Содержание

1. ВВЕДЕНИЕ

Алгебра разреженных матриц (Sparse algebra) является важным разделом математики, имеющее практическое применение. Разреженные матрицы встречаются при постановке и решении задач в различных научных и технических областях. Например, при возникновении оптимизационных задач большой размерности с линейными ограничениями. Такие матрицы формируются при численном решении дифференциальных уравнений в частных производных, а также в теории графов.

В первую очередь разберемся с понятием разреженная матрица. В одном из источников дается такое понятие разреженной матрицы: разреженной называют матрицу, имеющую малый процент ненулевых элементов. В ряде других источников встречается такая формулировка: матрица *NxN* называется разреженной, если количество ее ненулевых элементов есть *O(N).* Нельзя сказать, что данные определения являются точными в математическом смысле. Причиной этому можно считать то, что как показывает практика, классификация матрицы зависит не только от количества ненулевых элементов, но и от размера матрицы, распределения ненулевых элементов, архитектуры вычислительной системы и используемых алгоритмов [1].

В зависимости от поставленной задачи неизменным остается одно - важно определиться с выбором структуры хранения. Это может быть обычное хранение двумерного массива или более сложная схема, учитывающая особенности разреженности матрицы. Так как работа предполагает вычисления с использованием матриц, размерность которых предполагает, что плотное хранение просто недопустимо, а количество ненулевых элементов значительно мало, то первым важным шагом становится анализ всех эффективных способов хранения такой матрицы.

Эффективные методы хранения и обработки разреженных матриц на протяжении последних десятилетий вызывают интерес у широкого круга исследователей. Становится известно, что для решения определенных задач необходимо существенно усложнять как методы хранения, так и алгоритмы обработки. Многие тривиальные задачи при работе с матрицами в разреженном случае становятся весьма сложными. Основным правилом считается конструировать алгоритмы таким образом, чтобы общая память и число арифметических операций зависели линейно или были ограничены некоторой медленно растущей функцией от числа ненулевых элементов, но не от общего числа ее элементов [2].

Вычисления с симметричными разреженными матрицами пользуются большой популярностью, поскольку это позволяет в два раза уменьшить требования к памяти для хранения исходных данных. Но при этом за компактность приходится платить повышенной сложностью в реализации.

В данной работе рассматривается оптимизация алгоритма матрично-векторного умножения для симметрично-разреженных матриц. Ставится цель разработки программного кода с учетом возможностей современных многоядерных архитектур и инструментов для разработки программ в системах с общей памятью.

2. ПОСТАНОВКА ЗАДАЧИ

В данном дипломном проекте требуется оптимизировать алгоритм умножения симметричной разреженной матрицы на заполненный вектор-столбец при выборе способа хранения и методов распараллеливания.

В нашей задаче матрица обладает симметричным портретом (рисунок 2.1), поэтому размеры матрицы определяются как *NxN*.

Пусть *A* - матрица размера *NxN*, в которой процент ненулевых элементов мал, *b* - вектор-столбец размера *N*, *c* - результирующий вектор размера *N*. Будем считать, что элементы матрицы и вектора - числа типа *double*.

Требуется найти вектор *c = A*b* (знак * - векторное умножение).

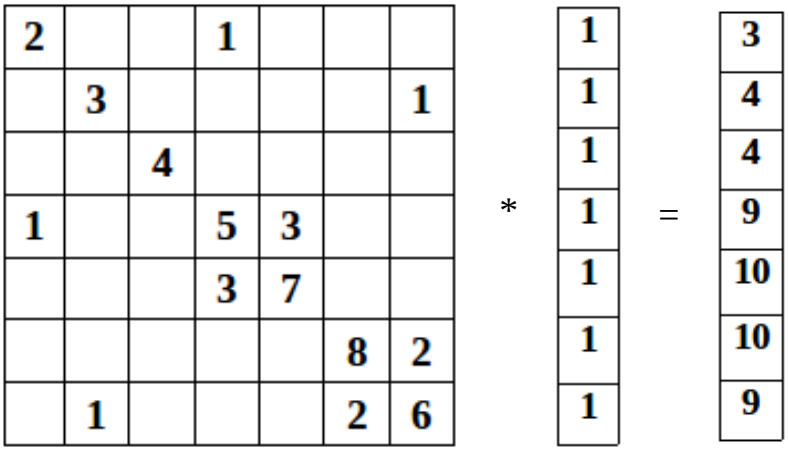

Рисунок 2.1 – Умножение матрицы на вектор

3. РАЗРЕЖЕННАЯ МАТРИЦА

Можно дать множество определений термину «матрица». В математике матрицей называется таблица (система элементов) размера *nxm*, в программировании матрица — это двумерный массив, а, например, в электронике матрица представляется набором проводников, которые можно замкнуть при их пересечении.

Матрицы встречаются повсеместно. Поэтому неважно программист ты, математик или простой обыватель, в любом случае часто работаешь с матрицами.

Нами, конечно, матрица рассматривается с точки зрения математики и программирования. По количеству отличных от нуля элементов матрицу можно разделить на два вида: плотная матрица (или по-другому матрица общего вида) и разреженная матрица.

Именно о разреженных матрицах и пойдет речь.

Разреженной матрицей называется матрица, в которой количество нулевых элементов намного преобладает над ненулевыми. В таких матрицах количество ненулевых элементов составляет до 20% [4].

Примеры разреженных матриц, доступных в коллекции университета Флориды[4], где *n* - размер матрицы, *nz* - количество ненулевых элементов, представлены на рисунке 3.1:

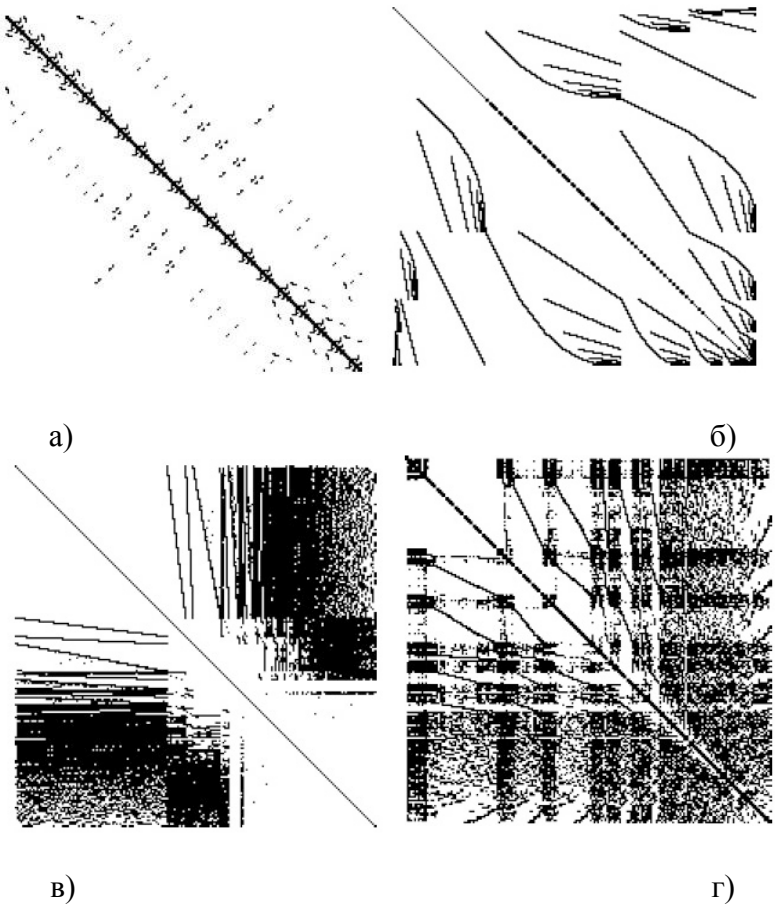

Рисунок 3.1 - Разреженные матрицы : а) и в) матрица shallow_water2 – n 81920, nz 204800; б) и г) parabolic_fem – n 525825, nz 2100225

На рисунке 3.1 представлены разреженные симметричные и положительно определенные матрицы.

Симметричной называется квадратная матрица A, у которой элемент a_{ij} равен элементу a_{ji} относительно главной диагонали.

Теперь разберемся, для чего нужны разреженные матрицы. Для начала вспомним два важных факта:

- Память под массив, написанный на языке программирования C, выделяется сразу после его создания
- Многомерные массивы занимают большие объемы памяти.

Получается, что размер самого большого массива, который мы хотим написать, ограничен объемом допустимой памяти. Предположим, что нам удалось разместить массив большого размера в памяти, но это дало ряд проблем, потому что при создании такого массива память, занятая им, оказывается недоступной для остальной части программы, что уменьшает объем доступных ресурсов. А этот факт, соответственно, отрицательно сказывается на производительности программы. В ситуациях, когда работа ведется не со всеми элементами матрицы, выделение памяти под весь массив является нецелесообразной тратой системных ресурсов.

Чтобы не возникало проблем потребности памяти при работе с разреженными матрицами, были придуманы некоторые приемы работы с ними. Все они основаны на одном: выделение памяти под элементы массива происходит только при необходимости. Поэтому самым главным преимуществом при работе с разреженной матрицей можно считать то, что для ее хранения памяти необходимо столько, сколько элементов действительно используются. Так, при сложении числа с нулем, мы получаем само число, а при умножении этого числа на нуль, мы получаем нуль. Соответственно, работа с разреженной матрицей предполагает обработку только ненулевых элементов этой матрицы.

Существует множество примеров приложений, в которых ведется работа с разреженными матрицами. Многие из них относятся к инженерным задачам, однако есть одно приложение, которое достаточно широко используется - электронная таблица. Вспомним, что матрица в такой таблице имеет большую размерность, хотя на практике используется лишь малая ее часть. При использовании разреженности память под элемент выделяется только по необходимости, поэтому электронная таблица занимает минимально необходимую память.

Приведем еще один пример, почему нецелесообразно хранить разреженную матрицу целиком. Допустим, что размер матрицы порядка 10^6 элементов. Тогда из определения разреженной матрицы следует, что количество ненулевых элементов составит 20%, а это $2 * 10^5$элементов. Пусть элементы будут типа double, следовательно, 1 элемент = 8 байт. Подсчитаем объем необходимой памяти при хранении матрицы целиком: $8 * 10^6$ байт, а при хранении ненулевых элементов: $8 * 2 * 10^5 = 16 * 10^5$байт. Из расчетов видно, что $16 * 10^5 < 8 * 10^6$. Отсюда и получаем, что выражение «чем больше, тем лучше» не работает, для эффективной работы необходимо хранить то, что действительно задействовано в работе.

Работа с разреженной матрицей предполагает особые способы доступа к элементам. Для упрощения этой работы были разработаны форматы хранения, наиболее популярные из которых и будут рассмотрены дальше.

4. ФОРМАТЫ ХРАНЕНИЯ РАЗРЕЖЕННЫХ МАТРИЦ

Эффективность алгоритма напрямую зависит от выбора формата хранения разреженной матрицы. Каждая схема была придумана для достижения определенных целей, таких как простота, универсальность, производительность или удобный доступ к элементам. В зависимости от поставленной задачи мы и выбираем формат хранения.

Существует пять наиболее распространенных схем хранения данных: Coordinate Storage (COO - координатный формат), Compressed Row Storage (CRS - разреженный строчный формат), Compressed Column Storage (CCS - разреженный столбцовый формат), Row-wise Representation Complete and Ordered (RR(C)O - упорядоченный строчечный формат) и Column-wise Representation Complete and Ordered (CR(C)O - упорядоченный столбцовый формат) [5].

В рамках дипломной работы были использованы Coordinate Storage (COO), Compressed Row Storage (CRS), Row-wise Representation Complete and Ordered (RR(C)O) и его модификация Row-wise Representation Lower Unordered Storage (RR(L)U).

