
}
```

Листинг 5 – добавление в биномиальную кучу

```
template <class T>
void bmHeap<T>::bmheap_insert(unsigned int key, T value,
bool is_child)
{
    bmNode<T> *nodest = new bmNode<T>(key, value, is_child);
    if (nodest == nullptr)
        return;
    if (heap == nullptr)
        heap = nodest;


     if(!is_child)
```

```
            bmheap_union(heap, nodest);
         else {
            node = nodest;
            bmheap_linktrees(nodest, this->bmheapMax());
        }
    }
```

Листинг 6 – представление структуры узла биномиального дерева

```
template <class T>
struct bmNode
{
    bmNode(unsigned int key, T value, bool is_child) :
        parent{nullptr}, child{nullptr}, sibling{nullptr},
        value{value}, degree{0},
        key{key}, is_child{is_child}
    {
    }

    bmNode<T> *parent;
    bmNode<T> *child;
    bmNode<T> *sibling;
    T value;
    unsigned int degree;
    unsigned int key;
    bool is_child;
};
```

Проведем анализ матрицы по листингу 4.

Дана матрица на рисунке 13, анализ проводится с 0 строчки алгоритма, вычислительная сложность анализа O(n2).

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Рисунок 13 – Разреженная матрица для анализа

Строчки которые не зависят от предыдущих столбцов, будут занесены в дерево без дочерних. Для каждой строчки формируется свое новое дерево. Илюстрация дерева приведена на рисунке 14.

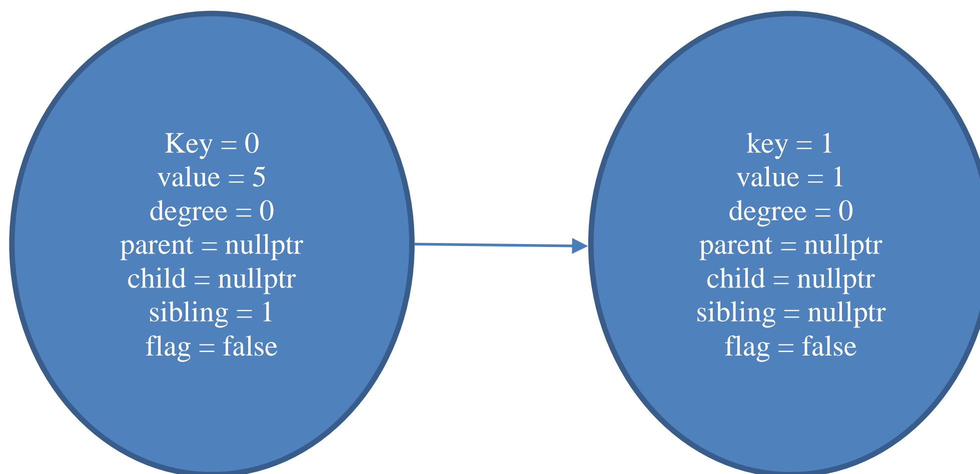

Рисунок 14 – Независимые узлы дерева

При анализе строчки i = 2 видна зависимость от строчки с индексом i = 0. Таким образом у узла key = 2 будет дочерний элемент с key = 1. Для изображения дерева впреть будет записывать только значение key. Вид дерева на второй итерации представлен на рисунке 15.

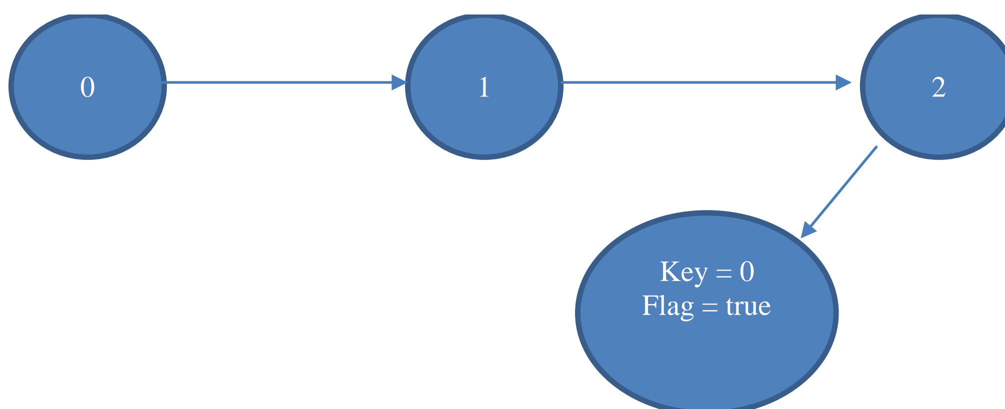

Рисунок 15 – Куча на второй итерации

Для всех дочерних узлов флаг будет равен true. По данным методам будет далее достроено дерево. Где в центре каждого узла написан ключ. Деревья сделаны по принципу max-heap. После выполнения всех шагов итерации куча примет вид представленный на рисунке 16.

Таким образом в верхнем ряду находятся элементы диагонали, а их дочерние, это элементы от которых они зависят. Если у узла нет дочерних, значит его можно высчитать и он ни от кого не зависит. Постоянно искать элементы в дереве и удалять их от туда очень накладно по времени работы. На анализ и так уходит большая чать времени.

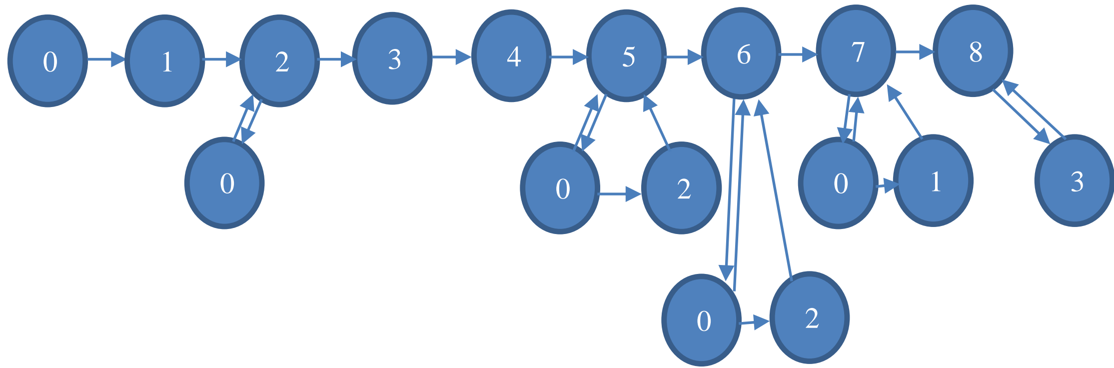

Рисунок 15 – Вид кучи после результата работы анализа

Пренебрегая временем анализа, мы гарантируем ускорение на классе задач.

Для того что бы все время не обходить дерево и не удалять элементы, произведем вторую часть анализа.

Второй анализ заключается в анализе дерева и определении компонент связности. Для данного анализа мы используем структуру векторов.

Вектор – это коллекция объектов какого-либо типа, к каждому из которых есть доступ по индексу. Вектор является контейнером, потому что может содержать любые типы, типом может выступать другой объект. Добавление элементов в вектор производится функцией данного шаблона push_back(element). Для данного анализа вектор был проинициализирован объектами типа вектор, который в свою очередь проинициализирован типом double, типом элементов в матрице.

Листинг 7 – Вторая часть анализа. Преобразование дерева.

