МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)**

**Институт информационных технологий, математики и механики**

Кафедра математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная Математика и Информатика»   
Профиль подготовки: «Вычислительные методы   
и суперкомпьютерные технологии»

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА**

**Тема:**

Решение разреженных треугольных систем алгебраических   
линейных уравнений с плотной правой частью

Выполнил:

Заведующий кафедрой: студент группы 381903-3М,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Романов А. А.

Проверил:

к.т.н, доцент кафедры   
МОиСТ ИИТММ,

Мееров И. Б.

Нижний Новгород  
2021

Оглавление

[Введение 3](#_Toc73708098)

[1. Обзор существующих методов решения треугольных систем 5](#_Toc73708099)

[2. Постановка задачи и цели работы 7](#_Toc73708100)

[3. Алгоритмы решения СЛАУ с треугольными матрицами 8](#_Toc73708101)

[3.1 Базовый алгоритм 8](#_Toc73708105)

[3.2 Супернодальный алгоритм 9](#_Toc73708106)

[3.3 Параллельный алгоритм с барьерной синхронизацией 12](#_Toc73708107)

[3.4 Алгоритм без синхронизаций с параллелизмом по варпам 15](#_Toc73708108)

[3.5 Алгоритм без синхронизаций с параллелизмом по потокам 18](#_Toc73708109)

[4. Программная реализация 20](#_Toc73708110)

[4.1 Структура программного комплекса 20](#_Toc73708112)

[4.2 Используемые структуры данных 21](#_Toc73708113)

[4.3 Реализация базового алгоритма 21](#_Toc73708114)

[4.5 Базовый супернодальный алгоритм 21](#_Toc73708115)

[4.6 Cупернодальный алгоритм с использованием функций BLAS 22](#_Toc73708116)

[4.7 Параллельный алгоритм с барьерной синхронизацией 23](#_Toc73708117)

[4.8 Параллельные алгоритмы OpenCL без синхронизаций 23](#_Toc73708118)

[5. Результаты вычислительных экспериментов 25](#_Toc73708119)

[5.1 Тестовая система 25](#_Toc73708120)

[5.2 Методика проведения экспериментов 25](#_Toc73708121)

[5.3 Тестовые задачи 26](#_Toc73708122)

[5.4 Анализ корректности 27](#_Toc73708123)

[5.5 Анализ производительности программных реализаций алгоритмов решения разреженных СЛАУ 28](#_Toc73708124)

[5.5.1 Системы с одной правой частью 28](#_Toc73708125)

[5.5.2 Системы с множественными правыми частями 34](#_Toc73708126)

[Заключение 40](#_Toc73708127)

[Список литературы 41](#_Toc73708128)

[Приложение А. Фрагменты исходного кода программы. 43](#_Toc73708129)

Введение

При решении многих прикладных задач требуется решать большие разреженные системы линейных алгебраических уравнений (СЛАУ). При решении дифференциальных уравнений в частных производных, в машинном обучении и в других областях матрица системы часто такова, что почти все ее элементы равны нулю [1]. Методы решения подобных систем подробно описаны в учебниках и научных публикациях. Так, итерационные методы стороят последовательность приближений к решению исходной системы и применяются для решения огромных систем, но в ряде случаев демонстрируют очень медленную сходимость. Напротив, прямые методы, основанные на разложении (факторизации) матрицы системы в произведение треугольных матриц [2], лишены этого недостатка, однако требуют больших объемов памяти для хранения промежутчных структур данных. В данной работе рассматриваются один из этапов прямого решения СЛАУ – решение треугольных систем. Рассматривается частный случай, когда правая часть системы - плотный вектор, а также более общий случай, когда правая часть системы представлена плотной матрицей. Известно, что в обоих случаях применяются схожие подходы, однако объемы и характер вычислений могут существенно отличаться.

В настоящее время существует немало реализаций прямых решателей для разреженных СЛАУ. Широко распространены и активно применяются MKL PARDISO, MUMPS, SuperLU, CHOLMOD и другие высокопроизводительные программные пакеты, которые могут решать системы огромной размерности и запускаться на суперкомпьютерах различной архитектуры. Некоторые пакеты работают только на традиционных центральных процессорах (CPU), другие могут быть запущены на графических процессорах (GPU). При всем многообразии имеющегося программного обеспечения, вопрос о том, какие алгоритмы лучше применять в тех или иных случаях, как их реализовывать и распараллеливать, как добиться наилучшей производительности в рамках отдельных этапов прямого решения, продолжает активно изучаться исследователями по всему миру. Публикации и доклады по этой тематике (подробнее в обзоре литературы далее) подтверждают этот факт.

Отметим ещё несколько особенностей практических задач, в которых возникает необходимость решать разреженные системы линейных уравнений. Часто размеры систем достаточно велики, и возникает вопрос выбора наиболее подходящей структуры хранения данных, эффективной реализации решателя систем уравнений, а также выбора платформы для вычислений. Сейчас широкое распространение получили многоядерные и многопроцессорные вычислительные платформы, что позволяет шире использовать параллельные и распределённые алгоритмы. В данной работе представлен обзор существующих методов решения разреженных треугольных систем, рассмотрены и реализованы некоторые подходы к их решению, проведены вычислительные эксперименты, подтверждена корректность разработанных алгоритмов и сделаны выводы по результатам работы.

Автор благодарит С. А. Лебедева, В. Д. Волокитина и Д. Р. Ахмеджанова за полезные обсуждения и внимание к работе.

1. Обзор существующих методов решения треугольных систем

Прямые методы решения разреженных СЛАУ основаны на разложении материцы системы в производение двух матриц достаточно простой структуры, позволяющей далее применить достаточно простой алгоритм для решения двух СЛАУ и построить решение исходной задачи. На практике в зависимости от свойств матрицы применяются разложение Холецкого, LU-разложение, QR-разложение. В работе рассматриваются системы с разреженной треугольной матрицей и подходы к их решению. Отметим, что базовым методом решения таких систем является широко распространенный метод Гаусса, который в случае плотных систем может быть легко реализован в виде весьма ограниченного по объему несложного кода. Однако в случае разреженных систем реализация «по определению» приводит к крайне неэффекивным с вычислительной точки зрения процедурам, никак не использующим факт расположение ненулевых элементов в матрице для организации параллелизма. Такой «наивный» подход оказывается неприменимым при решении задач большой размерности, которые регулярно возникают из практических приложений. Выясняется, что для повышения эффективности вычислений необходимо анализировать структуру разреженной матрицы, использовать соотвествующие графовые модели для выделения независимых порций вычислений и применять другие нетривиальные техники. Далее рассмотрены примеры работ, в которых описаны исследования, выполненные в данной области.

Базовым подходом к решению треугольных систем является метод исключения Гаусса. В связи с возможностью организовать параллельные вычисления только внутри итераций метода, значительные накладные расходы делают этот подход малоприменимым в реальных задачах большой размерности.

В 1982 году Роберт Шрайбер (Robert Schreiber) предложил новый метод реализации метода исключения Гаусса для разреженных матриц[3]. С помощью усовершенствованного варианта хранения матриц ему удалось увеличить эффективность исполнения алгоритма на многопроцессорных системах.

Эдвард Андерсон и Юсеф Саад (Edward Anderson, Youcef Saad) в 1989 году предложили различные блочные алгоритмы для решения разреженных систем[4], с помощью которых можно эффективно решать системы с множеством правых частей. Использование алгоритмов предполагало дополнительные расходы на предварительную подготовку данных, которые компенсировались ускорением от параллелизма по уровням графа зависимостей или блокам матрицы в зависимости от подхода.

В статье 1997 года Деммель и Гильберт (James Demmel, John Gilbert) опубликовали эффективный алгоритм[5], позволяющий преодолеть трудности с организацией параллелизма в методе исключения Гаусса. Высокая производительность алгоритма достигалась за счет использования метода редукции графа зависимостей, выделения супернодов на одном узле вычислительной системы для высокой его загрузки, а также планирования двух типов параллельных задач для достижения параллелизма более высокого уровня.

В 2011 году Максим Наумов предложил новый параллельный алгоритм решения разреженных систем на графических процессорах[6]. Решение производится в два этапа: на этапе анализа строится граф зависимостей на основе разреженной матрицы, независимые строки группируются в уровни параллелизма. На второй фазе вычисляется решение путем последовательного перебора построенных уровней.

В 2016 году Вейфен Лю, Анг Ли и Джонатан Хогг (Weifeng Liu, Ang Li, Jonathan Hogg) предложили асинхронный параллельный алгоритм[7], решающий проблему значительных затрат времени на предварительную подготовку данных. Авторы представили новый подход для решения разреженных СЛАУ, в котором переупорядочивание компонент матрицы естественным образом осуществляется на стадии решения. Таким образом, стоимость предварительной обработки удалось значительно снизить, а синхронизация между уровнями графа зависимостей была полностью исключена.

В 2020 году Джия Су, Фенг Жан и Вейфен Лю (Jiya Su, Feng Zhang, Weifeng Liu) усовершенствовали[8] предложенный ранее алгоритм[7], перейдя от параллелизма на уровне варпа (группы потоков Nvidia GPU) к параллелизму на уровне потоков внутри варпа и значительно ускорив решение системы. Кроме того, авторам удалось полностью исключить стадию препроцессинга.

Далее в работе приводится сравнение различных методов решения треугольных систем.

1. Постановка задачи и цели работы

Пусть дана система линейных уравнений где матрица является разреженной верхнетреугольной, а матрица правой части – плотная, .

Целью данной работы является сравнительный анализ различных алгоритмов, решающих данную систему. Для достижения этой цели необходимо решить следующие задачи:

1. Разработать базовый алгоритм решения системы .
2. Рассмотреть супернодальную версию алгоритма.
3. Оптимизировать супернодальный вариант с использованием функций BLAS библиотеки MKL.
4. Рассмотреть алгоритм с барьерной синхронизацией и современные подходы к решению системы, ориентированные на использование графического процессора.
5. Перейти к решению системы с несколькими правыми частями.
6. Подтвердить корректность получаемых результатов.
7. Провести вычислительные эксперименты и сделать сравнительный анализ по результатам работы.
8. Алгоритмы решения СЛАУ с треугольными матрицами

Так как алгоритмы решения систем и во многом схожи, не сокращая общности будем рассматривать только алгоритмы для верхнетреугольных матриц.



3.1 Базовый алгоритм

Пусть известна верхнетреугольная матрица *U*, и требуется найти решение системы . Предположим также, что система совместна. Тогда в простейшем случае решение *y* может быть получено с помощью применения обратного хода Гаусса:

Системаимеет вид

|  |  |
| --- | --- |
|  | () |

где – верхнетреугольная матрица размером , , и – векторы длины . На первой итерации алгоритма вычисляется значение . Уменьшенная система записывается аналогично системе (1) и решается рекурсивно. После завершения работы алгоритма вектор *x* будет содержать итоговое решение системы.