Как уже отмечалось ранее, матрица имеет симметричный портрет, поэтому при реализации рассматривались элементы только нижнего треугольника, что позволило сократить объем потребляемой памяти.

При дальнейшем описании мы будем рассматривать матрицы размера NxN, где NZ элементов являются ненулевыми, причем $NZ << N^2$.

4.1 Координатный формат (COO)

Coordinate Storage -один из самых простых в понимании и реализации форматов хранения. Матрица представляется в виде трех массивов, первый из которых хранит значения ненулевых элементов матрицы, второй и третий номера строк и столбцов этих элементов соответственно. Пример хранения нижнего треугольника симметричной матрицы в координатном формате представлен на рисунке 4.1:

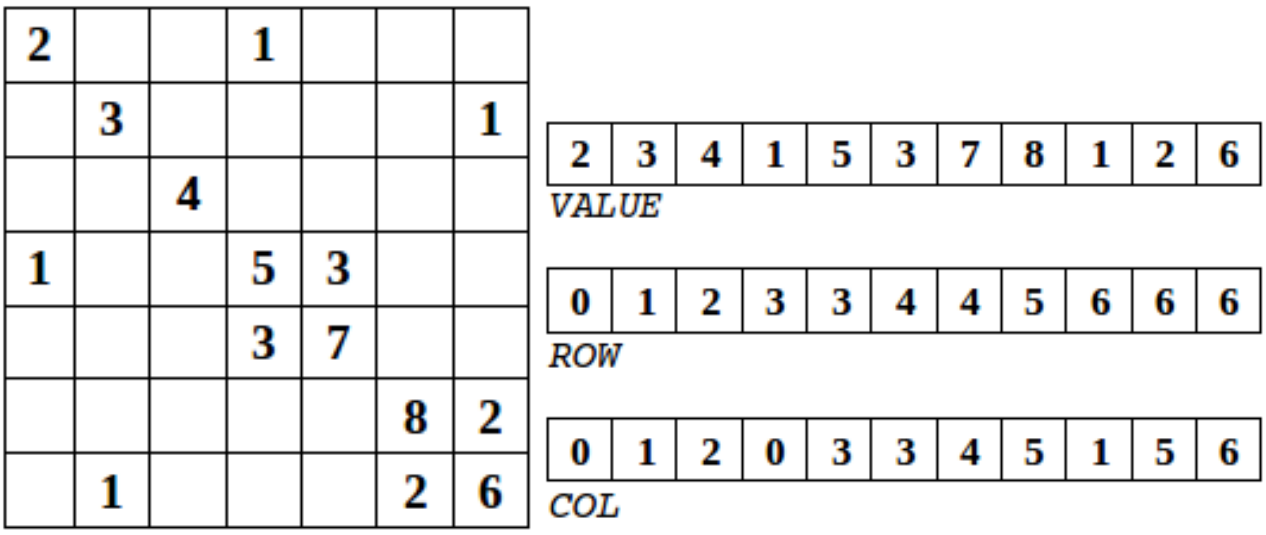

Рисунок 4.1 - Координатный формат

Массив *VALUE* хранит значения ненулевых элементов, *ROW* — индексы строк, а *COL* — индексы столбцов. Размеры массивов соответствуют количеству ненулевых элементов нижнего треугольника.

Оценим объем необходимой памяти (M):

N — размер матрицы, *NZ* — количество ненулевых элементов, тогда при хранении нижнего треугольника необходимо хранить количество элементов равное:

$$\frac{NZ-N}{2}+N=\frac{NZ-N+2N}{2}=\frac{NZ+N}{2} \quad (4.1)$$

Плотное представление: $M = 8\,N^2$ байт

Координатный формат (все ненулевые): $M = 8\,NZ + 4\,NZ + 4\,NZ = 16\,NZ$ байт.

Координатный формат (нижний треугольник): $M = 8\,\frac{NZ+N}{2} + 4\,\frac{NZ+N}{2} + 4\,\frac{NZ+N}{2} = 8\,NZ + 8\,N$ байт.

В координатном формате элементы можно хранить как в упорядоченном виде, так и в неупорядоченном. Неупорядоченное представление существенно упрощает операции вставки/удаления новых элементов, но приводит к поиску методом перебора. Упорядоченный вариант позволяет быстрее находить все элементы нужной строки (столбца, если упорядочивать сначала по столбцам), но приводит к перестановкам при вставках/удалениях элементов. В целом координатный формат достаточно прост, что является его несомненным достоинством.

4.2 Формат хранения CRS

Следующий формат хранения широко известен под названием Compressed Row Storage (CRS), устраняет некоторые недоработки координатного формата. В данной схеме хранения также представлено три массива: массив *VALUE* хранит значения ненулевых элементов нижнего треугольника, *COL* хранит индексы столбцов этих элементов, а вот массив *ROW_INDEX* меняет номера строк на индекс начала каждой строки или, по-другому, содержит количество элементов в каждой строке нижнего треугольника (рисунок 4.2)

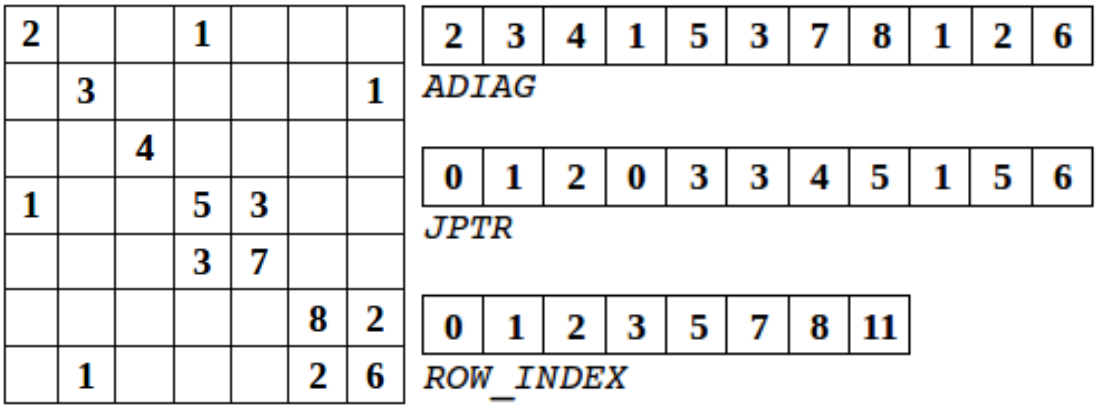

Рисунок 4.2 – Формат CRS

Количество элементов массива *ROW_INDEX* равно *N+1*, размеры массивов *VALUE* и *COL* не отличаются от координатного формата и остаются $\frac{NZ+N}{2}$ элементов. При этом следует заметить, что элементы строки i в массиве VALUE находятся по индексам от *ROW_INDEX[i]* до *ROW_INDEX[i+1]-1* включительно. Следовательно, обрабатывается случай пустых строк. Также добавлен еще один элемент в массив *ROW_INDEX* для устранения особенности обращения к последней строке. Элемент *ROW_INDEX[N+1]* хранит в себе номер последнего ненулевого элемента матрицы плюс 1, что соответствует количеству ненулевых элементов $\frac{NZ+N}{2}$.

Оценим объем необходимой памяти (M):

Полное представление: $M = 8\ N^2$ байт.

Формат CRS (все ненулевые): $M = 8\ NZ + 4NZ + 4\ (N+1) = 12\ NZ + 4\ N + 4$ байт.

Формат CRS (нижний треугольник): $M = 8\ \frac{NZ+N}{2} + 4\ \frac{NZ+N}{2} + 4\ (N+1) = 6\ NZ + 6\ N + 4\ N + 4 = 6\ NZ + 10\ N + 4$ байт.

Если массивы сформировать не для строк, а для столбцов, получится разреженный столбцовый формат (CSC).

4.3 Формат RR(C)O

Еще один формат хранения разреженных матриц - это формат RR(C)O. В большинстве литературы данный формат называют модификацией формата CRS. Действительно, принципы хранения матрицы совпадают, но есть одна отличительная особенность: в данном формате выделяется четвертый массив *ADIAG* размера *N*, который хранит диагональные элементы, соответственно в массиве *VALUE* будут храниться ненулевые элементы, не принадлежащие главной диагонали.

Существует ряд модификаций данного формата:

- существует как упорядоченный строчечный формат, рассмотренный только что, так и неупорядоченный RR(C)U (Row-wise Representation Complete and Unordered);
- при записи в массивы элементов не по строчкам, а по столбцам, получится упорядоченный - Column-wise Representation Complete and Ordered (CR(C)O) или неупорядоченный - Column-wise Representation Complete and Unordered (CR(C)U) столбцовый формат;
- существует модификация, предназначенная для симметричных разреженных матриц.

4.4 Формат RR(L)U

Сокращенное название формата RR(L)U происходит от английского словосочетания «Row — wise Representation, Lower, Unordered» (строчное представление, нижний треугольник, неупорядоченное). Данный формат является модификацией формата RR(C)U и предназначен для представления симметричных разреженных матриц. Отличительной чертой данного формата

является вынесенный массив с диагональными элементами, в остальном схема похожа на CRS (рисунок 4.3).

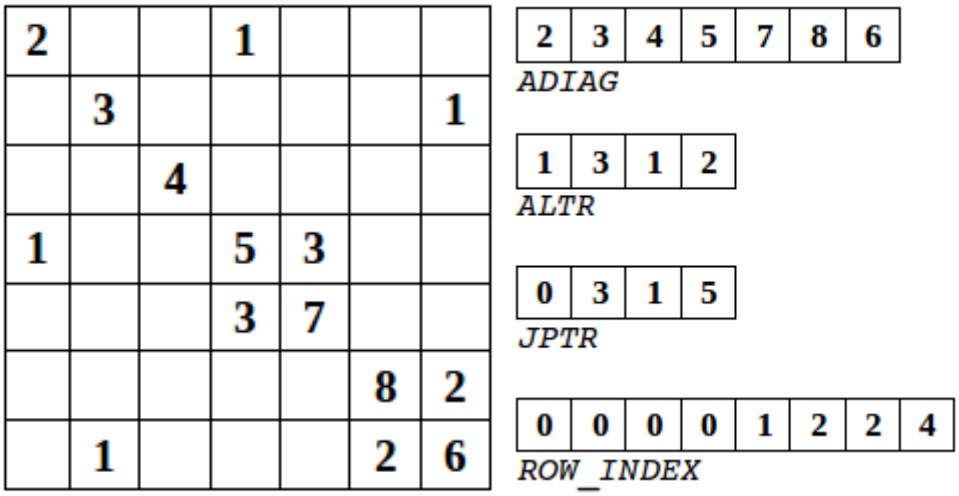

Рисунок 4.3 - RR(L)U

Массив *ADIAG* размера *N* содержит диагональные элементы матрицы. Массив *ALTR* значения, а *JPTR* индексы столбцов элементов нижнего треугольника, массив *ROW_INDEX* размера *N+1* имеет ту же систему представления, что и в предыдущем формате.

Преимуществами данного формата является достаточно быстрое обращение к диагональным элементам, меньший объем необходимой памяти для хранения (M).

Формат RR(L)U (нижний треугольник): $M = 8\,N + 8\,\frac{NZ-N}{2} + 4\,\frac{NZ-N}{2} + 4\,(N+1) = 8\,N + 6\,(NZ-N) + 4\,(N+1) = 8\,N + 6\,NZ - 6\,N + 4\,N + 4 = 6\,N + 6\,NZ + 4$ байт.