```
Template <class T>
vector<vector<unsigned  int>>  analisThree(bmHeap<T>  *h,
CRS<T> &mtx) {
     vector<vector<unsigned int>> answer;
     vector<unsigned int> index_answer;
     unsigned int SizeVec = mtx.getColSize() - 1;
     bmNode<T> *x = nullptr;
     unsigned int I = 0;
     x = h->getHeap();
     while(x && (I < SizeVec - 1)) {
        for(; x ; x = x->sibling) {
            if(x->degree == 0)
                index_answer.push_back(x->key);
        }


          answer.push_back(index_answer);
          for(unsigned int ii = 0; ii < index_answer.size();
          ii++)
```

```
                h->bmheap_deleteKey(h->bmheapMax(),
                index_answer[ii]);
            x = h->getHeap();
            I += index_answer.size();
            index_answer.clear();
        }
        return answer;
    }
```

Приходя назад к нашему примеру с рисунка 15 следует разобрать биномиальную кучу по данному алгоритму и представить её в виде вектора векторов.

В каждом внутреннем векторе будут содержаться элементы которые можно будет считать парраллельно.

Во время прохода по биномиальной кучи следует найти элементы у которых нет дочерних. Во время первого прохода будет сформирован вектор: {0, 1, 3, 4}. Далее необходимо исключить из дерева добавленные элеметы, причем при удалении их необходимо удалять и из дочерних узлов. На рисунке 16 представлен вид кучи после данного шага.

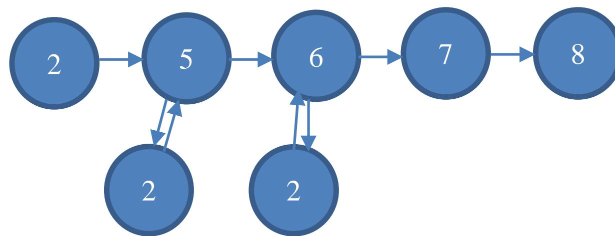

Рисунок 16 – Вид кучи после результата работы первого прохода

Производя те же шаги получится вектор {2, 7, 8}, куча примет вид как на рисунке 17.

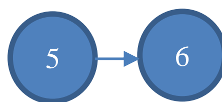

Рисунок 17 – Вид кучи после результата прохода n-1 шага

Во всем дереве дочерних элементов больше нет, в последний вектор поместятся все элементы данной {5, 6}.

Таким образом на выходе мы имеем ветор векторов “типа” элементов нашей матрицы, для данного примера это int.

{{0, 1, 3, 4}, {2, 7, 8}, {5, 6}} (11)

Теперь произведя полный анализ матрицы гарантируя диагональное преобладание мы можем решать методом Гаусса, распараллеливая наши вычисления.

4.3 Решение

Данный вектор из формулы 11 необходимо решать последовательно, но внутренние вектора, содержащие индексы строк в матрице, могут быть посчитаны параллельно.

Для параллельной реализации на машинах с общей памятью используется стандарт OpenMP.

OpenMP – открытый стандарт содержащий дерективы компилятора, библиотечные процедуры и переменные окружения для программирования многопоточных программ на системах с общей памятью. [6]

Ключевыми элементами является:

- Директива создания потоков parallel.
- Конструкции распределения работы между потоками do/for и section директивы.
- Конструкции для управления данными private и shared.
- Диррективы для синхронизации потоков такие как critical, atomic, barrier.
- Переменные окружения и процедуры библиотеки например такие как OMP_NUM_THREADS и omp_get_thread_num().

С помощью OpenMP была распараллелена последовательная версия программы. Стоит обратить внимание на индексацию, ведь теперь обход ведется не по строкам, а по элементам вектора, причем по элементам внешнего вектора обход последовательный, а по элементам внетрунних веторов обход параллельный.

OpenMP нам позволяет задавать колличество потоков используемых для распараллеливания. Таким образом ускорение программы будет в p раз, где p – количество потомков.

Реализация параллельной версии с применением OpenMP:

Листинг 8 – Реализация параллельного метода Гаусса.

```
template <class T>
vector<T>   parrallel_algorithm_math(vector<vector<unsigned
int>> s, CRS<T> &mtx) {
    unsigned int  ik = 0;
    unsigned int start, stop;
    unsigned int sz = s.size();
    unsigned int SizeVec = mtx.getColSize() - 1;
    vector<T> answer(SizeVec);


    #pragma omp parallel default(shared)
    {
      int i = 0;
      while( i < sz ) {

     #pragma omp barrier
     #pragma omp for nowait
          for(ik = 0; ik < s[i].size(); ++ik) {
```

```
                    T sum = 0.0;
                    start = mtx.getCol(s[i][ik]);
                    stop = mtx.getCol(s[i][ik] + 1) - 2;
                    for(unsigned int j = start; j < stop; ++j )
                         sum += mtx.getValue(j) *
                                   answer[mtx.getRow(j)];
                    answer[s[i][ik]] =
                         ((mtx.getValue(mtx.getCol(s[i][ik]+1) –
                              1) - sum) /
                         (mtx.getValue(mtx.getCol(s[i][ik] + 1) –
                              2)));
               }
               i++;
          }
          return answer;
     }
```

5. ЧИСЛЕННЫЕ ЭКСПЕРИМЕНТЫ

5.1 Чтение и конвертация матриц

Как говорилось ранее, в файле матрицы храняться в координатном формате. На листинге 9 представлена функция считывания из файла координатного формата хранения.

Метод Гаусса же использует строчный разреженный формат. Для преобразования координатного формата в строчный разреженный был реализован конвертер, который представлен на листинге 10. Важным замечанием является и то что, большие матрицы храняться в matrix market i/o (mtx) формате [10], для данного формата существует конвертер в любой другой формат файла. Но элементы внутри формата храняться в координатном формате не упорядоченно. Это не удобно для хранения в программе и будет плохо сказываться на поиске элемента для конкретной строки.

Листинг 9 – Чтение из файла матрицы

```
template <class T>
XY<T> read_xy(string nameFile)
{
    XY<T> myXY;
    ifstream input_file;
    unsigned int size_x(0);

    input_file.open(nameFile, std::ios::in);
    input_file >> size_x;

    int k = 0;
    if(size_x > 0) {
        for(unsigned int j(0); j < size_x && input_file;
                                                    ++j) {
            k++;
            T var, ind_i, ind_j;
            input_file >> ind_i >> ind_j >> var;
            myXY.addValue(var, ind_j, ind_i);
        }
    }
    return myXY;
}
```

Листинг 10 – Конвертер из координатного формата в разреженный строчный

```
template <class T>
CRS<T> converting_crs (XY<T> mtx)
{
    CRS<T> myCrs;
    vector<T> val;
    vector<unsigned int> row;
    unsigned int k(0), count(0);


    for(unsigned int i(0); k < mtx.sizeXY(); ++i) {
        if(k == mtx.getCol(i)) {
            val.push_back(mtx.getValue(i));
            row.push_back(mtx.getRow(i));
            count++;
        }
        else {
            myCrs.addLine(val, row, count);
            row.clear();
            val.clear();
            k++; i--;
        }
    }
    return myCrs;
}
```

Преобразовав и считав матрицу в нужном формате, реализуются функции решения. Листинг 8 предоставляет параллельное решение. Листинг 3 реализует последовательный вариант.

После получения ответа, производится его проверка и валидация.

5.2 Валидация результатов

На больших объемах данных валидировать в ручную становится не возможно. После решения мы имеем два вектора, для сравнения результатов, было принято решение посчитать норму разности векторов. В реальных задачах, в которых элементы внутри матриц не целые числа, может возникать погрешность. Для нашей задачи мы допускаем что норма разности векторов меньше чем 10^{-9}.

Норма разности векторов вычисляется как максимальный элемент в векторе. Реализация представлена на листинге 11.

Листинг 11 – Первый тест

```
template <class T>
vector<T> NormVector(vector<T> &a, vector<T> &b) {
    vector<T> norm;
    norm.reserve(a.size());



    for(unsigned int i = 0; i < a.size(); i++) {
        norm.push_back(a[i] - b[i]);
    }
```

```
    return norm;
}
```

Листинг 12 – Валидация

```
template<class T>
bool validation(vector<T> &a, vector<T> &b) {
    vector<T> norm = NormVector<T>(a, b);
    if(*(max_element(norm.begin(), norm.end())) < eps)
        return true;
    else
        return false;
}
```

5.3 Тестирование производительности

Для тестирования было разработано два теста.

Первый тест фиксирует ответ в массив answer и умножает его на матрицу без правой части, полученный ответ сравнивается с правой частью исходной системы. На листинге 13 представлена его реализация.

Листинг 13 – реализация первого теста