Алгоритм 1. Базовый алгоритм решения треугольной СЛАУ. Первый вариант.

*Вход:* Верхнетреугольная матрица U, матрица правой части Y.

*Выход:* решение системы Х.

1. В цикле по правым частям :
   1. Вычислить ;
   2. В цикле по компонентам вектора:

sum = 0.;

i.В цикле по ненулевым элементам строки

sum += ;

ii. ;

В случае системы с несколькими правыми частями существует два варианта организации вычислений:

1. Добавить внешний цикл по правым частям перед началом алгоритма (рассмотрен выше).

2. В силу однородности преобразований правой части системы, добавить циклы только в те части алгоритмов, где происходят вычисления с непосредственным участием матрицы (приведён в алгоритме 1.1).

Алгоритм 1.1. Базовый алгоритм решения треугольной СЛАУ.Второй вариант.

*Вход:* Верхнетреугольная матрица U, матрица правой части Y.

*Выход:* решение системы Х.

1. В цикле по правым частям :
   1. Вычислить ;
2. В цикле по компонентам вектора:
   1. В цикле по правым частям :

sum = 0.;

i.В цикле по ненулевым элементам строки

sum += ;

ii. ;

Оба варианта реализации рассматриваются для базового и супернодального подходов, а также для параллельного алгоритма с барьерной синхронизацией.

3.2 Супернодальный алгоритм

При выполнении разложения Холецкого получаемая матрица *U* имеет специфическую структуру: матрицу можно представить в виде блоков, каждый из которых состоит из двух характерных составляющих: плотная треугольная матрица,находящаяся в левой части блока, и оставшаяся часть блока, строки которой имеют идентичный портрет. Такой блок называют супернодом.

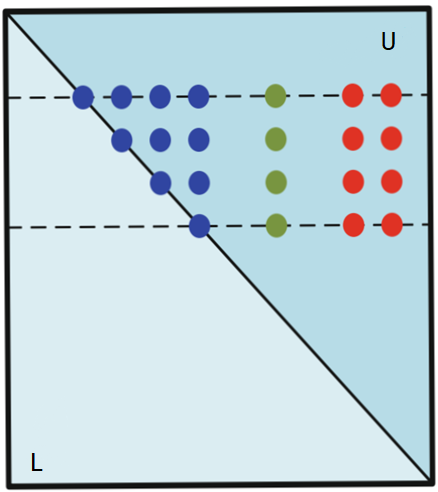


Рисунок 1. Портрет супернода верхнетреугольной матрицы. Синими точками обозначена плотная треугольная матрица, зелёными и красными – ненулевые элементы прямоугольной части, каждая строка которой имеет идентичный другим портрет. Адаптирован при помощи [9].

По сравнению с базовым алгоритмом, супернодальный вариант решения [9] является более оптимальным: необходимые операции над подматрицами супернода более эффективны из-за компактного расположения читаемых из кеша данных.

Для реализации алгоритма требуется знать расположение супернодов матрицы. В данной работе использованы матрицы и суперноды к ним, полученные с помощью решателя для алгебраических систем разреженных линейных уравнений с симметричной положительно определённый матрицей[10]. Массив супернодов содержит возрастающую последовательность индексов, причём значение является индексом строки, с которой начинается очередной супернод, а разность будет его размером. Были разработаны две версии супернодального алгоритма: в первом варианте вычисления, относящиеся к пунктам 1.a.iii. и 1.a.v., были реализованы без использования стороннего программного обеспечения, во втором варианте были использованы реализации операций BLAS второго и третьего уровня из библиотеки MKL.

Алгоритм 2. Супернодальный алгоритм решения треугольной СЛАУ.

*Вход:* Верхнетреугольная матрица U, вектор правой части Y, массив супернодов.

*Выход:* решение системы X.

1. В цикле по правым частям :
   1. В обратном цикле по индексам массива nodes:
      1. Приравнять
      2. Выделить треугольную и прямоугольную части супернода
      3. Решить плотную прямоугольную систему ,

где – элементы правой части, расположенные напротив прямоугольной подматрицы.

* + 1. Обновить вектор
    2. Решить плотную треугольную систему   
       где – элементы правой части, расположенные напротив треугольной подматрицы.

Супернодальный алгоритм, использующий функции BLAS библиотеки MKL, требует хранения матрицы в «ступенчатом» треугольном формате: в структуре CRS дополнительно хранятся нулевые элементы, находящиеся в верхнем треугольнике плотных частей супернодов. Для формирования изменённой структуры матрицы был разработан соответствующий алгоритм.

Алгоритм 3. Алгоритм формирования изменённой структуры CRS.

*Вход:* Структура матрицы val, col\_index, row\_ptr, массив супернодов snodes.

*Выход:* Структура ступенчатой верхнетреугольной матрицы val\_pad, col\_index\_pad, row\_ptr\_pad.

*Инициализация:*

Индекс столбца в суперноде col\_ind\_cnt = 0;

Отступ от нулевого столбца global\_pad = 0;

1. Вычислить объем дополнительной памяти для хранения матрицы в новом формате:  
   В цикле по индексам s\_ind массива snodes:
   1. node\_size = snodes[s\_ind + 1] – snodes[s\_ind];
   2. extra\_mem += (node\_size \* (node\_size - 1)) / 2;

memalloc();

1. В цикле по индексам s\_ind массива snodes:
   1. node\_size = snodes[s\_ind + 1] – snodes[s\_ind];
   2. В цикле по i, i = 0, ..., node\_size:
      1. В цикле по j, j < i; j++, global\_cnt++:
         1. val\_pad[global\_cnt] = 0.0;
         2. row\_ptr\_pad[global\_cnt] = col\_index[row\_ptr[snodes[s\_ind]]] + j;
         3. col\_ind\_cnt += 1;
      2. В цикле по j = row\_ptr[snodes[s\_ind] + i], j < row\_ptr[snodes[s\_ind] + i + 1]; ++j:
         1. val\_pad[global\_cnt] = val[j];
         2. row\_ptr\_pad[global\_pad] = col\_index[j];
         3. col\_ind\_cnt += 1;
      3. col\_index\_pad[snodes[j] + i + 1] = col\_ind\_cnt;
2. nz\_pad = col\_ind\_cnt;
   1. Параллельный алгоритм с барьерной синхронизацией

Данный раздел написан с использованием материалов из [11].

Возможность эффективно распараллелить наивный алгоритм умножения разреженных матриц отсутствует в связи с мелкой гранулярностью задачи. Общепринятым подходом, позволяющим решить эту проблему, является построение графа уровней зависимостей задач.

Для наглядности рассмотрим матрицу, изображённую на рисунке 2. Вычисление элемента искомого вектора зависит от того, вычислен ли элемент . В общем случае, если в матрице существует ненулевой элемент , то вычисление переменной зависит от переменной . Однако значения некоторых переменных могут быть найдены независимо, как, например, значения и , и .

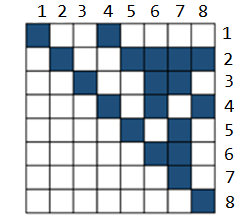


Рисунок 2. Портрет верхнетреугольной матрицы. Адаптирован при помощи [11].

Граф зависимости задач состоит из уровней, каждый из которых содержит задачи, которые можно решать независимо. В этой связи появляется возможность распараллелить алгоритм по уровням полученного графа. На рисунке 3 (a) представлено разбиение задач по уровням.

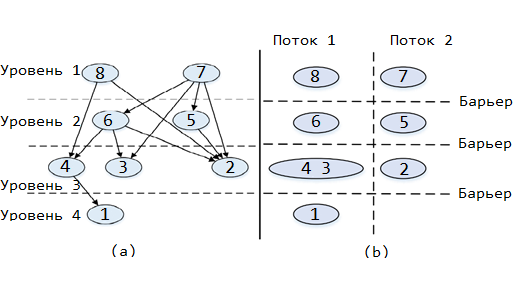


Рисунок 3. (а) Граф зависимости задач с разбиением по уровням, соответствующий матрице  
на рисунке 2. (б) Иллюстрация распределения нагрузки по потокам и барьерной синхронизации.

Адаптировано при помощи [11].

Для более эффективной масштабируемости алгоритма стоит учесть возможность ребалансировки нагрузки на потоки. Рассмотрим третий уровень представленного выше графа. Если для решения каждой из задач выделить по одному потоку, то после завершения вычислений первые два будут простаивать, ожидая завершения работы в третьем потоке. Простоя можно избежать, объединив пятую и шестую задачи и выделив для их решения один поток. В общем случае стоит распределить задачи по потокам так, чтобы количество вычислений и доступов к памяти для каждого из них было примерно одинаковым.

Так как исходный граф зависимостей имеет мелкую гранулярность (каждая задача соответствует отдельной строке матрицы), его параллельное исполнение сопряжено со значительными накладными расходами на синхронизации. Число синхронизаций пропорционально числу рёбер в графе или, что эквивалентно, числу ненулевых элементов в исходной матрице. При использовании барьеров на каждом уровне графа зависимостей, количество синхронизаций уменьшается до числа уровней графа, что положительно отражается на производительности алгоритма.

Иллюстрация распределения нагрузки по потокам и барьерной синхронизации представлена на рисунке 3 (б).

Весь алгоритм можно разделить на три части:

* Вычисление уровней графа зависимостей.
* Укрупнение задач и балансировка нагрузки на потоки.
* Нахождение части искомого вектора внутри потока.

Реализация каждой из вышеперечисленных частей представлена в алгоритмах 4-6.

Алгоритм 4. Вычисление уровней графа зависимостей.

*Вход:* Структура матрицы U,порядок системы n.

*Выход:* массив уровней графа levels, максимальный уровень max\_level.

*Инициализация:* levels[i] = 1, i = 0,...,n-1;

1. В цикле по i, i = (n – 1), ..., 0:
   1. Если в строке матрицы больше одного ненулевого элемента:
      1. В цикле по элементам строки матрицы:
         1. Если levels[col\_index[j]] > max\_level\_in\_line:
            1. max\_level\_in\_line = levels[col\_index[j]];
      2. levels[i] += max\_level\_in\_line;
   2. Если max\_level\_in\_line > max\_level:
      1. max\_level = levels[i];

Алгоритм 5. Балансировка нагрузки на потоки.

*Вход:* Структура матрицы U, порядок системы n, массив уровней графа levels, число потоков num\_threads, максимальный уровень графа max\_level.

*Выход:* разбиение задач tasks\_by\_level и строк исходной матрицы rows\_by\_level по уровням.