5. ПОСЛЕДОВАТЕЛЬНЫЙ АЛГОРИТМ

5.1 Общая схема решения задачи

Приступим к обсуждению алгоритма умножения разреженной матрицы на вектор. Для начала необходимо понять, как выполняется умножение по определению (рисунок 6.1).

$$A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_{1n} \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_1 + a_{12} \cdot b_2 + \dots + a_{1n} \cdot b_n \\ a_{21} \cdot b_1 + a_{22} \cdot b_2 + \dots + a_{2n} \cdot b_n \\ \dots \quad \dots \quad \dots \quad \dots \quad \dots \quad \dots \\ a_{m1} \cdot b_1 + a_{m2} \cdot b_2 + \dots + a_{mn} \cdot b_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \dots \\ c_{1m} \end{pmatrix}$$

Рисунок 6.1 - Умножение матрицы на вектор

Определение предполагает, что в результате умножения матрицы *A* размера *nxn* и вектора *b*, состоящего из *n* элементов, получается вектор *c* размера *n*, каждый *i-ый* элемент которого есть результат скалярного умножения *i-й* строки матрицы *A* (обозначим эту строчку a_i) и вектора *b*.

$$\mathrm{c}_i = (a_i, b_i) = \sum_{j=1}^{n} a_{ij} b_j \ , 1 \ \leq i \ \leq m \tag{6.1}$$

Получение результирующего вектора предполагает повторение *n* однотипных операций по умножению строки матрицы *A* на элементы вектора *b*. Каждая такая операция включает умножение элементов строки матрицы на элемент вектора (*n* операций) и последующее суммирование полученных произведений (*n-1* операций).

Общее количество необходимых скалярных операций есть величина

$$T = n(2n - 1) \tag{6.2}$$

Какие особенности вносит разреженное представление?

Во-первых, структура данных, построенная на основе формата RR(L)U, предполагает хранение ненулевых элементов матрицы, что намного уменьшает количество необходимых арифметических операций.

При вычислении скалярных произведений не нужно умножать нули и накапливать полученный нуль в частичную сумму, что положительно влияет на сокращение времени счета.

Во-вторых, стоит помнить, что из-за симметричности портрета хранение происходит только элементов нижнего треугольника, работа с верхним треугольником предполагает транспонирование. Транспонирование в данном случае происходит быстро, потому что работа со столбцами матрицы устроена эффективно. Для каждого элемента мы всегда можем узнать номер соответствующей строки и столбца и, соответственно, с легкостью можем также работать и с симметричным ему элементом.

5.2 Программная реализация в формате RR(L)U

При реализации алгоритма использовалась структура хранения матрицы A (см. рисунок 6.2)

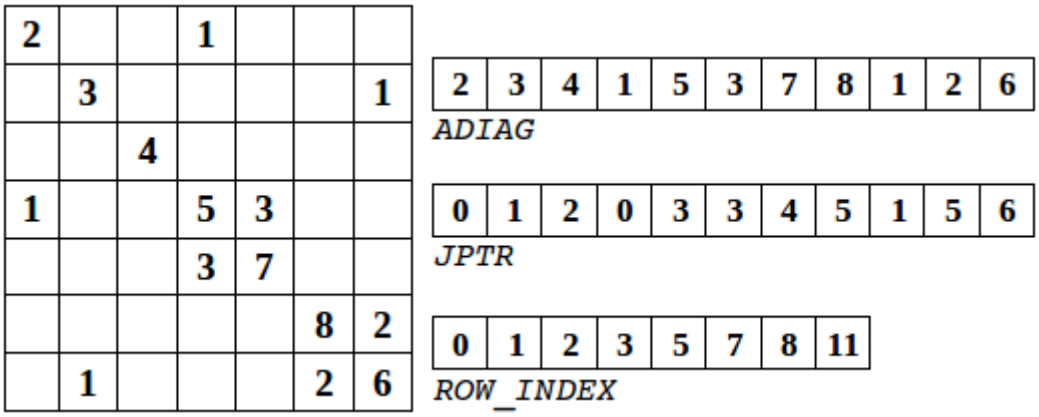

Рисунок 6.2 - Матрица в формате RR(L)U

Обсудим общую схему умножения разреженной матрицы на вектор:

1) В функцию MULTIPLICATE поступают массивы ADIAG, ALTR, JPTR, ROW_INDEX, хранящие исходную матрицу, вектор b, а также размер матрицы N.

2) Обрабатываются диагональные элементы, хранящиеся в массиве ADIAG, результат записывается в результирующий вектор.

3) Происходит умножение элементов нижнего треугольника, представленных в массиве ALTR, на соответствующие элементы вектора b, с последующей записью в результирующий вектор.

4) Обрабатываются симметричные элементы из верхнего треугольника.

Листинг 6.1 – Реализация функции MULTIPLICATE на языке Си

```
void multiplicate(double * adiag, double * altr, int *
jptr, int * iptr, int _diag, double * x, double * z)
{
    int i, j;
    for(i = 0; i < _diag; i++)
     z[i] = 0;
    for(i = 0; i < _diag; i++)
    {
          z[i] = x[i] * adiag[i];
          for(j = iptr[i]; j < iptr[i + 1]; j++)
          {
               z[i] = z[i] + x[jptr[j]] * altr[j];
               z[jptr[j]] = z[jptr[j]] + x[i] * altr[j];
          }
    }
}
```

5.3 Дополнительные функции

Проверка правильности результата один из важных аспектов при решении любых задач. При работе с большими матрицами необходимо гарантировать истину полученного результата.

Было реализовано ряд дополнительных функций, в которых осуществлялась проверка результирующего вектора.

5.3.1 Функция MULTY_COO()

1) Функция MULTY_COO осуществляет умножение матрицы, хранящейся в координатном формате, на вектор. На вход функции поступают три массива ELEM, ELEM_I, ELEM_J и их размер, а также вектор b. ELEM-значения ненулевых элементов матрицы, ELEM_I-номера строк, ELEM_J-номера столбцов соответствующих элементов.
2) Осуществляется умножение элементов массива ELEM с учетом положения в массивах, хранящих индексы строк и столбцов, на элементы вектора b.
3) Результат умножения записывается в результирующий вектор.

Листинг 6.2 – Реализация функции MULTY_COO на языке Си

```
void multy_coo(double * elem, int * elem_i, int * elem_j,
int _diag, int size, double * x, double * z)
{
    int k;

    for(k = 0; k < _diag; k++)
     z[k] = 0;

    for(k = 0; k < size; k++)
    {
          z[elem_i[k]] += elem[k] * x[elem_j[k]];
          if(elem_i[k] != elem_j[k])
          {
               z[elem_j[k]] += elem[k] * x[elem_i[k]];
          }

    }
}
```

5.3.2 Функция CONTROL_RESULT()

1) Функция CONTROL_RESULT осуществляет проверку подлинности результата, полученного при умножении матрицы на вектор в формате RR(L)U, с результатом функции MULTY_COO.
2) На вход функции поступают два массива RES_RR, RES_COO, а также размер SIZE.
3) Происходит сравнение элементов и вывод соответствующего сообщения.

Листинг 6.3 – Реализация функции CONTROL_RESULT на языке Си

```
void control_result(double *res_rr, double *res_coo, int
size)
{
     int i;
     for(i = 0; i < size; i++)
          if(res_rr[i] == res_coo[i])
               printf("good\n");
          else printf ("error in res_coo or res_rr\n")
}
```

6. ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ

В данном разделе будет рассматриваться реализация параллельного алгоритма умножения разреженной матрицы на вектор. Параллельный алгоритм представляется набором подзадач, каждая из которых выполняется в отдельном потоке. Потоки независимы между собой.

6.1 Общая схема разделения данных

При выполнении параллельных алгоритмов умножения матрицы A на вектор b, между процессами необходимо разделить не только элементы матрицы, но и элементы вектора b и вектора результата c. Элементы вектора c можно продублировать по всем процессорам, составляющих многопроцессорную вычислительную систему. При блочном разбиении данных из n элементов каждому процессору достается непрерывная последовательность из k элементов (предполагается, что количество элементов n нацело делится на число процессоров, поэтому $n = pk$).

Необходимо пояснить, почему дублирование вектора c между процессами является допустимым решением. Вектор c состоит из n элементов, т.е. содержит столько же элементов, что и одна строка матрицы или один столбец матрицы. Если процессор хранит строку (столбец) матрицы и одиночный элемент вектора c, то общее число хранимых элементов составляет $O(n)$. Если же процессор будет хранить строку (столбец) матрицы и вектор c, состоящий из n элементов, то общее число хранимых элементов также будет порядка $O(n)$. Таким образом, при дублировании и разделении вектора требования к объему памяти из одного класса сложности. Такая структура избавит нас от зависимости по данным, возникающей при формировании результирующего вектора.

6.2 Общая схема умножения при разделении данных по строкам

Для выполнения подзадачи скалярного произведения процессор должен хранить соответствующую строку матрицы A и копию вектора b. После выполнения каждая базовая подзадача будет определять один из элементов результирующего вектора c.

Как уже говорилось, предусматривается дублирование вектора c по всем процессам. Поэтому после получения результатов в каждом из процессоров, необходимо провести сбор данных.

В общем виде схема информационного взаимодействия продемонстрирована на рисунке 7.1. На рисунке представлено количество потоков, равное количеству строк матрицы. Каждый поток независимо от остальных обрабатывает свою часть задачи. Данный способ распараллеливания алгоритма достаточно просто реализуется на практике, т.к. организовать вычисления можно таким образом, что результат каждого из потоков никак не будет зависеть от решения других. Это является несомненным преимуществом данной реализации.

Такой вид распараллеливания наиболее часто встречается на практике.

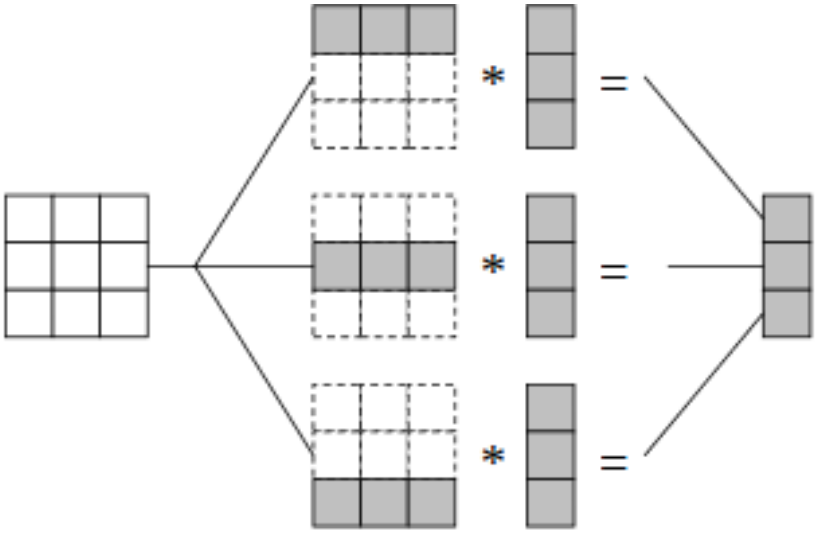

Рисунок 7.1 - Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

6.3 Анализ эффективности общего алгоритма

При анализе эффективности параллельных алгоритмов строится два типа оценок. Первая оценивается в количестве вычислительных операций, которые необходимы для получения результата, не учитывая время на передачу данных между процессорами, а также длительность вычислительных операций принимается одинаковой для всех вычислительных операций. Для этого типа оценок важен прежде всего порядок сложности алгоритма, поэтому константы в полученных соотношениях в расчет не берутся.

Второй тип оценок представляет более точные соотношения для предсказания времени выполнения алгоритма. На данном этапе оценка получается путем уточнения выражений, полученных на предыдущем этапе. Для этого вводятся параметры, которые задают длительность выполнения операций, строятся оценки трудоемкости коммуникационных операций, указываются все необходимые константы. Далее проверяется точность полученных выражений путем проведения вычислительных экспериментов.