```
template <class T>
void testRight(CRS<T> &mtx) {
    vector<T> answer2;
    vector<T> answer3;
    bmHeap<T> *h = initBmHeap<T>(mtx);
    vector<vector<unsigned int>> MySets = analisThree<T>(h,
mtx);
    double tt1=getWallClockSeconds();
    answer2 = parrallel_algorithm_math<T>(MySets, mtx);
    double tt2=getWallClockSeconds();
    std::cout << "time parallel " << tt2 - tt1 << endl;
    tt1 =getWallClockSeconds();
    answer3 = algorithm_math<T>(mtx);
    tt2 =getWallClockSeconds();
    std::cout << "time base " << tt2 - tt1 << endl;

    if(answer2.size() != answer3.size()) {
        std::cout << "size dot right" << std::endl;
    }
    else
        result(answer2, answer3);
}
```

Продемонстрируем пошаговое решение данного теста на листинге 13. Answer = {1, 4, 0.5, 1.75, -0.5, -0.5, 1/14, -23, 2.75}.

Матрицу возьмем из рисунка 8. Произведем умножение матрицы на вектор. На рисунке 18 продемонстрированно умножение. При умножении мы получили вектор {5, 4, 2, 7, -2, 1, 3, 7, 9}. Теперь необходимо найти норму разности вектора-результата умножения и вектора правой части СЛАУ.

Норма разности векторов это по компонентное вычетание из первого вектора второй и поиск максимума в получившемся векторе. Произведя эти вычисления, мы получим 0. Это означает что погрешность данных вычислений на этой матрице равнялась 0.

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \end{pmatrix} \times \begin{pmatrix} 1 \\ 4 \\ 0.5 \\ 1.75 \\ -0.5 \\ -0.5 \\ 1/14 \\ -23 \\ 2.75 \end{pmatrix} = \begin{pmatrix} 5 * 1 \\ 1 * 4 \\ 1 * 1 + 2 * 0.5 \\ 4 * 1.75 \\ 4 * -0.5 \\ 1 * 1 + 4 * 0.5 + 4 * -0.5 \\ 1 * 1 + 3 * 0.5 + 7 * 1/14 \\ 2 * 1 + 7 * 4 + 1 * -23 \\ 2 * 1.75 + 2 * 2.75 \end{pmatrix}$$

Рисунок 18 – Умножение матрицы на вектор

Для второго теста заранее генерируется произвольный вектор ответов и после решение СЛАУ, решение должно совпадать с эти методом.

На шаге 1 мы генерируем произвольный вектор,смотрите листинг 15. На шаге 2 мы умножаем (листинг 16) результат шага 1 на матрицу и получаем вектор правых частей. На шаге 3 в СЛАУ замещается (листинг 17) правая часть, на ту, что получилась в предыдущем шаге. Решение происходит на шеге 4. Результат 4-го шага сравнивается на шаге 5 с результатом шага 1.

На листинге 14 показан второй тест.

Листинг 14 – Второй тест

```
template <class U, class T>
void testLeft(U &mtx) {
    vector<T> answer = genAnswer<T>(mtx.sizeGaus());
    vector<T> right = mulMatrix<U, T>(answer, mtx);
    replays<U, T>(mtx, right);
    testRight<U, T>(mtx);

    if(validation<T>(resultRight, answer))
        cout << "Good! " << endl;
    else
        cout << "Err! " << endl;
}
```

Листинг 15 – Генерация вектора ответов. Шаг 1

```
template <class T>
static vector<T> genAnswer(unsigned int count) {
        vector<T> right;
        right.reserve(count);
        std::mt19937_64 gen(time(0));
        std::uniform_real_distribution<double> uid(1,20);
        for(unsigned int i = 0; i < count; ++i) {
```

```
            right.push_back(uid(gen));
        }
        return right;
}
```

Листинг 16 – Умножение матрицы на вектор

```
template <class U, class T>
vector<T> mulMatrix(vector<T> &vec, U &mtx) {
    T sum = 0;
    vector<T> answer;
    answer.reserve(mtx.sizeGaus());
    for(unsigned int i = 0, k = 0; i < mtx.sizeGaus(); ++i)
{
        for(unsigned int j = 0; j < mtx.sizeGaus() + 1; ++j)
{
                if(mtx.getRow(k) ==
                        mtx.getRow(mtx.getRowSize() - 1)) {
                    answer.push_back(sum);
                    sum = 0;
                    k ++;
                    j = -1;
                    break;
                }
                T val;
                if((val = mtx.gausValue(i, j)) != 0) {
                    sum += val*vec[j];
                    k++;
                }
        }
    }
    return answer;
}
```

Листинг 17 – Замена правой части в СЛАУ