*Инициализация:*

std::vector<int>\* rows\_by\_level = new std::vector<int>[max\_level+1];

std::vector<int>\* tasks\_by\_level = new std::vector<int>[max\_level+1];

rows\_by\_level[levels[i]] = i, i = 0,...,n-1;

1. В цикле по i, i = 0, ..., max\_level:
   1. Найти число ненулевых элементов на уровне level\_nz;
   2. Вычислить limit = level\_nz / num\_threads;
   3. Определить current\_task = 1, current\_nz = 0;
   4. В цикле по строкам i-го уровня:
      1. Если current\_task == num\_threads:
         1. task\_by\_level[i][current\_task] += 1;
         2. continue;
      2. task\_by\_level[i][current\_task] += 1;
      3. Если current\_nz + row\_nz > limit:
         1. current\_task += 1;
      4. Иначе:
         1. current\_nz += row\_nz;

Алгоритм 6. Вычисление элементов вектора *x*.

*Вход:* Структура матрицы U, вектор правой части y, порядок системы n, число потоков num\_threads, максимальный уровень графа max\_level, разбиение задач tasks\_by\_level и строк исходной матрицы rows\_by\_level по уровням.

*Выход:* вектор *x*, содержащий решение системы.

1. Инициализировать приватную секцию

#pragma omp parallel private(vertex, sum)

1. В цикле по i, i = 0, ..., max\_level:
   1. Инициализировать блочно-циклическое распределение итераций цикла

#pragma omp for schedule(static, 1)

* 1. В цикле по задачам уровня i:
     1. В цикле по строкам матрицы, принадлежащим задаче:
        1. vertex = rows\_by\_level[i][j];
        2. sum = 0;
        3. В цикле по ненулевым элементам строки:
           1. sum += val[k] \* x[col\_index[k]];
        4. Вычислить значение x[vertex] =   
           (b[vertex] - sum) / val[row\_ptr[vertex]];
  2. Алгоритм без синхронизаций с параллелизмом по варпам

Данный раздел написан с использованием материалов из [7].

Затраты времени на предварительную подготовку данных и синхронизации внутри параллельных секций составляют внушительную долю от общего времени работы алгоритма, рассмотренного в пункте 3.3. Согласно результатам, полученным Liu at al. в [7], стадия препроцессинга занимает в 4.39–12.65 раз больше времени, чем стадия непосредственного решения системы. Кроме этого, при значительном количестве уровней в графе зависимостей затраты на синхронизацию могут занимать 85% от общего времени исполнения алгоритма. Для повышения производительности параллельного алгоритма Liu at al. был предложен алгоритм, сокращающий время предобработки и исключающий синхронизации.

Предполагается, что алгоритм будет исполняться на графическом процессоре с помощью технологии OpenCL. Каждая компонента вектора вычисляется с использованием одного варпа(объединённой группы из 32 SIMT-потоков).

В связи с однонаправленностью графа, при параллельном вычислении компонент вектора необходимо обеспечить вычисление очередной компоненты только после вычисления всех предыдущих. Прежде чем вычислить текущее значение , необходимо найти количество компонент , от которых зависит. Другими словами, – это число варпов, которые должны закончить свою работу перед вычислением очередной компоненты. Их количество совпадает с числом ненулевых элементов в строке, исключая диагональный. Также оно совпадает с числом рёбер, входящих в вершину графа зависимостей. Вся процедура предобработки сводится к формированию массива ненулевых элементов. Алгоритм 7 содержит псевдокод процедуры предобработки.

Алгоритм 7. Препроцессинг метода решения СЛАУ без синхронизаций.

*Вход:* Структура матрицы U.

*Выход:* вектор in\_degree, содержащий количество ненулевых элементов для каждой строки матрицы.

*Инициализация:* memset(in\_degree, 0, n \* sizeof(int));

1. Инициализировать параллельную секцию
2. В цикле по числу ненулевых элементов:
   1. in\_degree[row\_index[i]] += 1; // atomic add

Вычислив массив , можно инициализировать необходимое количество варпов для параллельного решения системы. После того, как варп завершает вычисление компоненты , он уведомляет об этом все остальные варпы, зависящие от этой компоненты, с помощью атомарного обновления массива . Операция обновления должна быть атомарной потому, что различные варпы могут обновлять массив одновременно. Псевдокод метода приведён в алгоритме 8.

Алгоритм 8. Параллельный алгоритм без синхронизаций.

*Вход:* Структура матрицы U, размерность системы n, вектор правой части y.

*Выход:* вектор *x*, содержащий решение системы.

*Инициализация:* Выделение памяти для вспомогательных структур

malloc(\*d\_left\_sum,\*s\_left\_sum,\*d\_in\_degree,\*s\_in\_degree,n);

memset(\*d\_left\_sum,\*s\_left\_sum,\*d\_in\_degree,\*s\_in\_degree,0);

Вызов алгоритма 7.

1. Установить размер диагонального блока diag\_dim;
2. Создать варп для каждой компоненты вектора x.

В цикле по i, i = (n – 1), ..., 0:

1. Пока s\_in\_degree[i]+1 != d\_in\_degree[i]:
   * 1. Ожидание
2. x[i] = (b[i] – d\_left\_sum[i]-s\_left\_sum[i])/val[col\_ptr[i]];
3. Создать потоки по числу ненулевых элементов столбца

В параллельном цикле по числу ненулевых элементов столбца:

1. rid = row[j]
2. Если rid < i + diag\_dim – i % diag\_dim:

Использовать локальную память GPU

* + - * 1. atomic\_add(&s\_left\_sum[rid], val[j] \* x[i]);
        2. atomic\_add(&s\_in\_degree[rid], 1);

1. Иначе:

Использовать глобальную память

* + - * 1. atomic\_add(&d\_left\_sum[rid], val[j] \* x[i]);
        2. atomic\_sub(&d\_in\_degree[rid], 1);

1. Освободить вспомогательные массивы.

Алгоритм 8 стоит проиллюстрировать примером (см. рисунок 4). Диаграмму можно разделить на три региона: регион ожидания (верхняя часть диаграммы), критическая секция, содержащая массив left\_sum\_array (красный прямоугольник) и секция блокировки (зелёный прямоугольник). В секции ожидания варп постоянно проверяет, доступна ли критическая секция. Если нет, он ожидает её разблокировки, работая в т. н. spinning-цикле. В критической секции варп обновляет компоненты массива left\_sum, зависящие от переменной . После этого варп попадает в последнюю секцию, где обновляет массив in\_degree в том же порядке, в котором обновлял компоненты left\_sum. После того, как компоненты массива in\_degree обновлены до целевых значений (когда все зависимости между компонентами вектора разрешены), снимается блокировка с соответствующих варпов.

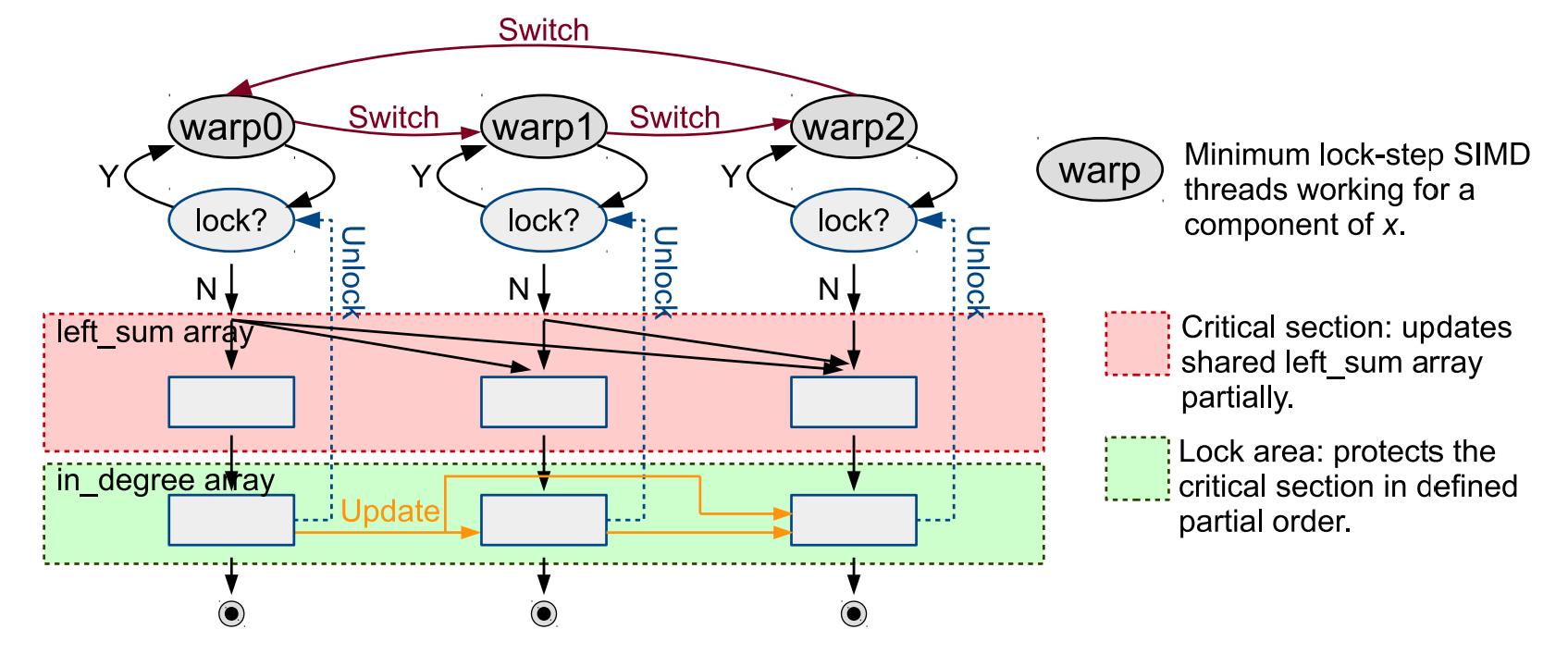


Рисунок 4. Иллюстрация работы алгоритма 8. Заимствовано из [7].

* 1. Алгоритм без синхронизаций с параллелизмом по потокам

Данный раздел написан с использованием материалов из [8].

Параллельный алгоритм без синхронизаций на уровне варпов демонстрирует существенное ухудшение производительности в том случае, если:

1. Среднее число компонент в уровне велико.
2. Число ненулевых элементов в каждой строке мало.

Для решения этих проблем необходимо:

1. Избежать простоя вычислений внутри варпа.
2. Избежать взаимных блокировок между варпами.

Кроме этого, нужно избежать затрат времени на предварительную обработку данных перед непосредственным решением системы.

Идея данного алгоритма заключается в использовании одного потока графического процессора для вычисления одной компоненты вектора . Данных подход позволяет решить обе обозначенные выше проблемы.

Особенностью этого подхода является необходимость хранения дополнительного массива индикаторов из элементов. Для соответствующей компоненты индикатор становится равен единице, если компонента вычислена.

Псевдокод данного метода представлен в алгоритме 9.

Алгоритм 9. Параллельный алгоритм без синхронизаций по уровням потоков.

*Вход:* Структура матрицы U, размерность системы n, вектор правой части y, вектор индикаторов get\_value.

*Выход:* вектор *x*, содержащий решение системы.