Алгоритм умножения матрицы на вектор при строчном разбиении обладает достаточно хорошей «локализацией вычислений». Каждый поток обрабатывает только «свои» данные, не требуя данных, которые обрабатывает другой поток, нет обмена данными между потоками, и не возникает необходимости синхронизации. Поэтому в данной задаче практически не возникает накладных расходов на организацию параллелизма, и с уверенностью можно ожидать линейного ускорения. Но на деле такого ускорения не достигается, потому что время решение задачи одним потоком получается из времени, когда процессор выполняет вычисления, и времени, когда считываются необходимые данные из оперативной памяти в кэш память, при этом время чтения может превосходить время счета.

Перейдем к трудоемкости алгоритма умножения матрицы на вектор. Разберем случай, когда матрица A квадратная, тогда последовательный алгоритма умножения имеет сложность:

$$T_1 = n^2 \tag{7.1}$$

При параллельной реализации каждый процессор (p — количество процессоров) выполняет умножение только полосы матрицы A на вектор b, размер этих полос составляет n/p строк. Тогда вычислительная сложность параллельного алгоритма составит:

$$T_p = \frac{n^2}{p} \tag{7.2}$$

С учетом этой оценки показатели ускорения и эффективности параллельной реализации имеют вид:

$$S_p = n^2\left(\frac{n^2}{p}\right) = p, E_p = \frac{n^2}{p(\frac{n^2}{p})} = 1 \tag{7.3}$$

Полученные оценки времени вычислений выражены в количестве операций, не учитывая при этом время передачи данных. Исходя из обсуждения выше, предположим, что выполняемые операции умножения и сложения имеют одинаковую длительность t, и система является однородной (все процессоры обладают одинаковой производительностью). Время выполнения последовательного алгоритма складывается из времени вычислений и времени доступа к памяти:

$$T_1 = T_{calc} + T_{mem} \tag{7.4}$$

С учетом введенных предположений время выполнения алгоритма, связанное непосредственно с вычислениями будет выглядеть так:

$$T_{calc} = n(2n-1)t \tag{7.5}$$

Если разделить объем извлекаемых данных из памяти M (в данном случае $M = 8\ n^2$) на пропускную способность β, то получим время доступа к памяти. Кроме этого, необходимо принять во внимание, что при выполнении операций доступа к памяти может возникнуть задержка (латентность доступа к памяти) α. В этом случае время доступа к памяти можно оценить как:

$$T_{mem} = \frac{M}{\beta} + n^2\alpha \tag{7.6}$$

Объединяя T_{calc} и T_{mem}, получим суммарное время выполнения алгоритма:

$$T_1 = n(2n-1)t + \frac{M}{\beta} + n^2\alpha \tag{7.7}$$

Рассмотрим теперь вопрос оценки параллельного алгоритма матрично-векторного умножения. Время выполнения алгоритма, связанное с вычислениями, составляет:

$$T_{calc} = \left\lceil \frac{n}{p} \right\rceil (2n-1)t \tag{7.8}$$

Общее время выполнения параллельного алгоритма:

$$T_p = \left\lceil \frac{n}{p} \right\rceil (2n-1)t + \frac{M}{\beta} + n^2\alpha \tag{7.9}$$

6.4 Алгоритм умножения при разделении данных по строкам

Представим реализацию алгоритма умножения симметричной разреженной матрицы на вектор. Необходимо вспомнить, что матрица хранится в формате RR(L)U (см. рисунок 6.2).

Рассмотрим схему информационного взаимодействия при хранении матрицы в формате RR(L)U (рисунок 7.2).

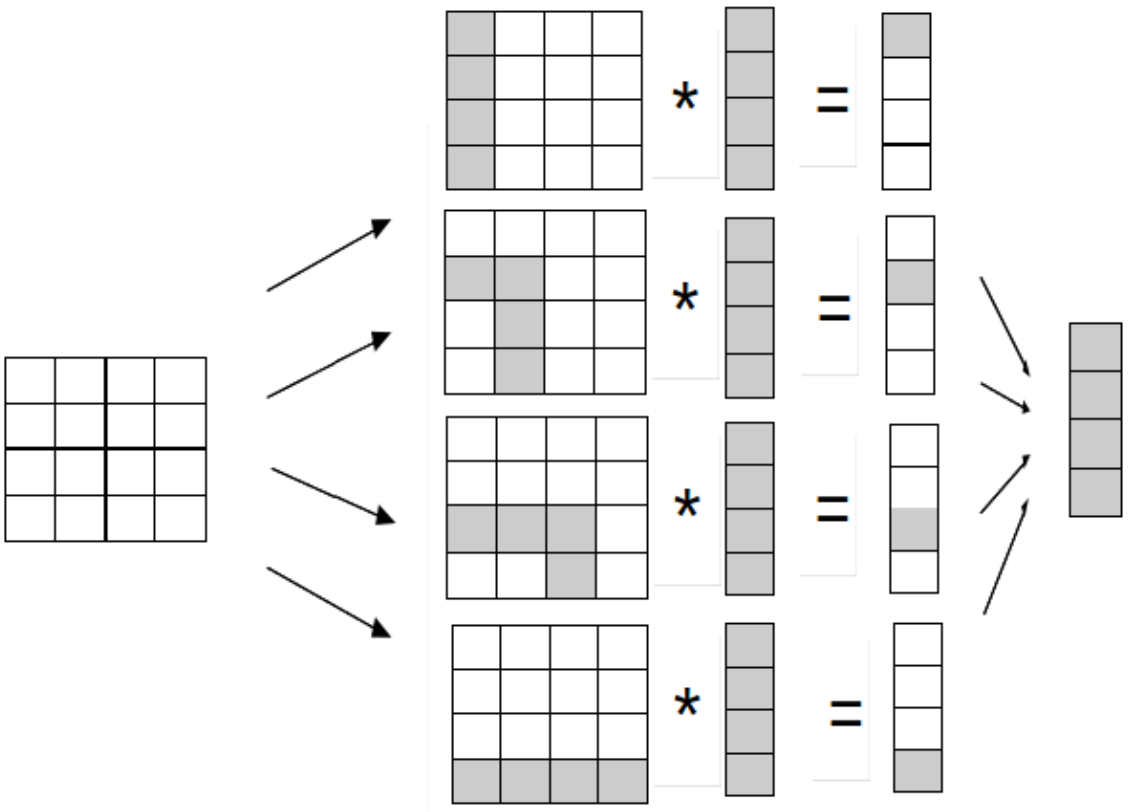

Рисунок 7.2 - Организация вычислений при выполнении алгоритма умножения матрицы на вектор с построчным разделением данных и форматом хранения матрицы RR(L)U

Наша задача разбивается на подзадачи, т.е. каждый процесс обрабатывает $\frac{n}{p}$ строк исходной матрицы *A*. Формат хранения предполагает, что работа

производится только с нижним треугольником. Поэтому необходимо организовать такой доступ к данным строки, при котором не возникнет зависимости по данным, а обработка данных будет производиться как при обычном делении на строки.

На рисунке 7.2 показано, как происходит умножение на каждом из процессов:

1) Производим умножение данных, находящихся в горизонтальной строке и принадлежащих нижнему треугольнику.
2) Как только обработается диагональный элемент, вычисления продолжаются по вертикальному столбцу. Вертикальные элементы являются симметричными для элементов, принадлежащих горизонтальной строке, обрабатываемой в пункте 1.
3) Продублируем результирующий вектор по количеству потоков. После завершения вычислений просуммируем все вектора для получения конечного решения.

Итак, нам стало известно, что из-за симметричности портрета матрицы мы можем работать как со строками (горизонталь), так и со столбцами (вертикаль). Поэтому, разумнее всего, при работе с нижним треугольником умножать сначала элементы в строке, а потом в соответствующем столбце.

Листинг 7.1 – Реализация функции умножения симметричной разреженной матрицы на вектор с разделением данных по строкам на языке Си

```
void multy_by_rows(double * adiag, double * altr, int * jptr, int * iptr, int Size, int size_altr, double * pVector, double * Result, int GridThreadsNum)
{
    int i, j, m, p, k;

    for(i = 0; i < Size; i++)
          Result[i] = pVector[i] * adiag[i];

    for(i = 0; i < Size; i++)
    {
          for(j = iptr[i]; j < iptr[i + 1]; j++)
          {
               Result[i] += pVector[jptr[j]] * altr[j];
          }
          for(k = i; k < Size; k++)
          {
               for(p = iptr[k]; p < iptr[k + 1]; p++)
                    if(jptr[p] == i)
                     Result[jptr[p]] += pVector[k]*altr[p];

          }
    }

}
```

6.4.1 Анализ эффективности алгоритма

Проведем анализ данной программной реализации. Построим два типа оценок. Первая оценка предполагает количество вычислительных операций без учета времени, тогда для последовательного алгоритма с учетом формата хранения RR(L)U получим:

$$T_1 = 2\sum_{i=0}^{n-1} 2nz_i + n \tag{7.10}$$

где *n*-размер матрицы *A*, *nzi*-количество ненулевых элементов в каждой строке *i*.

Для параллельного алгоритма оценка вычислительных операций выглядит следующим образом:

$$T_p = 2\sum_{i=0}^{\frac{n}{p}-1} nz_i + \frac{n}{p}, \tag{7.11}$$

где p-количество потоков.

Для вычисления следующей оценки предположим, что выполняемые операции умножения и сложения имеют одинаковую длительность *t*, и система является однородной. Тогда время вычислений для последовательного алгоритма представляется:

$$T_{\text{calc}} = 2\text{t}\sum_{i=0}^{n-1} 2nz_i + nt \tag{7.11}$$

И для параллельного соответственно:

$$T_p(calc) = 2\text{t}\sum_{i=0}^{\frac{n}{p}-1} nz_i + \frac{nt}{p} \tag{7.12}$$

Количество ненулевых элементов при нашей реализации составит *NZ*. Тогда объем извлекаемых данных, с учетом типа элементов *double*, составит *8NZ*.

Если разделить объем извлекаемых данных из памяти на пропускную способность β, то получим время доступа к памяти. Кроме этого, принимаем во внимание латентность доступа к памяти α. Тогда время доступа к памяти можно оценить как:

$$T_{mem} = \frac{8NZ}{\beta} + NZ\alpha \tag{7.13}$$

Объединяя *Tcalc* и *Tmem,* получим суммарное время выполнения алгоритмов:

$$T_1 = 2\text{t}\sum_{i=0}^{n-1} 2nz_i + nt + \frac{8NZ}{\beta} + NZ\alpha \tag{7.14}$$

$$T_p = 2\mathrm{t}\sum_{i=0}^{\frac{n}{p}-1} nz_i + \frac{nt}{p} + \frac{8NZ}{\beta} + NZ\alpha \quad (7.15)$$

6.5 Общая схема умножения при разделении данных по блокам

Рассмотрим следующий метод параллельной реализации, основанный на разделении данных на прямоугольные фрагменты (блоки).

Данный метод предполагает блочную схему представления данных, матрица A представляется как набор блоков:

$$A = \begin{matrix} A_{00} & A_{02} & \dots A_{0q-1} \\ \dots & \dots & \dots \\ A_{s-11} & A_{s-12} & \dots A_{s-1q-1} \end{matrix},$$

где $A_{ij}, 0 \ll i < s, 0 \ll j < q$, есть блок матрицы:

$$A_{ij} = \begin{matrix} a_{i0j0} & a_{i0jl} & \dots a_{i0jl-1} \\ \dots & \dots & \dots \\ a_{ik-1j0} & a_{ik-1jl} & \dots a_{ik-1jl-1} \end{matrix},$$

$$i_v = ik + v, 0 \le v < k, k = \frac{m}{s}, j_u = jl + u, 0 \le u \le l, l = \frac{n}{q},$$

где предполагается, что p = sq, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = ks$ и $n = lq$).