```
template <class U, class T>
void replays(U &mtx, vector<T> &right)
{
    for(unsigned int i = 0; i < mtx.sizeGaus(); ++i)
        mtx.setValue(mtx.gausPos(i,
                mtx.getRow(mtx.getRowSize()      -     1)),
right[i]);
}
```

5.4 Анализ результатов

Для воспроизведения программы были взяты реальные матрицы на сайте университета Флориды, во вкладке коллекция разряженных матриц. Для реализация были взяты матрицы приведенные в таблице 1.

Матрицы были преобразованы, для соответствия условиям алгоритма: была востановлена диагональ, исключен верхний треугольник и восстановлено диагональное преобладание.

Результаты запуска работы и анализа программы можно посмотреть в таблице 1 и таблице 2.

Таблица 1- результаты запуска программы

Имя матрицы	Adder_deop_02	Adder_deop_02	Adder_deop_66	Adder_deop_66
Плотность матрицы	7708	7708	7708	7708
Степень распараллеливаемости	13	13	13	13
Колличество зависимых	1812	1812	1812	1812
Колличество потоков	8	16	8	16
Время последовательной	0.000247727	0.00024797	0.000260562	0.000263631
время параллельной	0.000197219	0.00017474	0.000174703	0.000172887
Ускорение	0.76	0.79	0.67	0.68
Время анализа	0.133027	0.134181	0.13339	0.132759
Эффективность	0.095	0.049	0.084	0.0425

Таблица 2 - результаты запуска программы

Имя матрицы	Adder_deop_02	Adder_deop_66	Bcircuit
Плотность матрицы	7708	7708	291759
Степень распараллеливаемости	13	13	358
Колличество зависимых	1812	1812	68902
Колличество потоков	8	8	8
Время последовательной	0.000247727	0.000260562	0.0101779
время параллельной	0.000197219	0.000174703	0.00998043
Ускорение	0.7961	0.65	0.9
Время анализа	0.133027	0.13339	452.222
Эфективность	0.09951	0.08125	0.1125

По данным полученым с результатов приведенных в таблице 1 были построены графики изображенные на рисунке 18, на рисунке 19 и на рисунке 20.

Рисунок 18 показывает время решения разных матриц в зависимости от количества потоков. Рисунок 19 показывает ускорение в зависимости от количества потоков. Рисунок 20 показывает эффективность в зависимости от количества потоков.

По данным полученым с результатов приведенных в таблице 2 были построены графики изображенные на рисунке 21, на рисунке 22 и на рисунке 23.

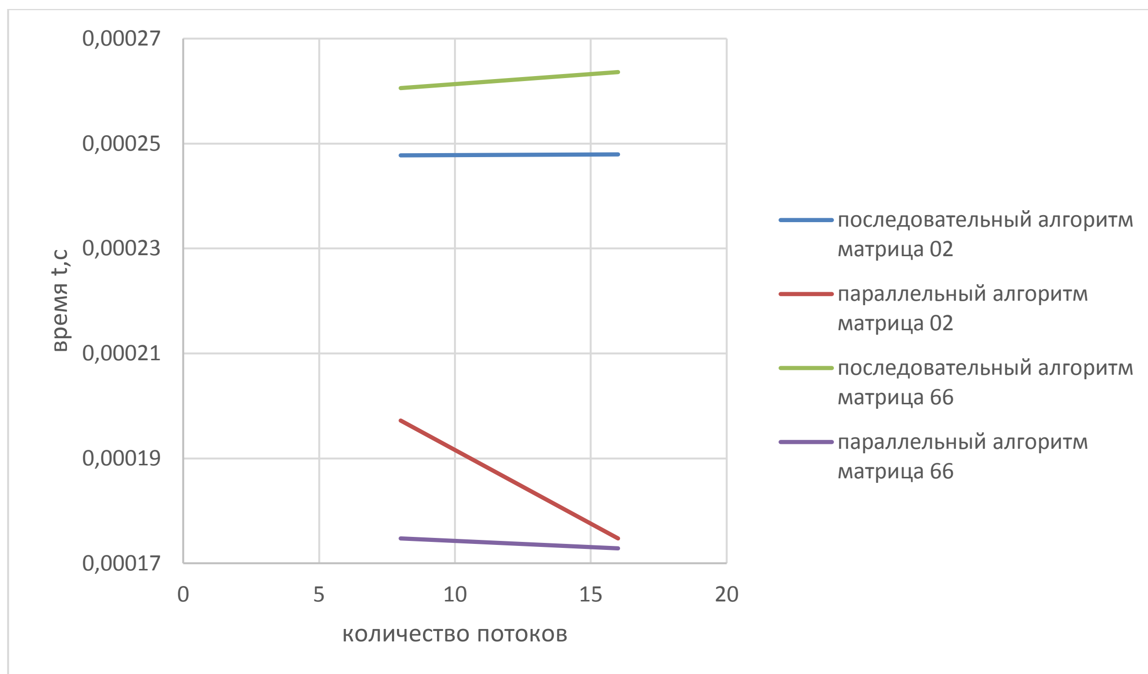

Рисунок 18 - Исследование времени работы алгоритма в зависимости от колличества потоков

Рисунок 21 показывает время решения разных матриц в зависимости от количества элементов в матрице. Рисунок 22 показывает эффективность в зависимости от количества элементов. Рисунок 23 показывает ускорение в зависимости от количества элементов в матрице.

Эфективность напрямую зависит от числа потоков, поэтому при увеличении потоков эфективность падает. На рисунке 20 показана эфективность работы алгоритма для матриц одинакового размера при разном колличестве потоков. Так как матрицы были выбраны не большие, эфективность снижается за счет не эфективного использования памяти.

На рисунке 19 видно, что ускорение для матриц одинакового размера растет с увеличением числа потоков. Ускорение зависит от структуры матрицы, рисунок 23. В некоторых случаях оно может быть отрицательным.

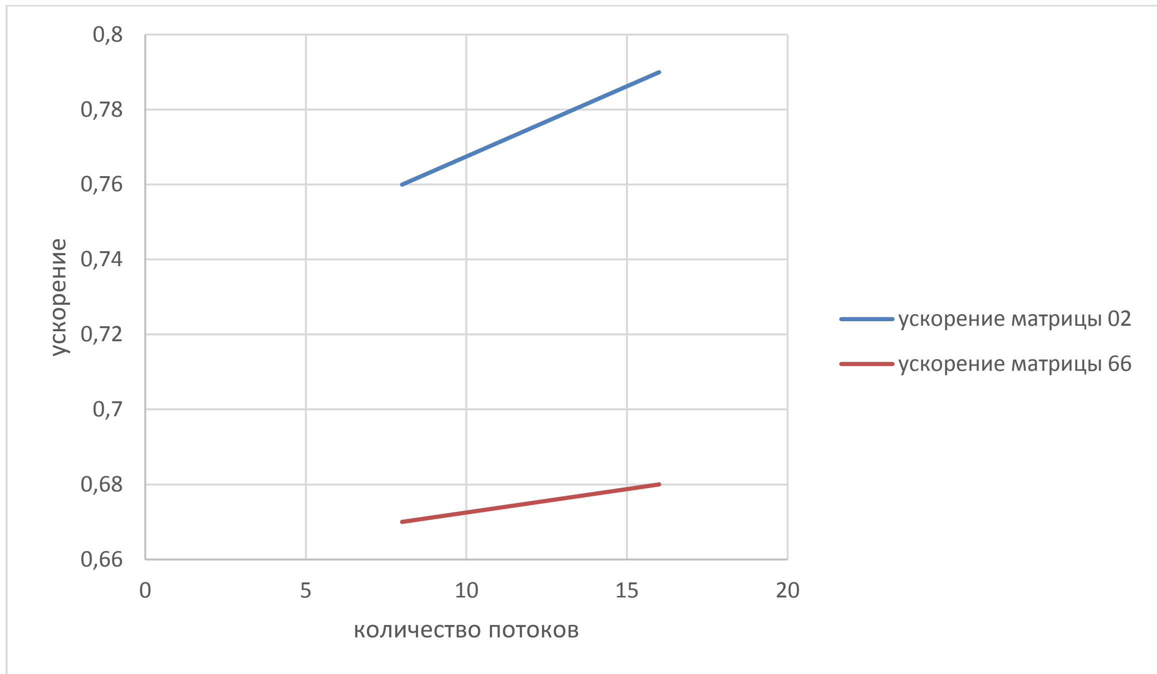

Рисунок 19 - Исследование ускорения от количества потоков

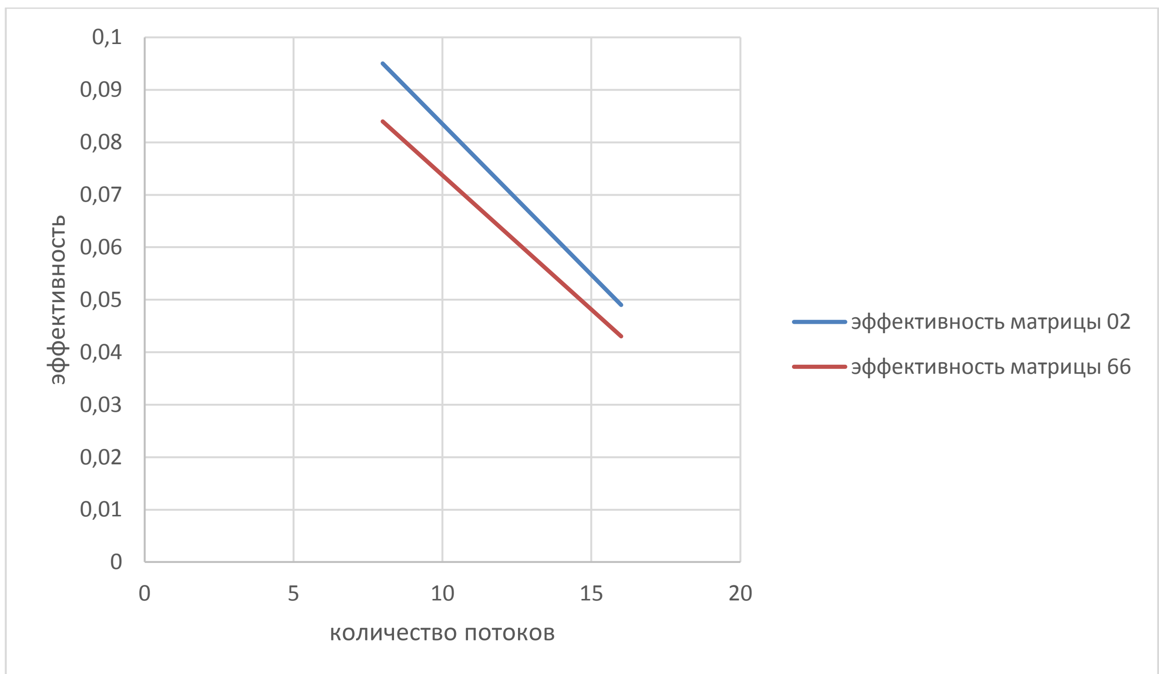

Рисунок 20 - Исследование эффективности в зависимости от количества потоков

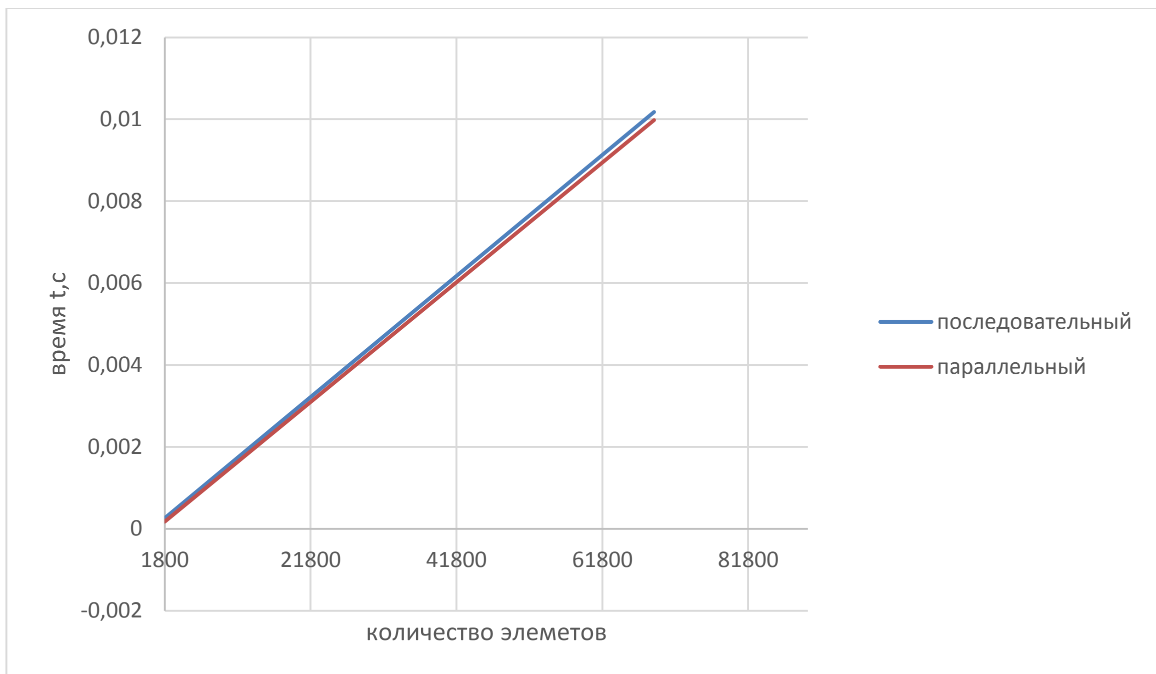