*Инициализация:* Выделение памяти для вспомогательных структур

malloc(\*get\_value, n);

memset(\*get\_value, 0);

global\_id = n – 1 - get\_global\_id(0);

left\_sum = 0;

1. В цикле по i, i = (n – 1), ..., 0: // свой поток для каждого i
   1. left\_sum = 0.;
   2. j = row\_ptr[i + 1] – 1;
   3. Пока j >= row\_ptr[i]:
      1. col = col\_index[j];
      2. Пока get\_value[col] == True:
         1. left\_sum += val[j] \* x[col];
         2. j -= 1;
         3. col = col\_index[j];
      3. Если i == col:
         1. x[i] = (y[i] – left\_sum) / val[row\_ptr[i]];
         2. get\_value[i] = 1;
         3. j -= 1;
2. free(\*get\_value);

В алгоритме проверяется два условия:

* + - 1. Вычислен ли нужный элемент вектора . Если да, то происходит накопление частичной суммы , после чего происходит переход к следующему значению в строке.
      2. Является ли текущий элемент последним. Если да, происходит вычисление соответствующего элемента вектора , и в массиве для него устанавливается индикатор, равный единице.

1. Программная реализация

Для решения разреженных треугольных систем был рассмотрен ряд алгоритмов:

Основные алгоритмы:

1. Базовый алгоритм решения треугольной СЛАУ.
2. Алгоритм выделения супернодов.
3. Базовый супернодальный алгоритм.
4. Супернодальный алгоритм с использованием функций BLAS.
5. Параллельный алгоритм с барьерной синхронизацией.
6. Параллельный алгоритм без синхронизаций на уровне варпов с использованием технологии OpenCL.
7. Параллельный алгоритм без синхронизаций на уровне потоков с использованием технологии OpenCL.

Вспомогательные алгоритмы:

1. Алгоритм вычисления уровней графа зависимостей.
2. Алгоритм балансировки нагрузки на потоки.
3. Функции проверки полученных результатов.
4. Функции чтения/записи, транспонирования матриц и проверки результата.

Программная реализация была выполнена на языке С++ с использованием компилятора из пакета Intel® OneAPI 2021.2. Приложение А содержит фрагменты исходного кода программы.

Для алгоритмов, перечисленных под пунктами 1-6 списка выше, реализована возможность решать систему вида .



4.1 Структура программного комплекса

Ниже приведён файловый состав программных модулей:

Основной вычислительный модуль:

**core.h, core.cpp** – реализация алгоритмов 1-7 из списка выше.

Вспомогательный модуль:

**util.h, util.cpp** – реализация алгоритмов 8-11 из списка выше, а также реализация некоторых дополнительных методов.

Интерфейс командной строки:

**main.cpp** – обеспечение взаимосвязи между пользователем и элементами программного комплекса.

4.2 Используемые структуры данных

Для хранения разреженных матриц используется широко распространённый формат CRS – compressed row storage[12], который использует три массива. Первый массив val хранит ненулевые значений элементов, второй – col\_index – номера столбцов для каждого элемента, а третий – row\_ptr – индексы первых элементов каждой строки. В последнем элементе row\_ptr[N] хранится значение , где – количество ненулевых элементов, – размерность матрицы.

Для хранения разреженной матрицы в формате CRS требуется   
 байт.

Наряду с форматом CRS также используется похожий формат CCS – compressed column storage. Его структура аналогична рассмотренному ранее формату: хранится массив ненулевых значений, массив соответствующих номеров строк и массив индексов, с которых начинается очередная столбец. Объём требуемой памяти одинаков для обоих форматов.

Для хранения разреженной матрицы в ступенчатом треугольном виде необходимо потратить дополнительную память на дозаполнение плотных треугольных частей до квадратов. Объём памяти зависит от числа супернодов и их размера.

Суперноды хранятся в дополнительном массиве, занимающие байт, где – количество супернодов.

Для хранения разбиения задач и строк матрицы по уровням графа зависимостей используются два вектора объёмом байт, где – номер последнего уровня в графе зависимостей.

В алгоритме без синхронизаций с параллелизмом по потокам также используется массив индикаторов, занимающий байт.

4.3 Реализация базового алгоритма

double base\_gauss\_upper(int n, double\* val, int\* col\_index, int\* row\_ptr, double\* y, double\* x, int rhs)

Функция реализована согласно алгоритму 1 пункта 3.1. На вход поступает размерность системы, структура CRS хранения разреженной матрицы, указатель на вектор правой части y и число правых частей системы. Результат записывается в вектор х. Функция возвращает время работы алгоритма. Здесь и далее рассмотрены алгоритмы, работающие с верхнетреугольными системами.

4.5 Базовый супернодальный алгоритм

double supernodal\_upper(int\* supernodes, size\_t sn, double\* val, int\* col\_index, int\* row\_ptr, double\* x, int rhs);

Разработанный алгоритм принимает на вход следующие параметры:

указатель на массив супернодов, их количество, а также структуру CRS и и число правых частей системы. Результат записывается в вектор х. Функция возвращает время работы алгоритма. В теле представленного алгоритма используются две вспомогательные функции, предназначенные для вычисления решения систем и   
 согласно алгоритму 2 пункта 3.2.

void get\_rect\_part\_upper(int isn, int dim, double\* x, double\* val, int\* col\_index, int\* row\_ptr);

void get\_tr\_part\_upper(int isn, int dim, double\* x, double\* val, int\* col\_index);

Функция get\_rect\_part\_upper принимает на вход индекс строки, с которой начинается супернод, его размер, вектор текущих значений и структуру матрицы. После решения системы обновляются компоненты вектора х:

Функция get\_tr\_part\_upper принимает на вход индекс строки, с которого начинается супернод, его размер, вектор текущих значений, массив ненулевых значений матрицы и массив номеров столбцов. После решения плотной треугольной системы обновлённые компоненты вектора х перезаписываются в массив.

## 4.6 Cупернодальный алгоритм с использованием функций BLAS

В работе также предложена реализация супернодального алгоритма с использованием функций BLAS.

double supernodal\_blas\_upper(int n, size\_t sn, int\* supernodes, double\* x, double\* val, int\* col\_index, int\* row\_ptr, int rhs);

Ранее рассмотренные функции get\_tr\_part\_upper и get\_rect\_part\_upper заменяются соответственно на вызовы cblas\_dtrsm(CblasColMajor, CblasLeft, CblasUpper, CblasNoTrans, CblasNonUnit, tr\_dim, 1, 1., val + shift, tr\_dim,

x + supernodes[i], tr\_dim);

и

cblas\_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, rec\_dim, 1, tr\_dim, 1., val + shift + tr\_dim, lda,

x + supernodes[i], tr\_dim, 1., c + supernodes[i], lda).

Функция cblas\_dtrsm позволяет решать треугольные системы вида  
, где – скаляр, *Х* и *B* – матрицы размерности , *A* – верхнетреугольная матрица размерности .

Функция cblas\_dgemm позволяет выполнять операции вида

,

где и – скаляры, матрица *А* имеет размер , матрица *B* имеет размер , матрица *C* имеет размер .

Рассмотренным функциям на вход подаются метапараметры CblasColMajor, CblasLeft, CblasUpper, CblasNoTrans, CblasNonUnit, определяющие выполнение вычислений, а также указатели на необходимые элементы подматриц, получаемые с помощью смещений. Подробную документацию по этим функциям можно изучить в [13].

4.7 Параллельный алгоритм с барьерной синхронизацией

double barrier\_sync(int n, double\* val, int\* col\_index, int\* row\_ptr, int nthreads, double\* y, double\* x, int rhs);

Функция реализована согласно алгоритму 6 пункта 3.3. На вход поступает размерность системы, структура CRS хранения разреженной матрицы, указатель на вектор правой части y, число правых частей системы rhs и число OpenMP-потоков nthreads. Результат записывается в вектор х. Функция возвращает время работы алгоритма.

В теле представленного алгоритма вызываются две вспомогательные функции, выполняющие нахождение уровней в графе зависимостей и балансировку нагрузки на вычислительные потоки (см. алгоритмы 4 и 5):

int get\_levels(int n, int\* col\_index, int\* row\_ptr, int\* levels);

double rebalance\_tasks(std::vector<int> rows\_by\_level, std::vector<int> tasks\_by\_level, int\* col\_index, int\* row\_ptr);

Первая функция принимает на вход портрет матрицы и размерность системы. Производится разбиение графа на уровни, результат записывается в массив levels. Функция возвращает номер последнего уровня.

Вторая вспомогательная функция производит укрупнение задач на каждом уровне. На вход поступают портрет матрицы, массив с уровнями levels. Результат работы записывается в массивы tasks\_by\_level и rows\_by\_level. Функция возвращает время работы метода.

## 4.8 Параллельные алгоритмы OpenCL без синхронизаций

Рассмотрим реализацию алгоритма без синхронизаций с параллелизмом на уровне варпов.

double syncfree\_warplevel(int n, int\* col\_index, int\* row\_ptr, double\* y, double\* x, int rhs);

Функция принимает на вход структуру треугольной матрицы, вектор правой части y и количество правых частей rhs. Результат записывается в вектор x. Функция возвращает время работы алгоритма.

В теле функции с помощью вызовов clGetPlatformIDs и clGetDeviceIDs определяется количество платформ и устройств, доступных для запуска OpenCL-ядер. Определяется платформа и девайс, на которых будет проходить исполнение программы. Далее при помощи функций clCreateContext и clCreateCommandQueue определяется контекст исполнения ядер и создаётся очередь команд для осуществления связи между хостом и девайсом. После этого из исходного кода ядер с помощью функции clCreateProgramWithSource создаётся объект программы OpenCL с последующей компиляцией при помощи clBuildProgram. Алгоритмы 7 и 8, рассмотренные в пункте 3.4, реализованы непосредственно в ядрах. После компиляции с помощью вызова clCreateKernel создаются объекты ядер, происходит заполнение буферов данными и исполнение программы на девайсе. Функция clEnqueueWriteBuffer добавляет объекты ядер в очередь команд. Изучить спецификации перечисленных функций можно в [14].

Реализация алгоритма без синхронизаций с параллелизмом на уровне потоков в целом аналогична реализации рассмотренного выше алгоритма. Недоступна возможность решать систему для нескольких правых частей. Также вместо двух ядер используется только одно, реализованное согласно алгоритму 9 пункта 3.5. С исходным кодом ядра можно ознакомиться в приложении А.

1. Результаты вычислительных экспериментов

## 5.1 Тестовая система

Результаты, описанные в этой главе, были получены на двух вычислительных системах, описание которых содержится в таблице 1.