В данном случае базовые подзадачи определяются на основе операций, производящихся с блоком матрицы A. Каждая подзадача должна иметь доступ к блоку такого же размера вектора b.

После умножения блоков матрицы A на вектор b каждая подзадача будет иметь вектор частичных результатов, определяемый следующей формулой:

$$c'_v(i,j) = \sum_{u=0}^{l-1} a_{ivju} b_{ju}, \quad (7.16)$$

$$i_v = ik + v, 0 \le v < k, k = \frac{m}{s}, j_u = jl + u, 0 \le u \le l, l = \frac{n}{q}$$

Предполагается, что каждый поток вычисляет отдельный блок вектора частичных результатов, независимо от остальных потоков. Результирующий вектор можно получить путем суммирования элементов блока вектора частичных результатов с одинаковыми индексами по всем подзадачам.

Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных представлена на рисунке 7.3:

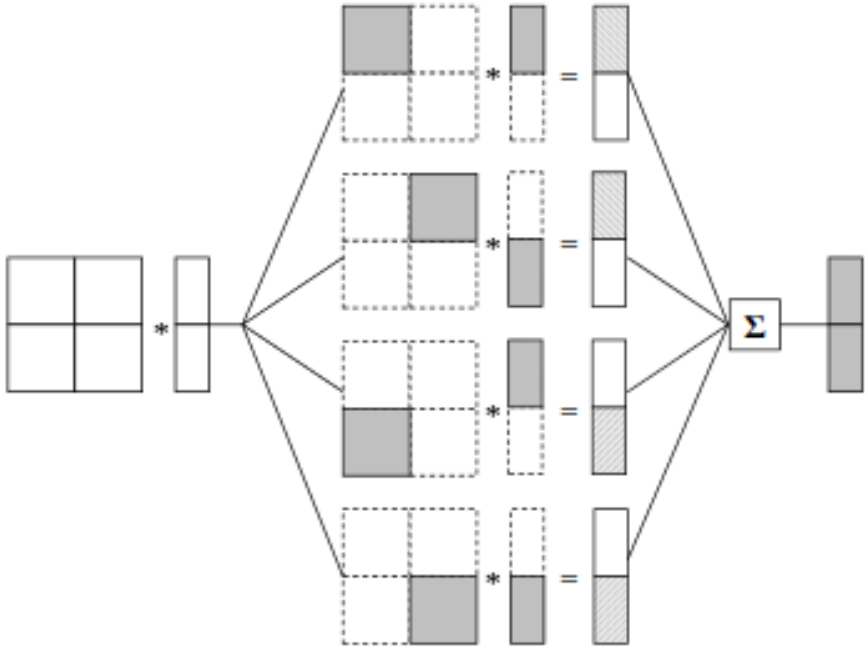

Рисунок 7.3 – Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

В связи с тем, что работа ведется с квадратными матрицами, следует, что число блоков в матрице совпадает по горизонтали и вертикали *(q)*. Тогда количество блоков во всей матрице составит q^2. Для эффективного выполнения алгоритма, основанного на блочном разделении, целесообразно количество потоков выбирать по количеству блоков или, по крайней мере, кратное ему, т.к. число вычислительных элементов может отличаться от числа потоков.

В данном способе умножения матрицы на вектор возникает информационная зависимость на этапе суммирования результатов, которая решается путем дублирования результирующего вектора по всем потокам и суммированием соответствующих элементов промежуточных результатов одним из потоков.

6.6 Анализ эффективности общего алгоритма

Проведем анализ эффективности алгоритма умножения матрицы на вектор при блочном разделении данных. Как уже говорилось, матрица *A* квадратная. Необходимо также отметить, что для получения результирующего вектора нужно сложить вектора частичных результатов, а это требует дополнительных вычислительных операций. Пусть π количество потоков использовалось для решения алгоритма, тогда количество дополнительных операций составит $n*\pi$, а время вычислений:

$$T_{calc} = \left(\frac{n(2n-1)}{p} + n\pi\right) \mathrm{t} \tag{7.17}$$

Оценки времени выполнения алгоритма с учетом времени доступа к памяти вспомним, что при делении объема извлекаемых данных M на пропускную способность β, то получается время доступа к памяти. В этом случае время работы алгоритма можно представить в виде следующей формулы:

$$T_p = \left(\frac{n(2n-1)}{p} + n\pi\right)\mathrm{t} + n^2\alpha + \frac{M}{\beta} \tag{7.18}$$

6.7 Форматы хранения при блочном разделении данных

Нами уже были рассмотрены форматы хранения разреженных матриц. В этом разделе будут представлены форматы хранения именно при реализации алгоритма с блочным разделением данных для разреженных матриц [5].

Наиболее часто используемый формат хранения разреженной матрицы является CRS (Compressed Row Storage). Именно на основе этого формата работают два следующих формата хранения блоков: BCRS (Block Compressed Row Storage) и SBCRS (Sparse Block Compressed Row Storage). На рисунке 7.4 представлен пример матрицы, формат CRS, BCRS и SBCRS.

$$\begin{pmatrix} 6 & 0 & 9 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 \end{pmatrix}$$

а)

$$value = \{6 \quad 9 \quad 4 \quad 4 \quad 5 \quad 3 \quad 5 \quad 8 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 2\}$$
$$col\_ind = \{1 \quad 3 \quad 6 \quad 6 \quad 2 \quad 3 \quad 4 \quad 5 \quad 5 \quad 6 \quad 6 \quad 7 \quad 7 \quad 8\}$$
$$row\_ptr = \{1 \quad 4 \quad 5 \quad 6 \quad 9 \quad 10 \quad 11 \quad 13 \quad 14\}$$

б)

$$value = \{6 \quad 0 \quad 0 \quad 0 \quad 9 \quad 0 \quad 0 \quad 0 \quad 0 \quad 4 \quad 0 \quad 4 \quad 0 \quad 5 \quad 0 \quad 0 \quad 0 \quad 0$$
$$3 \quad 5 \quad 0 \quad 0 \quad 8 \quad 0 \quad 6 \quad 0 \quad 0 \quad 5 \quad 0 \quad 4 \quad 0 \quad 0 \quad 3 \quad 0 \quad 2 \quad 2\}$$
$$bcol\_ind = \{1 \quad 2 \quad 3 \quad 1 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4\}$$
$$brow\_ptr = \{1 \quad 4 \quad 7 \quad 8 \quad 10\}$$

в)

$$block\_array = \begin{Bmatrix} 6 & 9 & 4 & 4 & 5 & 3 & 5 & 8 & 6 & 5 & 4 & 3 & 2 & 2 \\ 1 & 1 & 1 & 2 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 & 2 & 1 & 2 & 1 & 1 & 2 & 2 & 1 & 1 & 2 \end{Bmatrix}$$

$$block\_ptr = \{1 \quad 2 \quad 3 \quad 5 \quad 6 \quad 8 \quad 9 \quad 11 \quad 12\}$$

$$bcol\_ind = \{1 \quad 2 \quad 3 \quad 1 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4\}$$

$$brow\_ptr = \{1 \quad 4 \quad 7 \quad 8 \quad 10\}$$

г)

Рисунок 7.4 – а) пример матрицы; б) CRS формат; в) BCRS формат с блоками 2x2; г) SBCRS формат с блоками 2x2

6.7.1 Block Compressed Row Storage (BCRS)

В формате BCRS матрица делится на подматрицы размера *brxbc* (так называемые блоки), где *br* и *bc* являются фиксированными целыми числами. BCRS, подобно формату CRS, хранит только ненулевые блоки (блоки, в которых, по крайней мере, один ненулевой элемент). Пусть *brows = rows/br* — количество блоков в строке, а *bnze* — количество ненулевых блоков в матрице. BCRS представляется тремя массивами: *value, bcol_ind, brow_ptr*.

Массив *value* типа *double* длины *bnze*×*br*×*bc* хранит элементы ненулевых блоков: первые *br*×*bc* элементов первого ненулевого блока, следующие *br*×*bc* элементов второго блока и т.д. Целочисленный массив *bcol_ind* длины *bnze* хранит индексы столбцов ненулевых блоков. Целочисленный массив *brow_ptr* длины *(brows+1)* хранит указатели на начало строки каждого блока в массиве *bcol_ind*.

6.7.2 Алгоритм умножения разреженной матрицы на вектор в формате BCRS

Данный алгоритм не является достаточно эффективным, так как предполагается хранение нулевых элементов при хранении ненулевых блоков [6].

```
z=0
for b = 0 to brows-1
   y ( b )=0
   for j = brow_ptr ( b ) to ( brow_ptr ( b+1 ) – 1 )
      for k = 0 to br-1
         for t = 0 to bc-1
            y ( bc*b+k ) = y (bc*b+k ) + val (z ) * x
( bc*bcol_ind (j-1 )+t )
            z++
         end for t
      end for k
   end for j
end for b
```

Рисунок 7.5 – Псевдокод алгоритма в формате BCRS

6.7.3 Sparse Block Compressed Row Storage (SBCRS)

Формат строится следующим образом: матрица делится на блоки размера *SxS*, работа осуществляется с помощью четырех массивов, в которых хранятся ненулевые блоки матрицы.

Массив *block_array* размера *3nz*, где *nz* - количество ненулевых элементов матрицы, хранит в себе ненулевые значения, а также индекс строки и столбца, относительно блока, в котором этот элемент находится. Индекс строки и столбца получается путем сдвига на *nz*, относительно текущего элемента.

Массив *block_ptr* размера *bnze* - количество ненулевых блоков содержит в себе информацию о том, сколько элементов в каждом блоке. Каждое значение указывает на индекс начала элементов в массиве *block_array*.

Массивы *bcol_ind* и *brow_ptr* аналогичны BCRS формату, который описан выше.

6.7.4 Алгоритм умножения разреженной матрицы на вектор в формате SBCRS

Алгоритм на основе данного формата хранения является наиболее эффективным, потому что нет хранения нулевых элементов, к любому элементу быстрый доступ, в связи с тем, что в массивах хранится полная информация о местонахождении элемента, как в блоке, так и целиком в матрице [7]. Псевдокод представлен на рисунке 7.6:

```
for b = 0 to brows-1
  y ( b ) = 0
  for j = brow_ptr ( b  ) to ( brow_ptr ( b+1 ) – 1 )
    for z = block_ptr ( j ) to block_ptr ( j+1 ) - 1
      y(S*b+(row_pos(z)-1)) = y(S*b+(row_pos(z)-1))
      +block_array(z)*x(S*(bcol_ind(j)-1) +col_pos(z)-1
    end for z
  end for j
end for b
```

Рисунок 7.6 – Псевдокод алгоритма в формате SBCRS

6.8 Алгоритм умножения при блочном разделении данных

Рассмотрим реализацию алгоритма умножения симметричной разреженной матрицы в формате RR(L)U на вектор. Схема информационных связей представлена на рисунке 7.7.

Нижний треугольник матрицы делится на блоки размера *blocksizexblocksize*, причем блоки, находящиеся у диагональных элементов, могут хранить меньше элементов, чем размер блока. Связано это со «ступенчатым» видом нижнего треугольника.