Рисунок 21 - Исследование времени работы в зависимости от количества элементов в матрице

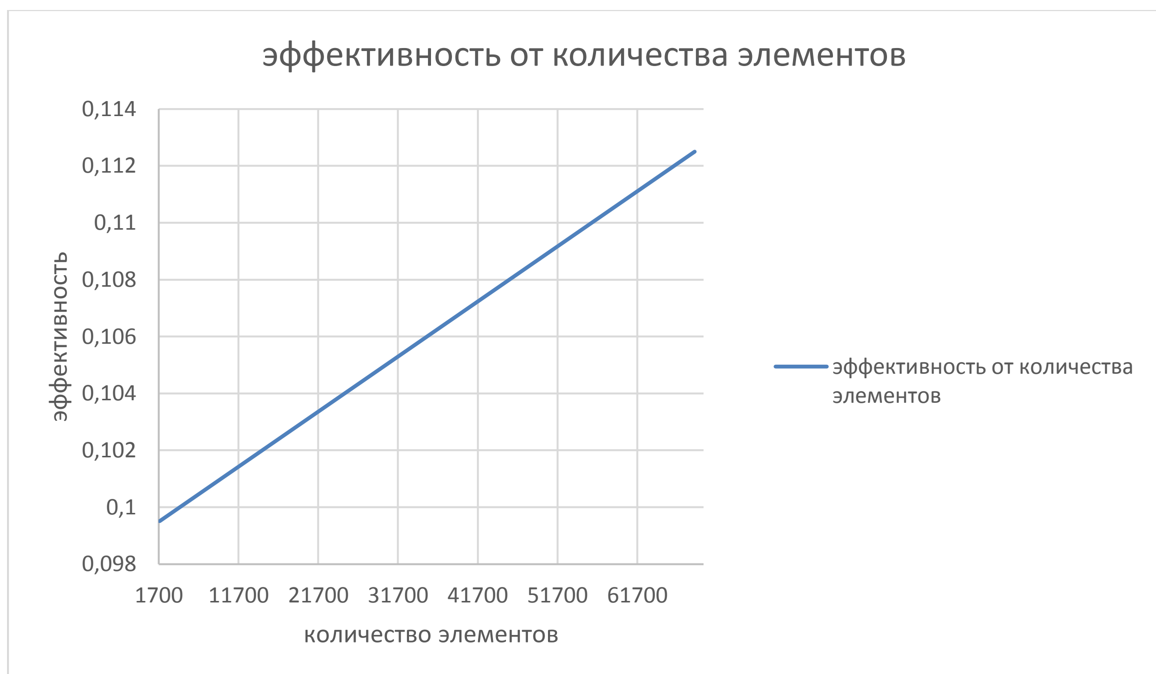

Рисунок 22 - Исследование эффективности от количества элементов в матрице

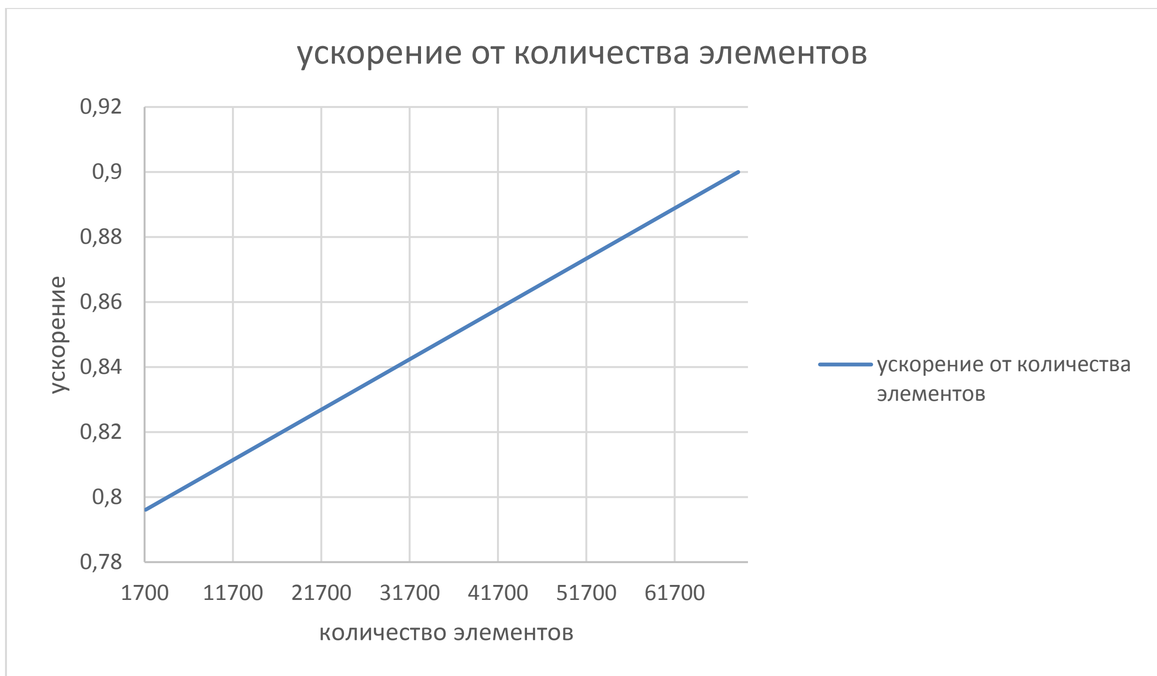

Рисунок 23 - Исследование ускорения в зависимости от количества элементов в матрице

6. ЗАКЛЮЧЕНИЕ

В рамках данного дипломного проекта было реализовано:

- Последовательный алгоритм решения треугольных разреженных СЛАУ при условии диагонального преобладания.
- На базе последовательного алгоритма был реализован параллельный алгоритм на системах с общей памятью.

В работе использовалась технология OpenMP, были изучены:

- метод Гаусса,
- методы валидации больших матриц,
- методы хранения разреженных систем.

По результатам данной работы были проведены исследования, построены графики ускорения, эффективности и времени работы алгоритмов.

Были сделаны выводы, что ускорение не зависит от числа элементов в матрице, оно непосредственно зависит от структуры разреженных матриц. Эффективность при увеличении числа потоков падает, но в случае одновременного увеличения числа потоков и размера матрицы, можно сохранить эффективность на прежнем уровне.

ПРИЛОЖЕНИЕ А

Библиография

1 Тьюарсон, Р. Разреженные матрицы : Науч. литература. – М.: Изд-во МИР, 1977. – 191 с.
2 Письменный, Д. Т. Конспект лекций по высшей математике: в 2 ч. Ч.1 [Текст] : учебник / Д. Т. Письменный. - 4-е изд. - М. : Айрис-пресс, 2004. - 280с.
3 Iwashita T., Nakashima H., Takahashi Y. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. – 2012. – С. 474-483
4 Mahesh V.J., Gupta A., Karypis G., Kumar V. A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems. – 6 с.
5 Alvarado F.L., Pothen A., Schreiber R. Highly parallel sparse triangular solution. – 18 с.
6 Курносов, М.Г. Лекция 6. Стандарт OpenMP. URL: http://www.mkurnosov.net/teaching/uploads/HPC/hpcs-fall2015-lec6.pdf (Дата обращения 11.04.2016)
7 Курносов, М.Г. Лекция 6. Биномиальные кучи (Binomial heaps). URL: http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall2013-lec6.pdf (Дата обращения 11.04.2016)
8 Интернет-Университет Информационных Технологий. Лекция 7. URL: http://www.intuit.ru/studies/courses/4447/983/lecture/14931 (Дата обращения 15.04.2016)
9 Мееров И.Б., Сысоев А.В., Сафонова Я., - Параллельные численные методы. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/LW_SparseMM_ppt.pdf (Дата обращения 20.05.2016)
10 Matrix Market. ANSI C library for Matrix Market I/O URL: http://math.nist.gov/MatrixMarket/mmio-c.html (Дата обращения 15.02.2016)
11 Алгебра и аналитическая геометрия: Теоремы и задачи [Текст] : [учеб. пособие] / Г. Д. Ким, Л. В. Крицков. - Москва : Планета знаний, 2007.Т. 1. - 2007. - 467, [3] с. : ил. -). - Предм. указ.: с. 460-465. - Указ. обозн.: с. 466-467. - ISBN 978-5-903242-01-6

ПРИЛОЖЕНИЕ Б

Наиболее употребляемые текстовые сокращения

СЛАУ – система линейных алгебраических уравнений
Mtx – Matrix Market I/O – формат для файлов с разреженными матрицами
CRS - Compressed Row Storage, разреженный строчный формат
СибГУТИ – Сибирский государственный университет телекоммуникаций и информатики
OpenMP – открытый стандарт для распараллеливания программ
CSC - Compressed Sparse Columns, разреженный столбцовый формат

ПРИЛОЖЕНИЕ В

Листинг программы

Листинг В.1 – xy.h