Таблица 1. Параметры тестовой системы.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Название системы | Процессор | RAM | GPU | ОС | Компилятор |
| СК «Лобачевский» | 2 x 8-core Intel® Xeon E5-2660  (2.2 GHz) | 64 GB | Nvidia Tesla K20X | CentOS 6.4 | Intel® C++ Composer  (в составе Intel® Parallel Studio XE 2017)) |
| Рабочая станция | AMD Ryzen 7 PRO 3700 (3.6 GHz) | 64 GB | Nvidia GeForce RTX 3090 | Ubuntu 20.04 | Intel® C++ Composer (в составе Intel® OneAPI 2021.2) |

## 5.2 Методика проведения экспериментов

Проводятся две группы экспериментов: решаются системы вида для и . Отдельно сравниваются последовательные алгоритмы, параллельный алгоритм с барьерной синхронизацией сравнивается с реализацией из библиотеки MKL, а также проводится сравнение между собой алгоритмов, использующих технологию OpenCL (измеряется только время работы ядра). Для всех алгоритмов подтверждается корректность их реализации. Полный набор рассматриваемых алгоритмов представлен в таблице 2.

Таблица 2. Алгоритмы, представленные в работе.

|  |  |  |
| --- | --- | --- |
| Название алгоритма | Одна правая часть | Несколько правых частей |
| Базовый | ✓ | ✓ |
| Базовый оптимизированный |  | ✓ |
| Супернодальный | ✓ | ✓ |
| Супернодальный оптимизированный |  | ✓ |
| Супернодальный с использованием функций BLAS MKL | ✓ | ✓ |
| С барьерной синхронизацией | ✓ | ✓ |
| С барьерной синхронизацией оптимизированный |  | ✓ |
| С параллелизмом по варпам | ✓ | ✓ |
| С параллелизмом по потокам | ✓ |  |

Для автоматического сбора результатов экспериментов был реализован скрипт на языке python3.6. Для каждого из рассматриваемых алгоритмов проводилось несколько запусков с использованием набора тестовых матриц. Каждый запуск проводился в новом процессе для исключения возможности прогрева кеша. После этого выбиралось наименьшее время работы алгоритма для всех матриц. Результаты сохраняются в таблицу в формате .csv.

Для привязки вычислительных потоков к ядрам CPU во время работы параллельного алгоритма с барьерной синхронизацией использовался следующий ключ:

KMP\_AFFINITY=granularity=fine,compact,1,0.

Привязка потоков подробно описана, например, в [15].

Во время тестирования на СК «Лобачевский» использовались дополнительные переменные окружения, позволяющие аллоцировать вычислительный узел(srun –N 1) и выбрать раздел кластера, на котором будут проводиться эксперименты(-p gpu).

## 5.3 Тестовые задачи

Были проведены две серии экспериментов: решались системы вида   
 при и . В экспериментах использовались разреженные матрицы, взятые из коллекции Suite Sparse[16], и верхнетреугольные матрицы, полученные после разложения Холецкого при помощи [10].

Для решения систем используются тестовые матрицы, представленные в таблице 3.

Заполненность матрицы вычисляется по формуле .

Таблица 3. Характеристики верхнетреугольных тестовых матриц.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Название матрицы | Порядок матрицы | Число ненулевых  элементов исходной матрицы | Число ненулевых  элементов верхнетреугольной матрицы | Заполненность исходной матрицы | Заполненность  верхнетреугольной матрицы |
| audikw\_1 | 943 695 | 39 297 771 | 1 224 758 754 | 8,83E-05 | 2,75E-03 |
| Hook\_1498 | 1 498 023 | 31 207 734 | 1 498 134 654 | 2,78E-05 | 1,34E-03 |
| bone010 | 986 703 | 36 326 514 | 1 057 516 857 | 7,46E-05 | 2,17E-03 |
| Emilia\_923 | 923 136 | 20 964 171 | 1 644 545 253 | 4,92E-05 | 3,86E-03 |
| StocF-1465 | 1 465 137 | 11 235 263 | 1 051 381 168 | 1,05E-05 | 9,80E-04 |
| parabolic\_fem | 525 825 | 2 100 225 | 30 626 512 | 1,52E-05 | 2,22E-04 |
| Flan\_1565 | 1 564 794 | 59 485 419 | 1 466 244 810 | 4,86E-05 | 1,20E-03 |
| boneS10 | 914 898 | 28 191 660 | 277 047 525 | 6,74E-05 | 6,62E-04 |
| bone010\_M | 986 703 | 12 437 739 | 365 517 106 | 2,56E-05 | 7,51E-04 |

## 5.4 Анализ корректности

Для подтверждения корректности решения системы каким-либо из реализованных методов вызываются библиотечные функции MKL mkl\_sparse\_d\_trsv или mkl\_sparse\_d\_trsm (в зависимости от количества правых частей системы). Далее вычисляется относительная погрешность: отношение евклидовой нормы разности двух полученных решений к евклидовой норме вектора *х*, полученного одним из методов библиотеки MKL. Полученные значения погрешностей отображены в таблицах 4 и 5.

Таблица 4. Относительные погрешности решения задачи .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Базовый | Супернодаль-ный | Супернодальный BLAS | Барьерная синхронизация, поток | Параллелизм по варпам, OpenCL | Параллелизм по потокам, OpenCL |
| audikw\_1 | 2.905e-15 | 1.865e-15 | 1.236e-15 | 2.905e-15 | 1.865e-15 | 3.981e-05 |
| Hook\_1498 | 1.262e-15 | 8.553e-16 | 4.011e-16 | 1.262e-15 | 5.636e-16 | 8.598e-05 |
| bone010 | 2.456e-15 | 1.277e-15 | 6.811e-16 | 2.456e-15 | 7.650e-16 | 9.775e-05 |
| Emilia\_923 | 2.491e-18 | 1.690e-18 | 1.153e-18 | 2.491e-18 | 1.476e-18 | 2.720e-08 |
| StocF-1465 | 2.190e-15 | 8.223e-16 | 6.591e-16 | 2.190e-15 | 9.324e-16 | 1.706e-04 |
| parabolic\_fem | 5.203e-16 | 3.198e-16 | 2.778e-16 | 5.203e-16 | 2.969e-16 | 3.029e-16 |
| Flan\_1565 | 1.972e-15 | 1.036e-15 | 6.675e-16 | 1.972e-15 | 8.365e-16 | 3.872e-04 |
| boneS10 | 2.228e-15 | 1.895e-15 | 1.413e-15 | 2.228e-15 | 1.643e-15 | 1.713e-15 |
| bone010\_M | 6.239e-11 | 5.618e-10 | 2.047e-10 | 6.239e-11 | 7.837e-12 | 1.451e-10 |

Таблица 5. Относительные погрешности решения задачи .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Базовый | Супернодальный | Супернодальный BLAS | Барьерная синхронизация,  1 поток | Параллелизм по варпам, OpenCL |
| audikw\_1 | 6.284e-15 | 4.035e-15 | 2.667e-15 | 6.284e-15 | 5.554e-15 |
| Hook\_1498 | 2.045e-15 | 1.386e-15 | 6.499e-16 | 2.045e-15 | 6.235e-14 |
| bone010 | 3.119e-15 | 1.621e-15 | 8.648e-16 | 3.119e-15 | 2.873e-15 |
| Emilia\_923 | 1.719e-17 | 1.166e-17 | 7.955e-18 | 1.719e-17 | 6.242e-15 |
| StocF-1465 | 3.252e-15 | 1.221e-15 | 9.786e-16 | 3.252e-15 | 1.885e-12 |
| parabolic\_fem | 5.285e-16 | 3.248e-16 | 2.821e-16 | 5.285e-16 | 2.978e-16 |
| Flan\_1565 | 2.329e-15 | 1.224e-15 | 7.886e-16 | 2.329e-15 | 2.841e-15 |
| boneS10 | 2.390e-15 | 2.033e-15 | 1.516e-15 | 2.390e-15 | 2.300e-15 |
| bone010\_M | 2.281e-11 | 7.633e-10 | 6.102e-10 | 2.281e-11 | 2.118e-10 |

Можно заметить, что относительная погрешность алгоритма с параллелизмом по потокам графического процессора имеет большую относительную погрешность, чем остальные методы. Тем не менее, её значения достаточно малы, чтобы сделать вывод о корректной работе алгоритма.

## 5.5 Анализ производительности программных реализаций алгоритмов решения разреженных СЛАУ

## 5.5.1 Системы с одной правой частью

В данном разделе собраны результаты экспериментов, проведённых для разреженных систем с одной правой частью. Результаты, собранные в таблицах 6-8, получены на тестовой системе СК «Лобачевский».

Время работы последовательных алгоритмов представлено в таблице 6. Полученные результаты проиллюстрированы на рисунке 5.

Таблица 6. Время работы последовательных алгоритмов,  
решающих задачу . Время представлено в секундах.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название матрицы | Базовый | Супернодальный | Супернодальный BLAS | trsv |
| **audikw\_1** | 2,611 | 1,992 | 1,761 | 1,742 |
| **Hook\_1498** | 2,726 | 3,005 | 2,442 | 2,060 |
| **bone010** | 2,101 | 1,745 | 1,723 | 1,490 |
| **Emilia\_923** | 2,904 | 2,258 | 1,962 | 2,386 |
| **StocF-1465** | 1,618 | 1,316 | 1,180 | 1,596 |
| **parabolic\_fem** | 0,074 | 0,070 | 0,131 | 0,055 |
| **Flan\_1565** | 1,843 | 1,655 | 2,073 | 2,108 |
| **boneS10** | 0,520 | 0,311 | 0,387 | 0,401 |
| **bone010\_M** | 0,515 | 0,492 | 0,473 | 0,588 |

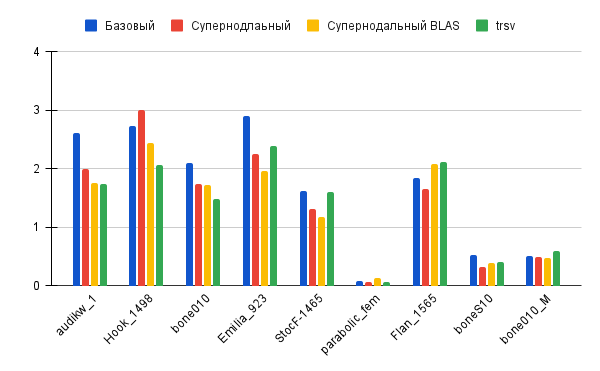


Рисунок 5. Сравнение времени работы последовательных алгоритмов,  
решающих задачу .

Базовый алгоритм показывает наихудшие результаты практически на всех тестовых матрицах. Супернодальная версия алгоритма в среднем показывает производительность на 20% выше базовой. Если алгоритме используются библиотечные реализации функций cblas\_dtrsm и cblas\_dgemm, удаётся повысить производительность примерно на 30% для более заполненных матриц. Супернодальные алгоритмы в ряде случаев работают быстрее алгоритма из библиотеки MKL, однако при их использовании необходимо потратить дополнительную память на хранение массива супернодов. Реализация метода из MKL лишена этого недостатка.

Рассмотрим далее результаты параллельного алгоритма с барьерной синхронизацией. Результаты собраны в таблице 7 и проиллюстрированы на рисунке 6.