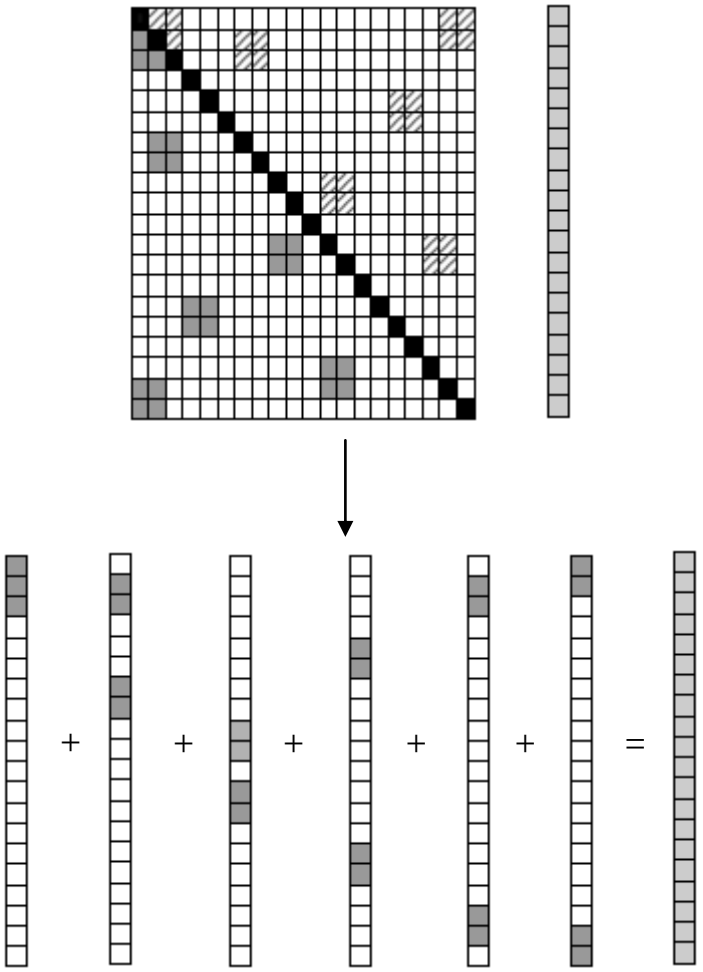

Рисунок 7.7 - Организация вычислений при выполнении алгоритма умножения матрицы на вектор с блочным разделением данных и форматом хранения матрицы RR(L)U

Обработке подвергаются блоки, в которых хотя бы один элемент ненулевой (ненулевые блоки). От каждого ненулевого блока из нижнего треугольника зависит два блока в результирующем векторе в связи с симметричностью портрета матрицы.

$$\begin{pmatrix} a_{ij} & a_{ij+1} \\ a_{i+1j} & a_{i+1j+1} \end{pmatrix} * \begin{pmatrix} a_j \\ a_{j+1} \end{pmatrix} \text{ и } \begin{pmatrix} a_{ij} & a_{ij+1} \\ a_{i+1j} & a_{i+1j+1} \end{pmatrix} * \begin{pmatrix} a_i \\ a_{i+1} \end{pmatrix},$$

причем $a_{ij} = a_{ji}$ для всех i,j.

Соответственно, умножая ненулевой блок на вектор, мы должны сразу организовать умножение симметричного ему блока на вектор. Для получения наиболее эффективного решения количество потоков p должно быть кратно количеству ненулевых блоков nz.

При формировании результирующего вектора возникает зависимость по данным: несколько потоков, обрабатывающие разные блоки, которые находятся в одной полосе, пытаются записать результат умножения в вектор. Для устранения данной зависимости мы продублируем результирующий вектор по количеству

потоков, а после завершения вычислений просуммируем соответствующие элементы частичных результатов для получения решения.

Организация блоков осуществляется следующим образом:

1) Исходная матрица разбивается на блоки размера *blocksizexblocksize.*
2) Заполняется структура данных *struct index_block* для каждого блока, в которой хранится начальное значение строки и столбца (*i_start, j_start*), а также значение *flag*, которое равно 1, если в блоке хотя бы один элемент ненулевой, равно 0, если в блоке все значения равны 0.
3) Создается массив структур по количеству ненулевых блоков *struct CRB*, в каждой из которых хранится количество ненулевых элементов в блоке, а также их значение и местоположение в матрице.
4) Происходит обработка всех ненулевых элементов по всем ненулевым блокам.

Реализация алгоритма умножения симметричной разреженной матрицы на вектор, с разделением данных по блокам на языке Си, описана в приложении В (листинг В.1).

6.8.1 Анализ

Проведем анализ эффективности алгоритма умножения матрицы на вектор при блочном разделении данных. Заметим, что для получения результирующего вектора нужно сложить вектора частичных результатов, а это требует дополнительных вычислительных операций.

$$T_1 = 2 \sum_{i=0}^{col\_block} 2nz_i + n, \tag{7.19}$$

где *col_block* – количество блоков, *nz* – количество ненулевых элементов в блоке.

Оценки времени выполнения алгоритма с учетом времени доступа к памяти вспомним, что при делении объема извлекаемых данных M на пропускную способность β, то получается время доступа к памяти. В этом случае время работы алгоритма можно представить в виде следующей формулы:

$$T_p = 2t \sum_{i=0}^{col\_block/p} 2nz_i + \frac{tn}{p} + nz\alpha + \frac{8nz}{\beta} \tag{7.20}$$

7. ОРГАНИЗАЦИЯ ВЫЧИСЛЕНИЙ В СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

Разработку параллельных программ можно разделить на три этапа:

- Декомпозиция задачи на подзадачи. На этом этапе очень выгодно так разбить задачу, чтобы ее части работали независимо друг от друга (локальность данных).
- Распределение подзадач по процессорам.
- При написании программы использовать средства осуществления параллелизма. Выбор библиотеки может зависеть от платформы, от уровня производительности и от самой задачи.

Все вычислительные системы можно разделить на несколько групп:

- Системы с распределенной памятью: каждый процессор имеет свою память и не может напрямую работать с памятью другого процессора. Библиотека, наиболее часто используемая при создании многопоточных программ в данных системах — MPI (Message Passing Interface).
- Системы с общей (разделяемой) памятью: процессор может без проблем работать с памятью другого процессора. В такой системе процессоры объединены с помощью шины, а сама система называется SMP-системой. Подходы к разработке параллельных программ: POSIX Threads, стандарт OpenMP, механизм передачи сообщений.
- Комбинированные системы или, по-другому, кластеры.

7.1 Симметричные мультипроцессорные системы (SMP)

В состав системы входит несколько однородных процессоров и массив общей памяти. Все процессоры с одинаковой скоростью имеют доступ к памяти. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.

За счет наличия общей памяти взаимодействие процессоров между собой сильно упрощено, хотя накладываются и определенные ограничения на их число - не более 32 в реальных системах.

Как последовательная, так и параллельная программа состоит из последовательных и параллельных областей. Последовательные области выполняются главным потоком (процессом). Этот же инициирует выполнение программы, а также завершает ее. Помимо этого, параллельная область последовательной программы также выполняется одним потоком, и этот поток является единственным на протяжении выполнения всей программы. В областях распараллеливания параллельной программы создается некоторое количество потоков. Порожденные потоки могут выполняться на одном процессоре или на разных. Если потоки (процессы) выполняются на одном, процессоре возникает конкуренция за доступ к данным. Разрешением конфликтов занимается планировщик операционной системы. В операционной системе Linux планировщик задач осуществляет обработку процессов с помощью стандартного

карусельного (round-robin) алгоритма. Следовательно, в параллельных программах при достижении области распараллеливания начинает выполняться ряд параллельных потоков.

В начале выполнения любой программы происходит инициализация и выполнение главного потока (процесса), который в свою очередь порождает параллельные потоки и передает им необходимые данные. Потоки из одной параллельной области могут выполняться как независимо друг от друга, так и передавая и получая сообщения от других потоков. Для обмена данными между параллельными процессами (потоками) в OpenMP используются общие переменные. Обращение к общим переменным в различных потоках обычно приводит к конфликту при доступе к памяти. Чтобы предотвратить конфликты, используют процедуру синхронизации (synchronization). Стоит также заметить, что операция синхронизации является весьма дорогой по временным затратам. Для того, чтобы ее избежать или хотя бы реже применять, необходимо тщательно продумывать структуру данных программы.

Выполнение параллельных потоков всегда начинается с их инициализации. Создаются дескрипторы порождаемых потоков и копируются данные из области главного потока в области созданных параллельных потоков.

Для получения выигрыша в быстродействии параллельной программы, необходимо, чтобы трудоемкость параллельных процессов в областях распараллеливания программы существенно превосходила бы трудоемкость порождения параллельных потоков. Иначе никакого выигрыша по быстродействию получить не удастся, а зачастую можно оказаться даже и в проигрыше.

После того, как параллельные потоки завершили работу, управление передается главному потоку. При этом стоит учесть проблему передачи данных от параллельных главному. Важную роль играет синхронизация завершения работы параллельных потоков, в связи с тем, что время выполнения потоков непредсказуемо. При выполнении синхронизации, потоки, завершившие работы, ожидают завершения других потоков. Естественно, при этом неизбежна потеря эффективности работы параллельной программы.

На основании изложенного выше можно сделать следующий важный вывод: при выделении параллельных областей программы и разработке параллельных процессов необходимо, чтобы трудоемкость параллельных процессов была достаточно большой. В противном случае параллельный вариант программы будет проигрывать в быстродействии последовательной программе.

7.2 Введение в OpenMP

OpenMP (Open Multi-Processing) – открытый стандарт для написания параллельных программ для систем с общей памятью, состоящий из набора директив компилятора и библиотечных функций.

С помощью данного механизма достаточно легко создавать многопоточные приложения на C/C++. Он поддерживается производителями аппаратуры (Intel, HP, SGI, Sun, IBM), разработчиками компиляторов (Intel, Microsoft, KAI, PGI, PSR, APR, Absoft).

7.3 Программная модель OpenMP

В начале программы происходит инициализация главного потока исполнения, затем при необходимости главным потоком порождаются дочерние. Работает на основе модели *fork-join*. Программирование основывается на вставке директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода.

Последовательный код

```
void main(){
  double x[1000];
  for(i=0; i<1000; i++){
    calc_smth(&x[i]);
  }
}
```

Параллельный код

```
void main(){
  double x[1000];
#pragma omp parallel for ...
  for(i=0; i<1000; i++){
    calc_smth(&x[i]);
  }
}
```

Рисунок 7.1 – Пример программ с использованием OpenMP

Директива *#pragma omp parallel for* указывает на то, что данный цикл следует разделить по итерациям между потоками. Количество потоков - величина изменяемая, которая контролируется из программы или переменную окружения OMP_NUM_THREADS.

Еще одним немаловажным пунктом является то, что именно разработчик ответственен за синхронизацию потоков и зависимость между данными.

Как происходит взаимодействие потоков?

В модели с разделяемой памятью взаимодействие потоков происходит через разделяемые переменные. Чтобы не возникало ошибки соревнования *(race condition)* необходима аккуратность обращения с такими переменными. Гонка за данными происходит по причине того, что последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому. Чтобы контролировать возникновение ошибки, необходима синхронизация. Для этого используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки. Стоит отметить, что синхронизация может потребовать от программы дополнительных накладных расходов и лучше подумать и распределить данные таким образом, чтобы количество точек синхронизации было минимизировано.

7.4 Основы OpenMP

7.4.1 Синтаксис

В целом конструкции OpenMP представляют директивы компилятора. Для C/C++ директивы имеют следующий вид:

#pragma omp конструкция [условие [условие] …]

В большинстве своем директивы OpenMP применимы только к структурным блокам, которые имеют единственную точку входа и единственную точку выхода.

7.4.2 Параллельные регионы

Параллельные регионы являются основным понятием в OpenMP. Именно тот блок, где задан параллельный регион будет выполняться параллельно. Как только компилятор встречает *pragma omp parallel*, он вставляет инструкции для создания параллельных потоков.