```
#ifndef XY_H
#define XY_H

#include <vector>
#include <istream>
#include "bmheap.h"

using std::vector;
using std::ifstream;
using std::string;

template <class T>
class XY
{
public:
    XY()
    {
    }

    T getValue(unsigned int ind) const
    {
        return value_[ind];
    }

    T getRow(unsigned int ind) const
    {
        return row_[ind];
    }

    T getCol(unsigned int ind) const
    {
        return col_[ind];
    }

    void addValue(T val, unsigned int row, unsigned int n)
    {
        col_.push_back(n);
        row_.push_back(row);
        value_.push_back(val);
    }

    unsigned int sizeXY()
    {
        return getCol(col_.size() - 1) + 1;
    }


private:
```

```
    vector<T> value_;
    vector<T> row_;
    vector<T> col_;
};

template <class T>
XY<T> read_xy(string nameFile)
{
    XY<T> myXY;
    ifstream input_file;
    unsigned int size_x(0);

    input_file.open(nameFile, std::ios::in);
    input_file >> size_x;

    int k = 0;
    if(size_x > 0) {
        for(unsigned int j(0); j < size_x && input_file;
++j) {
            k++;
            T var, ind_i, ind_j;
            input_file >> ind_i >> ind_j >> var;
            myXY.addValue(var, ind_j, ind_i); // value row
col
        }
    }
    return myXY;
}

#endif // XY_H
```

Листинг В.2 – crs.h

```
#ifndef CRS_H
#define CRS_H

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include "xy.h"

using std::vector;
using std::ifstream;
using std::string;
using std::cout;
using std::endl;



template <class T>
class XY;

template <class T>
```

```
class CRS
{
public:
    CRS()
    {
        col_.push_back(0);
    }

    CRS(const CRS& rht)
    {
        std::copy(rht.value_.begin(),     rht.value_.end(),
std::back_inserter(value_));
        std::copy(rht.row_.begin(),         rht.row_.end(),
std::back_inserter(row_));
        std::copy(rht.col_.begin(),         rht.col_.end(),
std::back_inserter(col_));
    }

    unsigned int getValueSize() const
    {
        return value_.size();
    }

    unsigned int getColSize() const
    {
        return col_.size();
    }

    unsigned int getRowSize() const
    {
        return row_.size();
    }

    T getValue(unsigned int ind) const
    {
        return value_[ind];
    }

    T getRow(unsigned int ind) const
    {
        return row_[ind];
    }

    T getCol(unsigned int ind) const
    {
        return col_[ind];
    }



   void setValue(unsigned int index, T value)
    {
        value_[index] = value;
    }
```

```
    void addLine(const vector<T> &val,const vector<unsigned
int> &row,
                   unsigned int n)
    {
        col_.push_back(n);
        std::copy(val.begin(),                   val.end(),
std::back_inserter(value_));
        std::copy(row.begin(),                   row.end(),
std::back_inserter(row_));
    }

    unsigned  int  gausPos(unsigned  int  i,  unsigned  int  j)
const
    {
       unsigned int k = 0;

       for(k = col_[i]; k < col_[i+1] && row_[k] != j; ++k);
       if(row_[k] != j || k == col_[i+1])
           return value_.size();
       return k;
    }

    T gausValue(unsigned int i, unsigned int j) const
    {
        const unsigned int index = gausPos(i, j);
        if(index == value_.size())
            return 0;
        else
           return getValue(index);
    }

    unsigned int sizeGaus() const
    {
        return getColSize() - 1;
    }

private:
    vector<T> value_;
    vector<unsigned int> row_;
    vector<unsigned int> col_;
};



template <class T>
CRS<T> converting_crs (XY<T> mtx)
{
    CRS<T> myCrs;
    vector<T> val;
    vector<unsigned int> row;
    unsigned int k(0), count(0);

    for(unsigned int i(0); k < mtx.sizeXY(); ++i) {
        if(k == mtx.getCol(i)) {
            val.push_back(mtx.getValue(i));
```

```
            row.push_back(mtx.getRow(i));
            count++;
        }
        else {
            myCrs.addLine(val, row, count);
            row.clear();
            val.clear();
            k++; i--;
        }
    }
    return myCrs;
}

#endif // CRS_H
```

Листинг В.3 – bmheap.h

```
#ifndef BMHEAP_H
#define BMHEAP_H

#include <iostream>

using namespace std;

template <class T>
struct bmNode
{
    bmNode(unsigned int key, T value, bool is_child) :
        parent{nullptr}, child{nullptr}, sibling{nullptr},
            value{value},       degree{0},       key{key},
is_child{is_child}
    {
    }

    bmNode<T> *parent;
    bmNode<T> *child;
    bmNode<T> *sibling;
    T value;
    unsigned int degree;
    unsigned int key;
    bool is_child;
};


template <class T>
class bmHeap
{
public:
    bmHeap () : heap(nullptr), node(nullptr)
    {
    }

    bmNode<T> *getNode() const;
    bmNode<T>* bmheapMax() const;
    bmNode<T> *getHeap() const;
```