Таблица 7. Время работы параллельного алгоритма с барьерной синхронизацией,  
решающего задачу . Время представлено в секундах.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Число потоков/  Название матрицы | 1 | 2 | 4 | 8 | 16 | 24 | 32 |
| **audikw\_1** | 4,358 | 3,232 | 2,803 | 2,621 | 2,599 | 2,542 | 2,488 |
| **Hook\_1498** | 5,638 | 4,193 | 3,588 | 3,492 | 3,099 | 3,213 | 3,079 |
| **bone010** | 3,862 | 2,863 | 2,447 | 2,400 | 2,110 | 2,197 | 2,166 |
| **Emilia\_923** | 6,023 | 4,521 | 3,826 | 3,649 | 3,536 | 3,499 | 3,394 |
| **StocF-1465** | 3,951 | 2,902 | 2,489 | 2,230 | 2,186 | 2,333 | 2,334 |
| **parabolic\_fem** | 0,212 | 0,142 | 0,110 | 0,094 | 0,092 | 0,087 | 0,091 |
| **Flan\_1565** | 5,431 | 3,962 | 3,351 | 3,129 | 2,808 | 2,939 | 2,833 |
| **boneS10** | 1,160 | 0,858 | 0,715 | 0,646 | 0,682 | 0,631 | 0,687 |
| **bone010\_M** | 1,538 | 1,119 | 0,913 | 0,841 | 0,795 | 0,840 | 0,819 |

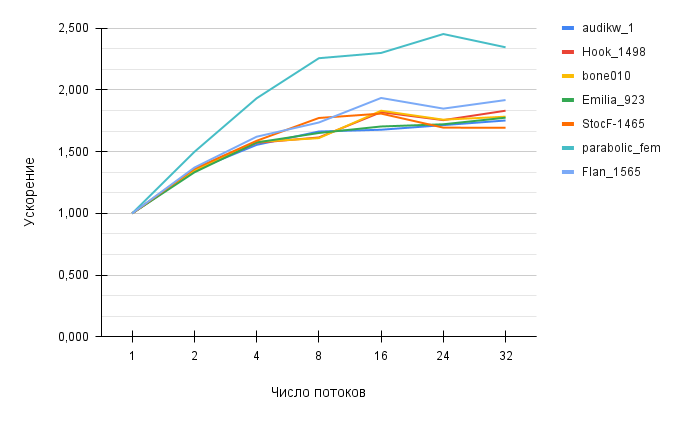


Рисунок 6. Масштабируемость параллельного алгоритма с барьерной синхронизацией,  
 решающего задачу . Время представлено в секундах.

Для всех тестовых матриц, кроме parabolic\_fem, ускорение от параллелизма не превышает значения 1,93. Из графика видно, что использование гипер-трединга не даёт выигрыша в скорости, что может говорить о высокой загруженности вычислительных ядер.

В случае parabolic\_fem на 24 вычислительных потоках удаётся достичь ускорения в 2,45 раза по сравнению с параллельной версией кода, запущенной на одном потоке.

В таблице 8 приведены результаты запусков алгоритма mkl\_sparse\_d\_trsv.

Таблица 8. Время работы алгоритма mkl\_sparse\_d\_trsv,  
решающего задачу . Время представлено в секундах.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Число потоков/  Название матрицы | 1 | 2 | 4 | 8 | 16 | 24 | 32 |
| **audikw\_1** | 1,742 | 1,711 | 1,739 | 1,751 | 1,568 | 1,712 | 1,722 |
| **Hook\_1498** | 2,061 | 2,176 | 2,167 | 2,180 | 2,133 | 2,150 | 2,151 |
| **bone010** | 1,491 | 1,490 | 1,495 | 1,500 | 1,361 | 1,470 | 1,364 |
| **Emilia\_923** | 2,386 | 2,369 | 2,369 | 2,388 | 2,311 | 2,376 | 2,347 |
| **StocF-1465** | 1,596 | 1,589 | 1,587 | 1,597 | 1,572 | 1,460 | 1,464 |
| **parabolic\_fem** | 0,055 | 0,059 | 0,055 | 0,060 | 0,063 | 0,064 | 0,060 |
| **Flan\_1565** | 2,108 | 2,175 | 2,169 | 2,181 | 1,922 | 2,203 | 1,976 |
| **boneS10** | 0,401 | 0,404 | 0,401 | 0,413 | 0,397 | 0,400 | 0,398 |
| **bone010\_M** | 0,589 | 0,589 | 0,589 | 0,596 | 0,560 | 0,545 | 0,537 |

Можно заметить, что библиотечная функция демонстрирует плохую масштабируемость, не превышающую в пике значения 1,1. Однако даже будучи запущенной на одном потоке, mkl\_sparse\_d\_trsv обгоняет параллельную версию алгоритма с синхронизацией, запущенную на 16 потоках.

Перейдём к рассмотрению параллельных алгоритмов, ориентированных на использование GPU. Результаты вычислительных экспериментов собраны в таблице 9 и проиллюстрированы на рисунке 7.

Таблица 9. Время работы параллельных алгоритмов,  
использующих технологию OpenCL. Запуски проведены на рабочей станции.  
Время представлено в секундах.

|  |  |  |
| --- | --- | --- |
| Название матрицы | Параллелизм по варпам | Параллелизм по потокам |
| **audikw\_1** | 0,577 | 1,996 |
| **Hook\_1498** | 0,772 | 2,589 |
| **bone010** | 0,587 | 1,724 |
| **Emilia\_923** | 0,627 | 2,555 |
| **StocF-1465** | 0,680 | 1,944 |
| **parabolic\_fem** | 0,081 | 0,052 |
| **Flan\_1565** | 0,945 | 2,657 |
| **boneS10** | 0,341 | 0,507 |
| **bone010\_M** | 0,409 | 0,572 |



Рисунок 7. Сравнение времени работы параллельных алгоритмов,  
использующих технологию OpenCL. Решается задача .  
Время представлено в секундах.

Согласно статье [814], алгоритм с параллелизмом на уровне потоков должен показывать ускорение в среднем в 4.97 раза по сравнению с алгоритмом, использующим параллелизм на уровне варпов. Полученные результаты демонстрируют существенное замедление первого алгоритма для всех тестовых матриц, кроме parabolic\_fem. Это может быть вызвано значительным увеличением числа ненулевых элементов матрицы в процессе разложения Холецкого (см. таблицу 3.). Чтобы проверить эту гипотезу, был проведён ещё один эксперимент с использованием оригинальных матриц коллекции SuiteSparse[16]. Результаты собраны в таблицe 10 и проиллюстрированы на рисунке 8.

Таблица 10. Время работы параллельных алгоритмов, использующих технологию OpenCL. Запуски проведены на Nvidia RTX 2080 Super. Время представлено в секундах.

|  |  |  |
| --- | --- | --- |
| Название матрицы | Параллелизм по варпам | Параллелизм по потокам |
| **audikw\_1** | 0,173 | 0,109 |
| **Hook\_1498** | 0,111 | 0,054 |
| **bone010** | 0,158 | 0,098 |
| **Emilia\_923** | 0,047 | 0,023 |
| **StocF-1465** | 0,06 | 0,01 |
| **parabolic\_fem** | 0,003 | 0,001 |
| **Flan\_1565** | 0,312 | 0,192 |
| **boneS10** | 0,096 | 0,025 |
| **bone010\_M** | 0,062 | 0,009 |

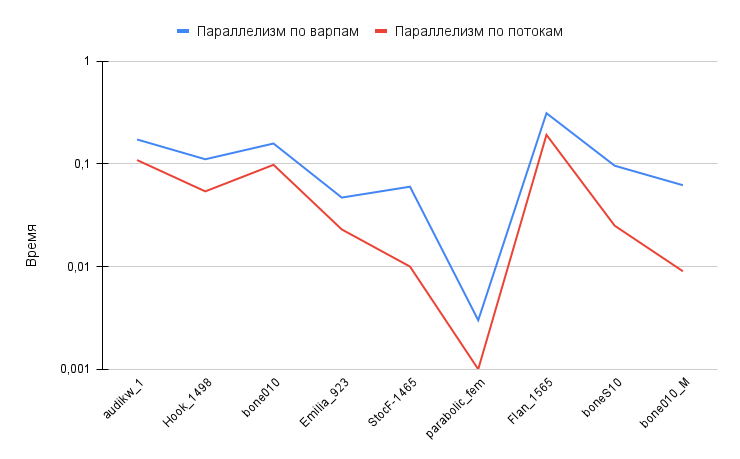


Рисунок 8. Сравнение времени работы параллельных алгоритмов,  
использующих технологию OpenCL. Решается задача .  
Запуски проведены на Nvidia RTX 2080 Super. Время представлено в секундах.

При использовании оригинальных разреженных матриц алгоритм с параллелизмом на уровне потоков показывает ускорение в 1,58 – 6,88 раз по сравнению с алгоритмом, использующим параллелизм на уровне варпов.

Таким образом, выбор алгоритма для применения на практике обусловлен свойствами разреженных матриц, такими как их размерность и заполненность, поэтому он может быть сделан только после проведения предварительных экспериментов.

## 5.5.2 Системы с множественными правыми частями

В данном разделе собраны результаты экспериментов, проведённых для разреженных систем со ста правыми частями. Результаты, собранные в таблицах 11-13, получены на тестовой системе СК «Лобачевский».

Время работы последовательных алгоритмов представлено в таблице 11. Полученные результаты проиллюстрированы на рисунке 9.

Таблица 11. Время работы последовательных алгоритмов,  
решающих задачу . Время представлено в секундах.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название матрицы | Базовый | Супернодальный | Супернодальный BLAS | trsm |
| **audikw\_1** | 159,052 | 148,703 | 95,220 | 237,187 |
| **Hook\_1498** | 214,525 | 205,839 | 125,331 | 340,989 |
| **bone010** | 131,673 | 126,720 | 86,827 | 235,173 |
| **Emilia\_923** | 237,865 | 224,713 | 122,031 | 366,351 |
| **StocF-1465** | 140,854 | 131,540 | 108,165 | 207,173 |
| **parabolic\_fem** | 5,200 | 5,995 | 16,787 | 2,960 |
| **Flan\_1565** | 208,813 | 196,850 | 132,957 | 293,477 |
| **boneS10** | 40,332 | 35,150 | 41,393 | 34,145 |
| **bone010\_M** | 51,104 | 46,744 | 51,553 | 46,606 |

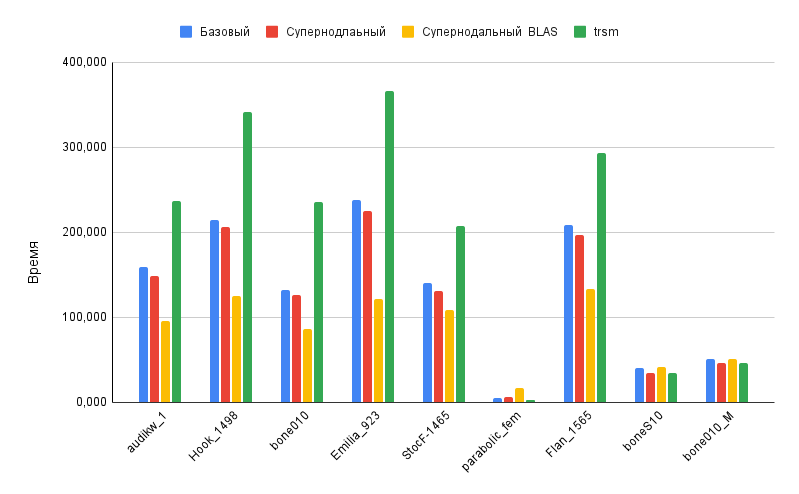


Рисунок 9. Сравнение времени работы последовательных алгоритмов,  
решающих задачу .