Каждый порожденный поток, независимо от других потоков, выполняет блок кода. По умолчанию синхронизация между потоками отсутствует, поэтому последовательность выполнения конкретного оператора различными потоками не определена. После выполнения параллельного участка кода, все потоки, кроме главного завершаются.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков – 1. Идентификатор потока может быть определен с помощью функции omp_get_thread_num().

7.4.3 Модель исполнения

Существует две модели исполнения: динамическая, когда количество используемых потоков в программе может варьироваться от одной области параллельного выполнения к другой, и статическая, когда количество потоков фиксировано.

7.4.4 Конструкции OpenMP

7.4.4.1 Условия выполнения

Условиями выполнения определяется то, каким образом будет выполняться параллельный участок кода и область видимости переменных внутри этого участка кода. Опишем следующие условия:

1) shared(var1, var2, ...)

Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками. Всем потокам будет доступна одна и та же область памяти.

2) private(var1, var2, ...)

Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

3) firstprivate(var1, var2, ...)

Это условие аналогично условию private за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменная до входа в параллельную секцию.

4) lastprivate(var1, var2, ...)

Приватные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода.

5) reduction(оператор:var1, var2, ...)

Это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

6) if(выражение)

Это условие говорит о том, что параллельное выполнение необходимо только если выражение истинно. Это условие определяет область видимости переменных внутри параллельного участка кода по умолчанию.

7) schedule(type[,chank])

Этим условием контролируется то, как итерации цикла распределяются между потоками.

7.4.4.2 Параллельный цикл for

Барьерной синхронизацией для потоков является конец цикла. Все потоки, которые закончили выполнение, ожидают завершения оставшихся потоков. После чего только основная нить продолжает выполняться. Используя условие nowait для цикла можно разрешить основной нити не дожидаться завершения дочерних нитей.

7.4.5 Переменные окружения OpenMP

1) OMP_NUM_THREADS

Устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров.

2) OMP_SCHEDULE

Устанавливает тип распределения работ в параллельных циклах с типом runtime.

3) OMP_DYNAMIC

Разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.

4) OMP_NESTED

Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.

7.4.6 Библиотечные функции OpenMP

Для более эффективной работы параллельной программы при использовании OpenMP пользователю предоставляется возможность использования библиотечных функций. Библиотека OpenMP предоставляет пользователю следующий набор функций:

1) void omp_set_num_threads(int num_threads)

Устанавливает количество потоков, которое может быть запрошено для параллельного блока.

2) int omp_get_num_threads()

Возвращает количество потоков в текущей команде параллельных потоков.

3) int omp_get_max_threads()

Возвращает максимальное количество потоков, которое может быть установлено.

4) int omp_get_thread_num()

Возвращает номер потока в команде (целое число от 0 до количества потоков – 1).

5) int omp_get_num_procs()

Возвращает количество физических процессоров доступных программе.

6) int omp_in_parallel()

Возвращает не нулевое значение, если вызвана внутри параллельного блока. В противном случае возвращается 0.

7) void omp_set_dyamic(expr)

Разрешает/запрещает динамическое выделение потоков.

8) int omp_get_dynamic()

Возвращает разрешено или запрещено динамическое выделение потоков.

9) void omp_set_nested(expr)

Разрешает/запрещает вложенный параллелизм.

10) int omp_get_nested()

Возвращает разрешен или запрещен вложенный параллелизм.

7.4.7 Зависимость по данным

Для предотвращения появления самого главного врага параллельных программ — зависимости по данным, каждая итерация цикла не должна зависеть от результата предыдущей. Независимость — синоним успешной работы.

Утверждение 1

Только те переменные, в которые происходит запись на одной итерации и чтение их значения на другой создают зависимость по данным.

Утверждение 2

Только разделяемые переменные могут создавать зависимость по данным.

Следствие. Если переменная не объявлена как приватная, она может оказаться разделяемой и привести к зависимости по данным.

7.4.8 Средства синхронизации в OpenMP

В OpenMP предусмотрены следующие конструкции синхронизации:

1) critical – критическая секция. Представляет фрагмент кода, выполнение которого осуществляется только одним потоком в каждый текущий момент времени.

2) atomic – атомарность операции. Определяет переменную, доступ к которой является неделимой операцией.

3) barrier – точка синхронизации. Представляет собой определенной событие, которое должны достигнуть все потоки для дальнейшего продолжения вычислений.

4) master – блок, который будет выполнен только основным потоком. Все остальные потоки пропустят этот блок. В конце блока неявной синхронизации нет.

5) ordered – выполнять блок в заданной последовательности. Данная директива предусматривает выполнение параллельного фрагмента в определенном порядке.

6) flush – немедленный сброс значений разделяемых переменных в память. Если какой-либо поток модифицировал значение какого-либо элемента, то это значение непременно должно быть записано в общую память.

7.5 Реализация алгоритма с использованием OpenMP

7.5.1 Строчное разделение данных

Листинг 8.1 – Реализация алгоритма с использованием средств разработки OpenMP

```
void multy_by_rows(double * adiag, double * altr, int *
jptr, int * iptr, int Size, int size_altr, double *
pVector, double * Result, int GridThreadsNum)
{

omp_set_num_threads(GridThreadsNum);
    double t = wtime();
    #pragma omp parallel for private(j,k,p)
    for(i = 0; i < Size; i++)
    {
     for(j = iptr[i]; j < iptr[i + 1]; j++)
     {
         Result[i] += pVector[jptr[j]] * altr[j];
     }
     for(k = i; k < Size; k++)
     {
         for(p = iptr[k]; p < iptr[k + 1]; p++)
         {
              if(jptr[p] == i)
              {
               Result[jptr[p]] += pVector[k] * altr[p];
               //printf("k %d i %d ThreadID %d\n",k, i,
omp_get_thread_num());
              }
         }
     }
    }
    t = wtime() - t;
    printf("elapsed time(serial) CЃC,CЂPsPєPё: %.16f sec
\n", t);

}
```

7.5.2 Блочное разделение данных

Листинг 8.2 – Реализация алгоритма с использованием средств разработки OpenMP

```
void multy_by_block(int Size, double * adiag, double *
altr, int * jptr, int * iptr, double *pVector, double *
Result, int GridThreadsNum)
{
```

```
          …
omp_set_num_threads(GridThreadsNum);
     double t = wtime();
     #pragma omp parallel for private(k)
        for(p = 0; p < nnz_block; p++)
     {
         for(k = 0; k < CRB_block[p].nz; k++)
         {
          CRB_block[p].result[CRB_block[p].row[k]] +=
pVector[CRB_block[p].col[k]] * CRB_block[p].value[k];
          CRB_block[p].result[CRB_block[p].col[k]] +=
pVector[CRB_block[p].row[k]] * CRB_block[p].value[k];

         }
     }
     t = wtime() - t;
     printf("elapsed time(serial) block: %.16f sec \n", t);


    for(p = 0; p < nnz_block; p++)
     for(i = 0; i < Size; i++)
         Result[i] += CRB_block[p].result[i];


          …
}
```

8. ЧИСЛЕННЫЕ ЭКСПЕРИМЕНТЫ

8.1 Чтение и конвертация матрицы

Прежде, чем приступить к решению поставленной задачи, а именно умножение разреженной матрицы на вектор, необходима инициализация самой матрицы.

Работа происходит с матрицами из Florida Collection Sparse Matrix.

Матрица хранится в файле в координатном формате. При считывании элементов происходит преобразование в формат хранения, с которым будет продолжена дальнейшая работа. Данный этап называется конвертирование, а функция преобразования - конвертер.

В нашем случае работа предполагается в формате RR(L)U, который достаточно подробно был описан выше.

Рассмотрим схему работы конвертера:

1) В функции INITIALIZATION происходит считывание матрицы из файла, хранящейся в координатном формате.

2) На вход функции CONVERTER поступают три массива: value, row, col, в которых содержатся значения ненулевых элементов, индексы строк и столбцов. Далее происходит преобразование в формат RR(L)U, т.е. формируется четыре массива, первый из которых хранит диагональные элементы, второй и третий ненулевые элементы нижнего треугольника и индексы их столбцов, а четвертый массив количество элементов в каждой строке нижнего треугольника.

3) Сформированные массивы участвуют в дальнейшей работе программы.

Листинг 8.1 – Реализация функций INITIALIZATION и CONVERTER на языке Си

```
void initialization(FILE *fp, int sz1, double VAL[sz1], int
I[sz1], int J[sz1])
{
    int i, k = 0;

    for(i = 0; !feof(fp); i++) {
        fscanf(fp, "%d %d %lf\n", &I[i], &J[i], &VAL[i]);
        k++;
    }
    printf("%d == %d\n", sz1, k);
}

void converter(double * adiag, int *jptr, double * altr,
int * iptr, double * elem, int * elem_i, int * elem_j, int
size,int size_altr, int size_diag)
{
    int k, m;
    for(k = 0; k < size_diag + 1; k++)
     iptr[k] = 0;
```

```
for(k = 0; k < size; k++)
{
      if(elem_i[k] > elem_j[k])
      {
         for(m = elem_i[k] + 1; m < size_diag + 1; m++)
                iptr[m]++;
      }
}
m = 0;
for(k = 0; k < size; k++)
{
    if((elem_i[k] == elem_j[k]) &&
           (elem_i[k] < size_diag))
        adiag[elem_i[k]] = elem[k];
    if((elem_i[k] > elem_j[k]) && (m < size_altr))
    {
        altr[m] = elem[k];
        jptr[m] = elem_j[k];
        m++;
    }
}
```

8.2 Валидация полученных результатов

Валидация предполагает проверку решения на истинность. При работе с числами в формате double, может возникать определенная погрешность. После решения последовательного и параллельного алгоритма, мы имеем два вектора решений, значения которых необходимо сравнить. В связи с возникновением погрешности и большим размером вектором, вручную осуществить проверку невозможно. Поэтому было принято решение считать норму разности векторов. При этом если норма меньше числа EPS (в нашем случае 10^{-9}), значит получено верное решение.

За норму берем максимальный элемент в массиве. Реализация функции получения нормы представлена на листинге 8.2.

Листинг 8.2 – Реализация функции NORM()

```
int Norm(double *A, double *B, int size)
{
    int i = 0;

    double max = -10000;
    for(i = 0; i < size; i++)
        if (abs(A[i]-B[i]) > max)
            max = abs(A[i]-B[i]);
    if(max > 0.000000001)
        return 0;
    else
        return 1;
}
```

8.3 Тесты на производительность алгоритма

8.3.1 Ускорение

Ускорением параллельного алгоритма называется отношение последовательной версии к параллельной.

$$S = \frac{T_1}{T_p} \tag{8.1}$$

Результаты работы параллельных алгоритмов представлены в таблицах 8.1 и 8.2, где первый столбец названия матриц из Florida Sparse Matrix Collection, *nxn* – размер матрицы, nz – количество ненулевых элементов, p – количество потоков.

Таблица 8.1 – Ускорение параллельного алгоритма при делении по блокам

			Sp block			
	n	nz	p = 1	p = 2	p = 4	p = 8
Qc2534	2534	463360	1	1,93	3,95	8,18
Sherman3	5005	20033	1	1,98	3,99	9,43
Fidamp15	9287	98519	1	1,9	3,42	7,55
Bcsstk17	10974	219812	1	1,96	3,85	7,76

Таблица 8.2 – Ускорение параллельного алгоритма при делении по строкам

			Sp row			
	n	nz	p = 1	p=2	p=4	p=8
Qc2534	2534	463360	1	1,12	2,05	3,74
Sherman3	5005	20033	1	1,18	2,14	4,38
Fidamp15	9287	98519	1	1,09	2,29	3,11
Bcsstk17	10974	219812	1	1,11	2,81	2,98

Приведены графики зависимости ускорения параллельного алгоритма от количества потоков (рисунок 8.1, рисунок 8.2).