```
    void bmheap_deleteKey(bmNode<T> *h, unsigned int key);
    void bmheap_insert(unsigned int key, T value, bool
is_child);

private:
    bmNode<T> *heap;
    bmNode<T> *node;
    void bmheap_union(bmNode<T> *a, bmNode<T> *b);
};

template <class T>
static void bmheap_linktrees(bmNode<T> *y, bmNode<T> *z)
{
    y->parent = z;
    y->sibling = z->child;
    z->child = y;
    z->degree++;
}

template <class T>
bmNode<T>* bmHeap<T>::bmheapMax() const
{
    bmNode<T> *maxNode = nullptr, *node;
    unsigned int maxkey = 0;

    for(node = heap; node != nullptr; node = node->sibling)
{
        if(node->key >= maxkey) {
            maxkey = node->key;
            maxNode = node;
        }
    }
    return maxNode;
}

template <class T>
bmNode<T> *bmHeap<T>::getHeap() const
{
    return heap;
}


template <class T>
static void bmheap_mergelists_general(bmNode<T> *a,
                                      bmNode<T> *b,
                                      bmNode<T> *head,
                                      bmNode<T> *sibling,
                                      bmNode<T> *end)
{
    end = head = nullptr;
    while (a != nullptr && b != nullptr) {
        if ((a->degree > b->degree) && (a->is_child)) {
            sibling = a->sibling;
            if (end == nullptr) {
                end = a;
```

```
                head = a;
            }
            else {
                end->sibling = a;
                /* Add to the end */
                end = a;
                a->sibling = nullptr;
            }
            a = sibling;
        }
        else {
            sibling = b->sibling;
            if (end == nullptr) {
                end = b;
                head = b;
            }
            else {
                end->sibling = b;
                /*Add to the end */
                end = b;
                b->sibling = nullptr;
            }
            b = sibling;
        }
    }
}


template <class T>
static bmNode<T>* bmheap_mergelists(bmNode<T> *a, bmNode<T>
*b)
{
    bmNode<T> *head = nullptr, *sibling = nullptr, *end =
nullptr;
    bmheap_mergelists_general(a, b, head, sibling, end);


    while (b != nullptr) {
        sibling = b->sibling;
        if (end == nullptr) {
            end = b;
        }
        else {
            end->sibling = b;
            end = b;
            b->sibling = nullptr;
        }
        b = sibling;
    }
    while (a != nullptr) {
        sibling = a->sibling;
        if (end == nullptr) {
            end = a;
        }
```

```
        else {
            end->sibling = a;
            end = a;
            a->sibling = nullptr;
        }
        a = sibling;
    }
    return head;
}

template<class T>
void bmHeap<T>::bmheap_union(bmNode<T> *a, bmNode<T> *b)
{
    bmNode<T> *h, *x, *nextx;

    h = bmheap_mergelists(a, b);
    if(h == nullptr) {
        if(a)
           heap = a;
        if(b)
           heap = b;
        if(!a && !b)
            heap = nullptr;
        return;
    }

    x = h;
    nextx = h->sibling;
    while(nextx != nullptr) {
        if((x->degree != nextx->degree) ||
                (nextx->sibling != nullptr &&
                nextx->sibling->degree == x->degree) ||
                !(nextx->is_child)) {
            x = nextx;
        }
        nextx = x->sibling;
    }
    heap = h;
}


template <class T>
void bmHeap<T>::bmheap_insert(unsigned int key, T value,
bool is_child)
{
    bmNode<T> *nodest = new bmNode<T>(key, value, is_child);
    if (nodest == nullptr)
        return;
    if (heap == nullptr)
        heap = nodest;
    if(!is_child)
        bmheap_union(heap, nodest);
    else {
        node = nodest;
        bmheap_linktrees(nodest, this->bmheapMax());
```

```
    }
}


template <class T>
bmNode<T>* bmHeap<T>::getNode() const {
    return node;
}

template <class T>
void bmHeap<T>::bmheap_deleteKey(bmNode<T> *h, unsigned int
key)
{
    bmNode<T> *x = nullptr, *prev = nullptr,
              *xmin = nullptr, *prevmin = nullptr,
              *child = nullptr, *sibling = nullptr;

    x = h;
    prev = nullptr;
    while(x != nullptr) {
         if (x->key == key) {
               xmin = x;
               prevmin = prev;

               if(prevmin != nullptr)
                    prevmin->sibling = xmin->sibling;
               else
                    h = xmin->sibling;


               child = xmin->child;
               prev = nullptr;

               while (child != nullptr) {
                   sibling = child->sibling;
                   child->sibling = prev;
                   prev = child;
                   child = sibling;
               }


               if(xmin->parent != nullptr) {
                   xmin->parent->degree--;
                   if(xmin->parent->child == xmin) {
                       xmin->parent->child = xmin->sibling;
                   }
               }

               delete xmin;
               bmheap_union(h, prev);
        }
        prev = x;
        bmheap_deleteKey(x->child, key);
        x = x->sibling;
```

```
    }
}


template <class T>
static void printHeapHelper(bmNode<T>* currNode)
{
    if(currNode == nullptr)
        return;

    // Print node num
    std::cout << "nodeNum= " << currNode->key;
    // Print dist
    std::cout  <<  ",  degree=  "  <<  currNode->degree  <<  ",
parent= ";
    // Print parent
    if (currNode->parent != nullptr)
        std::cout  << currNode->parent->key << ", children=
";
    else
        std::cout << "nullptr, children= ";
    // Print children
    if (currNode->child != nullptr)
        std::cout << currNode->child->key << ", sibling= ";


     else
        std::cout << "nullptr, sibling= ";
    //Print sibling
    if (currNode->sibling != nullptr)
        std::cout << currNode->sibling->key;
    std::cout << endl;
    // Print child nodes, then sibling nodes
    printHeapHelper(currNode->child);
    printHeapHelper(currNode->sibling);
}


template <class T>
void printHeap(bmNode<T>* root)
{
    if (root == nullptr)
    {
        std::cout << "Empty heap " << endl;
        return;
    }

    // Print minNode, trees, and maxRank
    std::cout << "maxNode= " << root->key << std::endl;

    printHeapHelper(root);
    cout << endl;
}
#endif // BMHEAP_H
```

Листинг В.4 – allfunc.h

```
#ifndef ALLFUNC_H
#define ALLFUNC_H
#include "bmheap.h"
#include <random>
#include <set>
#include <iomanip>
#include <algorithm>
#include <omp.h>
#include <assert.h>
#include "crs.h"

#define eps 0.0000001
template <class T>
bmHeap<T> *initBmHeap(CRS<T> &mtx);
template <class T>
vector<T> algorithm_math(CRS<T> &mtx);
template <class T>
vector<T>   parrallel_algorithm_math(bmHeap<T>   *h,CRS<T>
&mtx);


template <class U, class T>
void replays(U &mtx, vector<T> &right);

template <class T>
vector<T> NormVector(vector<T> &a, vector<T> &b);
// test when exam right part of matrix
template<class T>
bool validation(vector<T> &a, vector<T> &b);
template <class T>
vector<set<T>> analisThree(bmHeap<T> *h);


template <class T>
void result(vector<T> &a, vector<T> &b)
{
    if(validation<T>(a, b))
        cout << "Good! " << endl;
    else
        cout << "Err! " << endl;
}

long long int getWallClockNanos() {
    struct timespec tv;
        int errcode;

    errcode = clock_gettime(CLOCK_MONOTONIC,&tv);
    assert(errcode == 0);