Как и в случае решения системы c одной правой частью, базовый метод показывает наихудшие результаты. Супернодальная версия алгоритма, реализованная без использования сторонних библиотечных функций, показывает результаты, сравнимые с результатами базовой версии. При использовании в алгоритме оптимизированных функций cblas\_dtrsm и cblas\_dgemm удаётся повысить производительность примерно на 46% по сравнению с базовой версией.

Ускорения библиотечной функции mkl\_sparse\_d\_trsm не наблюдается. Возможно, данная функция не была распараллелена. Тем не менее, в случае решения системы относительно небольшой размерности, алгоритм из библиотеки MKL показывает наилучшие результаты.

В разделе 3 была упомянута возможность модификации последовательных алгоритмов для случая с несколькими правыми частями. Для базового и супернодального алгоритмов, а также для параллельного алгоритма с барьерной синхронизацией были подготовлены соответствующие реализации. Результаты запусков оптимизированных алгоритмов отражены в таблице 12.

Таблица 12. Время работы оптимизированных последовательных алгоритмов,  
решающих задачу . Время представлено в секундах.

|  |  |  |
| --- | --- | --- |
| Название матрицы | Базовый оптимизированный | Супернодальный оптимизированный |
| **audikw\_1** | 154,517 | 117,789 |
| **Hook\_1498** | 196,607 | 149,137 |
| **bone010** | 141,48 | 99,703 |
| **Emilia\_923** | 227,859 | 183,361 |
| **StocF-1465** | 159,633 | 120,091 |
| **parabolic\_fem** | 5,135 | 4,17 |
| **Flan\_1565** | 201,419 | 141,808 |
| **boneS10** | 45,88 | 27,829 |
| **bone010\_M** | 51,431 | 37,919 |

В ряде случаев оптимизированная версия базового алгоритма показывает прирост производительности на 3-9%, однако на тестовых матрицах bone010, boneS10 и StocF-1465 наблюдается ухудшение скорости вычислений на 7-12%.

Оптимизация супернодального алгоритма привела к ускорению вычислений на всех матрицах в среднем на 28%. Значительный прирост производительности можно объяснить сокращением числа загрузок правых частей в кеш (в неоптимизированной версии множество раз загружается только один вектор, в то время как в оптимизированной версии загружается вся матрица целиком).

Вернёмся к рассмотрению параллельного алгоритма с барьерной синхронизацией. Результаты экспериментов собраны в таблице 13 и проиллюстрированы на рисунке 9.

Таблица 13. Время работы параллельного алгоритма с барьерной синхронизацией,  
решающего задачу . Время представлено в секундах.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Число потоков/  Название матрицы | 1 | 2 | 4 | 8 | 16 | 24 | 32 |
| **audikw\_1** | 164,861 | 92,127 | 65,218 | 59,230 | 49,169 | 52,547 | 59,629 |
| **Hook\_1498** | 233,278 | 137,255 | 99,130 | 94,001 | 58,932 | 71,525 | 80,744 |
| **bone010** | 167,285 | 97,217 | 75,542 | 78,151 | 40,490 | 48,191 | 52,308 |
| **Emilia\_923** | 256,861 | 155,396 | 112,958 | 107,939 | 90,014 | 101,065 | 117,630 |
| **StocF-1465** | 172,856 | 98,587 | 69,759 | 56,377 | 45,997 | 59,742 | 72,272 |
| **parabolic\_fem** | 10,949 | 5,796 | 3,338 | 2,238 | 2,405 | 1,966 | 2,272 |
| **Flan\_1565** | 236,494 | 133,524 | 93,314 | 76,280 | 47,891 | 58,969 | 76,089 |
| **boneS10** | 50,432 | 28,749 | 18,327 | 14,011 | 12,580 | 14,302 | 15,275 |
| **bone010\_M** | 60,551 | 33,275 | 20,753 | 16,329 | 14,503 | 18,813 | 20,357 |

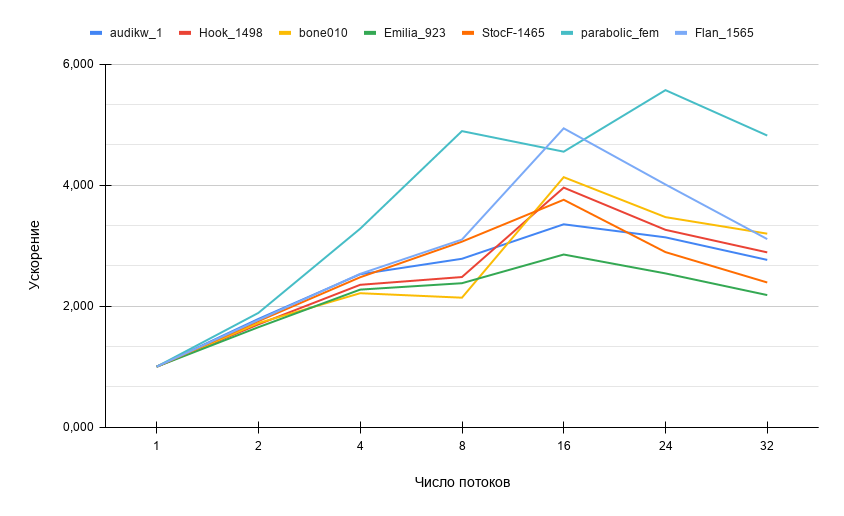


Рисунок 10. Масштабируемость параллельного алгоритма с барьерной синхронизацией,  
 решающих задачу . Время представлено в секундах.

В целом, производительность алгоритма аналогична случаю с одной правой частью. На большинстве матриц максимальное ускорение достигается при использовании 16 вычислительных потоков и варьируется в интервале 2,9–4,9. При использовании 24 и 32 потоков происходит деградация масштабируемости.

Отдельного рассмотрения требует матрица parabolic\_fem. Пиковое ускорение алгоритма при тестировании на этой матрице составило 5,7. При переходе на второй сокет наблюдается снижение ускорения на 7%, в то время как для всех других тестовых матриц наблюдается повышение ускорения на 11 – 90%. Это может быть связано с аллокацией матриц. Изначально данные загружаются в кеш первого процессора. Если они не помещаются в кеш, они будут частично размещены в кеше соседнего процессора. Так как часть данных находится географически далеко от процессора, к которому привязан вычислительный поток, возрастает время чтения/записи. Из всех тестовых матриц только parabolic\_fem полностью поместилась в кеш одного процессора, поэтому только она показывает высокую масштабируемость в пределах одного сокета. При переходе на второй сокет происходит производительность падает, т. к. требуется затратить дополнительное время на дальний доступ. Для всех остальных матриц ситуация противоположная.

Оптимизированная параллельная версия также демонстрирует пиковое ускорение на 16 вычислительных потоках OpenMP. Его значение примерно на 21% выше, чем у неоптимизированной версии.

Рассмотрим теперь алгоритм без синхронизаций с параллелизмом по варпам графического процессора. Результаты экспериментов приведены в таблице 14.

Таблица 14. Время работы алгоритма OpenCL с параллелизмом на уровне варпов.  
решается задача . Эксперименты проведены на рабочей станции.  
Время представлено в секундах.

|  |  |
| --- | --- |
| Название матрицы | Параллелизм по варпам |
| **audikw\_1** | 4,008 |
| **Hook\_1498** | 5,107 |
| **bone010** | 3,281 |
| **Emilia\_923** | 12,402 |
| **StocF-1465** | 9,100 |
| **parabolic\_fem** | 0,195 |
| **Flan\_1565** | 5,300 |
| **boneS10** | 0,984 |
| **bone010\_M** | 2,244 |

Вычисления, проведённые на графическом процессоре, бессмысленно сравнивать с результатами, полученными при использовании CPU. Однако стоит заметить, что алгоритмы, реализованные при помощи технологии OpenCL и запущенные на видеокарте, демонстрируют результаты, в десятки раз превышающие наилучшие результаты, полученные с использованием CPU-ориентированных алгоритмов.

По результатам экспериментов, представленных в этом разделе, можно сделать следующие основные выводы:

1. Последовательные алгоритмы:
   1. Супернодальный алгоритм, разработанный с использованием библиотечных реализаций функций BLAS, показывает наилучший результат.
   2. Базовый метод демонстрирует наихудшую производительность, однако он может быть применим в случае отсутствия информации о расположении супернодов.
   3. Модификация версий кода, предназначенных для решения систем с несколькими правыми частями, обеспечивает дополнительный выигрыш в скорости на 3–28%, в зависимости от метода и тестовой матрицы.
2. Параллельные алгоритмы:
   1. Алгоритм с барьерной синхронизацией достигает пикового ускорения в среднем в 4 раза на 16 вычислительных потоках.
   2. Алгоритмы, реализованные с использованием технологии OpenCL и запущенные на графическом процессоре, демонстрируют наилучшие результаты среди всех рассмотренных методов.

Заключение

В работе получены следующие основные результаты:

1. Изучены различные подходы к решению разреженных треугольных систем алгебраических уравнений с плотной правой частью.
2. Разработаны различные варианты последовательных алгоритмов решения системы для и .
3. Рассмотрен параллельный алгоритм с синхронизацией по уровням графа зависимостей.
4. Рассмотрены современные подходы к решению системы, ориентированные на использование графического процессора.
5. Подтверждена корректность полученных результатов.
6. Проведены вычислительные эксперименты на современных серверных и графических процессорах. Выполнен анализ производительности разработанных программных реализаций.
7. Сделаны выводы о практической применимости различных подходах к решению разреженных систем.

Список литературы

1. Писсанецки С., Икрамов Х. Д., Капорин И. Е. Технология разреженных матриц: Пер. с англ. – Мир, 1988.
2. Вержбицкий В. М. Основы численных методов. – Москва : Высшая школа, 2005.
3. Schreiber R. A new implementation of sparse Gaussian elimination //ACM Transactions on Mathematical Software (TOMS). – 1982. – Т. 8. – №. 3. – С. 256-276.
4. Anderson E., Saad Y. Solving sparse triangular linear systems on parallel computers //International Journal of High Speed Computing. – 1989. – Т. 1. – №. 01. – С. 73-95.
5. Demmel J. W., Gilbert J. R., Li X. S. An asynchronous parallel supernodal algorithm for sparse gaussian elimination //SIAM Journal on Matrix Analysis and Applications. – 1999. – Т. 20. – №. 4. – С. 915-952.
6. Naumov M. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU //NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011. – 2011. – Т. 1.
7. Liu W. et al. A synchronization-free algorithm for parallel sparse triangular solves //European Conference on Parallel Processing. – Springer, Cham, 2016. – С. 617-630.
8. Su J. et al. CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs //49th International Conference on Parallel Processing-ICPP. – 2020. – С. 1-11.
9. Chen X., Wang Y., Yang H. Parallel sparse direct solver for integrated circuit simulation. – Springer International Publishing, 2017.
10. Povelikin R., Lebedev S., Meyerov I. Multithreaded Multifrontal Sparse Cholesky Factorization Using Threading Building Blocks //Russian Supercomputing Days. – Springer, Cham, 2019. – С. 75-86.
11. Park J. et al. Sparsifying synchronization for high-performance shared-memory sparse triangular solver //International Supercomputing Conference. – Springer, Cham, 2014. – С. 124-140.
12. Sparse matrix storage format. [Электронный ресурс]. URL: <https://scc.ustc.edu.cn/zlsc/sugon/intel/mkl/mkl_manual/index.htm#GUID-9FCEB1C4-670D-4738-81D2-F378013412B0.htm>.
13. Intel® Math Kernel Library 2020 – C. BLAS and Sparse BLAS Routines. [Электронный ресурс]. URL:<https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-routines.html>
14. The OpenCL™ Specification. [Электронный ресурс]. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL\_API.pdf
15. Thread Affinity interface. [Электронный ресурс]. URL: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html
16. SuiteSparse Matrix Collection. [Электронный ресурс]. URL: <https://sparse.tamu.edu/>

Приложение А. Фрагменты исходного кода программы.

double supernodal\_upper\_opt(size\_t sn, int\* supernodes, double\* x, double\* val, int\* row, uint64\_t\* col\_index,

int n, int rhs) {

double t1 = omp\_get\_wtime();

int dim;

double sum;

int cnt;

for (int snode\_ind = (int)sn - 1; snode\_ind >= 0; --snode\_ind) {

cnt = 0;

dim = supernodes[snode\_ind + 1] - supernodes[snode\_ind];

for (int i = supernodes[snode\_ind]; i < supernodes[snode\_ind] +

dim; ++i) {

for (int rh = 0; rh < rhs; ++rh){

sum = 0.;

for (int j=col\_index[i]+dim-cnt; j< col\_index[i + 1];

++j) {

sum += val[j] \* x[rh \* n + row[j]];

}

x[rh \* n + i] -= sum;

}

cnt++;

}

for (int rh = 0; rh < rhs; ++rh){

x[rh \* n + supernodes[snode\_ind] + dim - 1] =

x[rh \* n + supernodes[snode\_ind] + dim - 1] /

val[col\_index[supernodes[snode\_ind] + dim - 1]];

}

cnt = 0;

for (int j = supernodes[snode\_ind] + dim - 2; j >=

supernodes[snode\_ind]; --j) {

for (int rh = 0; rh < rhs; ++rh){

sum = 0.;

for(int k=col\_index[j]+1; k<=col\_index[j]+1+cnt;++k) {

sum += val[k] \* x[rh \* n + row[k]];

}

x[rh \* n + j] = (x[rh \* n + j] - sum) /

val[col\_index[j]];

}

cnt++;

}

}

return omp\_get\_wtime() - t1;

}

#pragma OPENCL EXTENSION cl\_khr\_fp64 : enable

#define WARP\_SIZE 64

#define SUBSTITUTION\_BACKWARD 1

#define OPT\_WARP\_NNZ 1

#define OPT\_WARP\_RHS 2

#define OPT\_WARP\_AUTO 3

inline

void atom\_add\_d\_fp64(volatile \_\_global double\* val,

double delta)

{

union { double f; ulong i; } old;

union { double f; ulong i; } new;

do

{

old.f = \*val;

new.f = old.f + delta;

} while (atom\_cmpxchg((volatile \_\_global ulong\*)val, old.i, new.i) != old.i);

}

inline

void atom\_add\_s\_fp64(volatile \_\_local double\* val,

double delta)

{

union { double f; ulong i; } old;

union { double f; ulong i; } new;

do

{

old.f = \*val;

new.f = old.f + delta;

} while (atom\_cmpxchg((volatile \_\_local ulong\*)val, old.i, new.i) != old.i);

}

\_\_kernel

void sptrsv\_syncfree\_opencl\_analyser(\_\_global const int\* d\_cscRowIdx,

const int m,

const int nnz,

\_\_global int\* d\_graphInDegree)

{

const int global\_id = get\_global\_id(0);

if (global\_id < nnz)

{

atomic\_fetch\_add\_explicit((atomic\_int\*)&d\_graphInDegree[d\_cscRowIdx[global\_id]], 1,

memory\_order\_acq\_rel, memory\_scope\_device);

}

}

\_\_kernel

void sptrsv\_syncfree\_opencl\_executor(\_\_global const int\* d\_cscColPtr,

\_\_global const int\* d\_cscRowIdx,

\_\_global const VALUE\_TYPE\* d\_cscVal,

\_\_global volatile int\* d\_graphInDegree,

\_\_global volatile VALUE\_TYPE\* d\_left\_sum,

const int m,

\_\_global const VALUE\_TYPE\* d\_b,

\_\_global VALUE\_TYPE\* d\_x,

\_\_local volatile int\* s\_graphInDegree,

\_\_local volatile VALUE\_TYPE\* s\_left\_sum,

const int warp\_per\_block)

{

const int global\_id = get\_global\_id(0);

const int local\_id = get\_local\_id(0);

int global\_x\_id = global\_id / WARP\_SIZE;

if (global\_x\_id >= m) return;

// substitution is forward or backward

global\_x\_id = m - 1 - global\_x\_id;

// Initialize

const int local\_warp\_id = local\_id / WARP\_SIZE;

const int lane\_id = (WARP\_SIZE - 1) & local\_id;

int starting\_x = (global\_id / (warp\_per\_block \* WARP\_SIZE)) \*

warp\_per\_block;

starting\_x = m - 1 - starting\_x;

// Prefetch

const int pos = d\_cscColPtr[global\_x\_id + 1] - 1;

const VALUE\_TYPE coef = (VALUE\_TYPE)1 / d\_cscVal[pos];

if (local\_id < warp\_per\_block) {

s\_graphInDegree[local\_id] = 1;

s\_left\_sum[local\_id] = 0; }

barrier(CLK\_LOCAL\_MEM\_FENCE);

// Consumer

int loads, loadd;

do {

// busy-wait

} while ((loads = atomic\_load\_explicit((atomic\_int\*)&s\_graphInDegree[local\_warp\_id],

memory\_order\_acquire, memory\_scope\_work\_group)) !=

(loadd = atomic\_load\_explicit((atomic\_int\*)&d\_graphInDegree[global\_x\_id],

memory\_order\_acquire, memory\_scope\_device)));

VALUE\_TYPE xi = d\_left\_sum[global\_x\_id] + s\_left\_sum[local\_warp\_id];

xi = (d\_b[global\_x\_id] - xi) \* coef;

// Producer

const int start\_ptr = d\_cscColPtr[global\_x\_id];

const int stop\_ptr = d\_cscColPtr[global\_x\_id + 1] - 1;

for (int jj = start\_ptr + lane\_id; jj < stop\_ptr; jj += WARP\_SIZE) {

const int j = stop\_ptr - 1 - (jj - start\_ptr);

const int rowIdx = d\_cscRowIdx[j];

const bool cond = (rowIdx > starting\_x - warp\_per\_block);

if (cond) {

const int pos = starting\_x - rowIdx;

atom\_add\_s\_fp64(&s\_left\_sum[pos], xi \* d\_cscVal[j]);

mem\_fence(CLK\_LOCAL\_MEM\_FENCE);

atomic\_fetch\_add\_explicit((atomic\_int\*)&s\_graphInDegree[pos], 1,

memory\_order\_acquire, memory\_scope\_work\_group);

}

else {

atom\_add\_d\_fp64(&d\_left\_sum[rowIdx], xi \* d\_cscVal[j]);

mem\_fence(CLK\_GLOBAL\_MEM\_FENCE);

atomic\_fetch\_sub\_explicit((atomic\_int\*)&d\_graphInDegree[rowIdx], 1,

memory\_order\_acquire, memory\_scope\_device);

}

}

// Finish

if (!lane\_id) d\_x[global\_x\_id] = xi;

}

#pragma OPENCL EXTENSION cl\_khr\_global\_int32\_base\_atomics : enable

#pragma OPENCL EXTENSION cl\_khr\_global\_int32\_extended\_atomics : enable

#pragma OPENCL EXTENSION cl\_khr\_int64\_base\_atomics : enable

#pragma OPENCL EXTENSION cl\_khr\_int64\_extended\_atomics : enable

#pragma OPENCL EXTENSION cl\_khr\_fp64 : enable

#ifndef VALUE\_TYPE

#define VALUE\_TYPE double

#endif

#define WARP\_SIZE 64

\_\_kernel

void sptrsv\_syncfree\_opencl\_executor(

\_\_global const int \*d\_csrRowPtr,

\_\_global const int \*d\_csrColIdx,

\_\_global const VALUE\_TYPE \*d\_csrVal,

\_\_global volatile int \*d\_get\_value,

const int m,

\_\_global const VALUE\_TYPE \*d\_b,

\_\_global volatile VALUE\_TYPE \*d\_x)

{

int global\_id = get\_global\_id(0);

if(global\_id>=m)

return;

global\_id = m - 1 - global\_id;

//printf("%d\n", global\_id);

int col,j,i;

VALUE\_TYPE xi;

VALUE\_TYPE left\_sum=0;

i=global\_id;

j=d\_csrRowPtr[i+1]-1;

//printf("I: %d J: %d\n", i, j);

while(j >= d\_csrRowPtr[i])

{

col=d\_csrColIdx[j];

//printf("%d\n", d\_get\_value[col]);

if(atomic\_load\_explicit((atomic\_int\*)&d\_get\_value[col],memory\_order\_acquire, memory\_scope\_device)==1)

{

left\_sum+=d\_csrVal[j]\*d\_x[col];

j--;

col=d\_csrColIdx[j];

}

if(i==col)

{

//printf("%d\n", i);

xi = (d\_b[i] - left\_sum) / d\_csrVal[d\_csrRowPtr[i]];

d\_x[i] = xi;

mem\_fence(CLK\_GLOBAL\_MEM\_FENCE);

d\_get\_value[i]=1;

j--;

}

}

}