    return 1000000000ll * tv.tv_sec + tv.tv_nsec;
}
```

```
double getWallClockSeconds() {
    return getWallClockNanos() / 1e9;
}

template <class T>
void testRight(CRS<T> &mtx) {
    vector<T> answer2;
    vector<T> answer3;
    double tt1 =getWallClockSeconds();
    bmHeap<T> *h = initBmHeap<T>(mtx);
    vector<vector<unsigned int>> MySets = analisThree<T>(h,
mtx);
    double tt2 =getWallClockSeconds();
    std::cout << "time ANALIS " << tt2 - tt1 << endl;


    float t2 = clock();
    tt1 =getWallClockSeconds();
    answer3 = algorithm_math<T>(mtx);
    t2 = clock() - t2;
    tt2 =getWallClockSeconds();
    std::cout << "time base " << t2 << ", " << tt2 - tt1 <<
endl;

    float t = clock();
    tt1=getWallClockSeconds();
    answer2 = parrallel_algorithm_math<T>(MySets, mtx);
    t = clock() - t;
    tt2=getWallClockSeconds();
    std::cout << "time parallel " << t << ", " << tt2 - tt1
<< endl;

    if(answer2.size() != answer3.size()) {
        std::cout << "size dot right" << std::endl;
    }
    else
        result(answer2, answer3);
}


template<class T>
bool validation(vector<T> &a, vector<T> &b) {
    vector<T> norm = NormVector<T>(a, b);
    if(*(max_element(norm.begin(), norm.end() - 1)) < eps)
        return true;
    else
        return false;
}


template <class T>
static vector<T> genAnswer(unsigned int count) {
        vector<T> right;
        right.reserve(count);
```

```
        std::mt19937_64 gen(time(0));
        std::uniform_real_distribution<double> uid(1,20);
        for(unsigned int i = 0; i < count; ++i) {
            right.push_back(uid(gen));
        }
        return right;
}


// mul vectorAnswer on left part of Matrix
template <class T>
vector<T> mulMatrix(vector<T> &vec, CRS<T> &mtx) {
    T sum = 0;
    vector<T> answer;
    answer.reserve(mtx.sizeGaus());
    for(unsigned int i = 0, k = 0; i < mtx.sizeGaus(); ++i)
{
        for(unsigned int j = 0; j < mtx.sizeGaus() + 1; ++j)
{
            if(mtx.getRow(k) == mtx.getRow(mtx.getRowSize()
- 1)) {
                answer.push_back(sum);
                sum = 0;
                k ++;
                j = -1;
                break;
            }
            T val;
            if((val = mtx.gausValue(i, j)) != 0) {
                sum += val*vec[j];
                k++;
            }
        }
    }
    return answer;
}


// test when setRandAnswer and mul with matrix
template <class T>
void testLeft(CRS<T> &mtx) {
    vector<T> answer = genAnswer<T>(mtx.sizeGaus());
    vector<T> right = mulMatrix<T>(answer, mtx);
    vector<T> answer3;
    replays<T>(mtx, right);
    testRight<T>(mtx);
 }


template <class T>
void replays(CRS<T> &mtx, vector<T> &right)
{
    for(unsigned int i = 0; i < mtx.sizeGaus(); ++i)
        mtx.setValue(mtx.gausPos(i,
```

```
                 mtx.getRow(mtx.getRowSize()     -     1)),
right[i]);
}


template <class T>
vector<T> algorithm_math(CRS<T> &mtx) {
    vector<T> answer;
    answer.reserve(mtx.sizeGaus());
    T sum = 0.0;
    for(unsigned int i = 0; i < mtx.getColSize() - 1; ++i)
{
        sum = 0.0;
        for(unsigned int j = mtx.getCol(i);
                    j < mtx.getCol(i+1) - 2; ++j)
               sum += mtx.getValue(j)
                        * answer[mtx.getRow(j)];
        answer.push_back((mtx.getValue(mtx.getCol(i+1) - 1)
                         -              sum)               /
(mtx.getValue(mtx.getCol(i+1) - 2)));
    }
    return answer;
}

template <class T>
vector<T>   parrallel_algorithm_math(vector<vector<unsigned
int>> s, CRS<T> &mtx) {
    unsigned int  ik = 0;
    unsigned int start, stop;
    unsigned int sz = s.size();
    unsigned int SizeVec = mtx.getColSize() - 1;
    vector<T> answer(SizeVec);
    //omp_set_num_threads(16);
    #pragma omp parallel default(shared)
    {
      //int num = omp_get_thread_num();
      unsigned int i = 0;
      while( i < sz ) {
    #pragma omp barrier
     #pragma omp for nowait
        for(ik = 0; ik < s[i].size(); ++ik) {
            T sum = 0.0;
            start = mtx.getCol(s[i][ik]);
            stop = mtx.getCol(s[i][ik] + 1) - 2;
            for(unsigned int j = start; j < stop; ++j )
                   sum += mtx.getValue(j) *
answer[mtx.getRow(j)];
      answer[s[i][ik]]                                 =
((mtx.getValue(mtx.getCol(s[i][ik]+1) - 1)
                             -           sum)             /
(mtx.getValue(mtx.getCol(s[i][ik] + 1) - 2)));
        }
          i++;
     }
    }
```

```
    return answer;
}



template <class T>
void show(CRS<T> &mtx) {
    cout << "Class matrix:" << endl;
    for(unsigned int i(0); i < mtx.getValueSize(); ++i)
        cout << mtx.getValue(i) << " ";
    cout << endl;
    for(unsigned int i(0); i < mtx.getRowSize(); ++i)
        cout << mtx.getRow(i) << " ";
    cout << endl;
    for(unsigned int i(0); i < mtx.getColSize(); ++i)
        cout << mtx.getCol(i) << " ";
    cout << endl;
}



template <class T>
bmHeap<T> *initBmHeap(CRS<T> &mtx)
{
    bmHeap<T> *h = new bmHeap<T>();
    for(unsigned int i = 0; i < mtx.getColSize() - 1; i++)
{
        h->bmheap_insert(i, (mtx.getValue(mtx.getCol(i+1) -
2)), false);

        for(unsigned int j = mtx.getCol(i);
                    j < mtx.getCol(i+1) - 2; ++j)
                h->bmheap_insert(mtx.getRow(j),
mtx.getValue(j), true);
    }
  //  printHeap(h->bmheapMax());
    return h;
}

template <class T>
vector<T> NormVector(vector<T> &a, vector<T> &b) {
    vector<T> norm;
    norm.reserve(a.size());

    for(unsigned int i = 0; i < a.size(); i++) {
        norm.push_back(a[i] - b[i]);
    }
    return norm;
}



template <class T>
vector<vector<unsigned  int>>  analisThree(bmHeap<T>  *h,
CRS<T> &mtx) {
```

```
    vector<vector<unsigned int>> answer;
    vector<unsigned int> index_answer;

    unsigned int SizeVec = mtx.getColSize() - 1;

    bmNode<T> *x = nullptr;
    unsigned int i = 0;
    x = h->getHeap();
     std::cout << "SizeVec : " << SizeVec << std::endl;
    while(x && (i < SizeVec - 1)) {
        //PґPsP±P°PIP»PµPSPёPµ
        for(; x ; x = x->sibling) {
            if(x->degree == 0)
                index_answer.push_back(x->key);
        }


     answer.push_back(index_answer);
        for(unsigned int ii = 0; ii < index_answer.size();
ii++)
            h->bmheap_deleteKey(h->bmheapMax(),
index_answer[ii]);
        x = h->getHeap();
        i += index_answer.size();
          index_answer.clear();
    }
    cout << "count parallel size vect" << answer.size() <<
endl;
    return answer;
}

#endif // ALLFUNC_H
```

Листинг В.5 – main.cpp

```
#include <iostream>
#include <ctime>
#include "crs.h"
#include "allfunc.h"
#include "xy.h"


#define nameFile "text"
using namespace std;
int main()
{
    vector<double> answerCRS, answer2CRS;
    vector<double> answerXY;
    XY<double> myXy = read_xy<double>(nameFile);
    CRS<double> myCrs = converting_crs<double>(myXy);

    testRight(myCrs);
    //testLeft(myCrs);
    return 0;
}
```