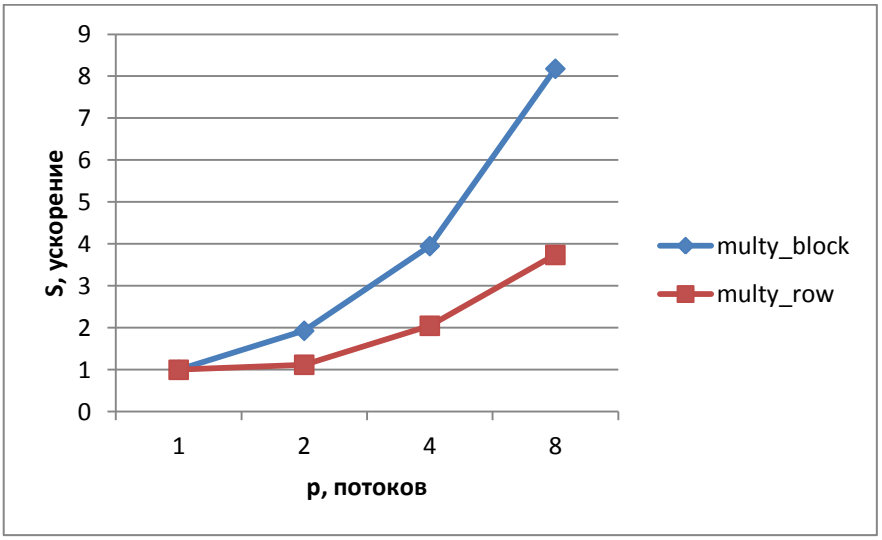

Рисунок 8.1 – Графики зависимости ускорения алгоритма от количество потоков для матрицы Qc2534.

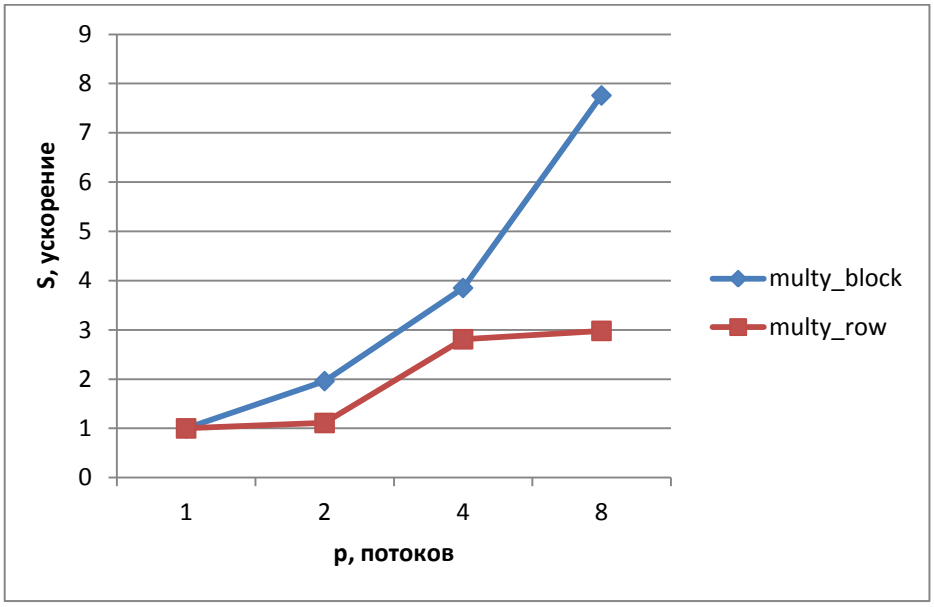

Рисунок 8.2 – Графики зависимости ускорения параллельной программы от числа потоков для матрицы Bcsstk17.

8.3.2 Эффективность

Для оценки масштабируемости параллельного алгоритма используется понятие эффективности. Под эффективностью параллельного алгоритма понимается отношение:

$$E_p = \frac{S_p}{p} \tag{8.2}$$

Поскольку $S_p \le p$, то $E_p \le 1$. Если алгоритм достигает максимального ускорения ($S_p = p$), то $E_p = 1$.

Результаты расчетов представлены в таблице 8.3. Рассчитана эффективность последовательных алгоритмов для четырех матриц из Florida Sparse Matrix Collection.

Таблица 8.3 – Данные об эффективности параллельных алгоритмов

			Ep block				Ep row			
	n	nz	1	2	4	8	1	2	4	8
Qc2534	2534	463360	1	0,965	0,9875	1,0225	1	0,56	0,5125	0,4675
Sherman3	5005	20033	1	0,99	0,9975	1,17875	1	0,59	0,535	0,5475
Fidamp15	9287	98519	1	0,95	0,855	0,94375	1	0,545	0,5725	0,38875
Bcsstk17	10974	219812	1	0,98	0,9625	0,97	1	0,555	0,7025	0,3725

Представлены графики эффективности работы параллельных алгоритмов (рисунок 8.3, рисунок 8.4).

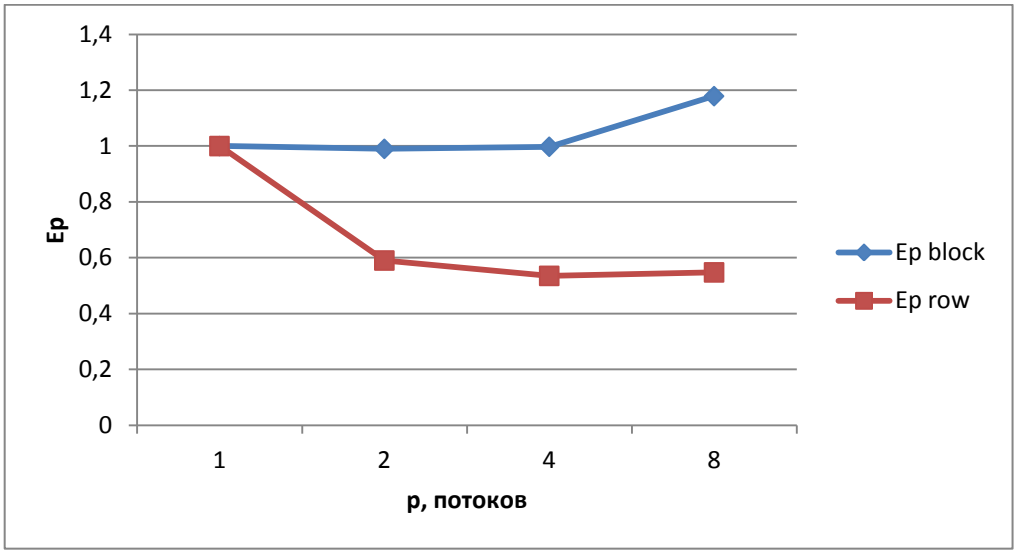

Рисунок 8.3 – Графики зависимости эффективности алгоритмов от числа потоков для матрицы Sherman3

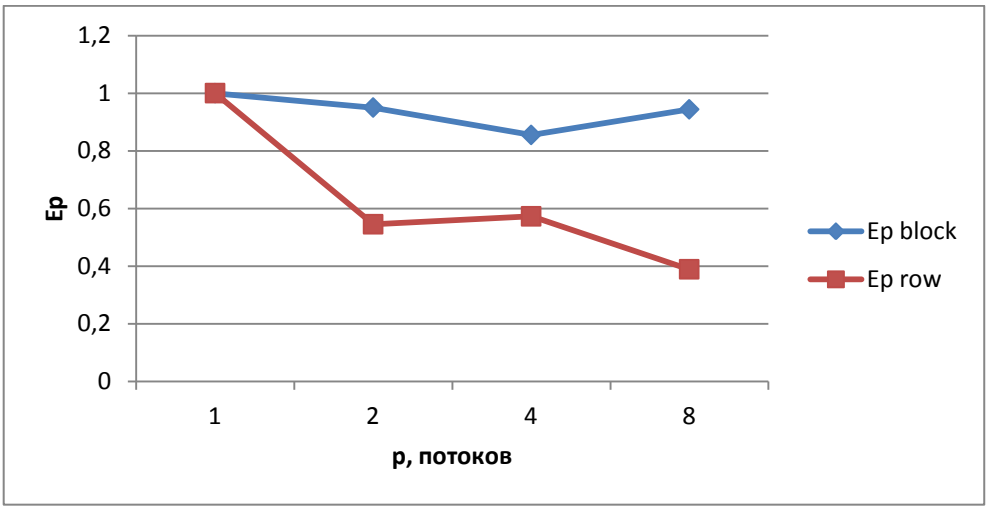

Рисунок 8.4 – Графики зависимости эффективности алгоритмов от числа потоков для матрицы Fidamp15

8.3.3 Время работы алгоритма

Таблица 8.4 – Время работы последовательного и параллельных алгоритмов

	tp block	tp row	t1
Qc2534	14,3597	16,8597	19,0081
Sherman3	15,6679	17,6179	21,0001
Fidamp15	21,3312	22,0912	24,6301
Bcsstk17	29,0013	30,7413	36,1987

На рисунке 8.5 представлены графики времени выполнения алгоритмов в зависимости от размера матрицы и количества ее ненулевых элементов.

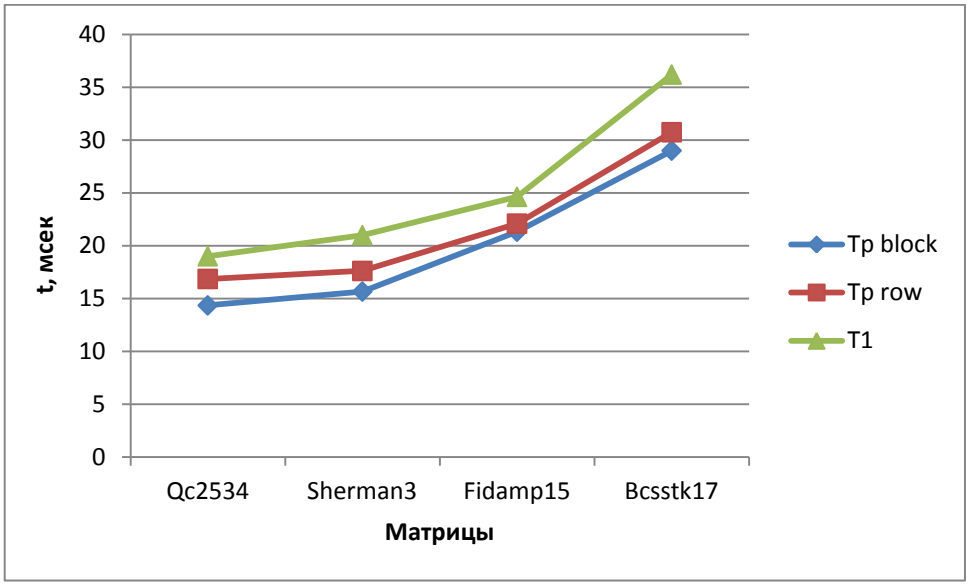

Рисунок 8.5 – Графики зависимости времени выполнения алгоритмов от размера данных

Таблица 8.5 – Данные времени выполнения параллельных алгоритмов в зависимости от количества потоков

	block				row			
	tp1	tp2	tp4	tp8	tp1	tp2	tp4	tp8
Qc2534	14,3597	7,440259	3,635367	1,755465	16,8597	15,0533	8,224244	4,507941
Sherman3	15,6679	7,913081	3,926792	1,661495	17,6179	14,93042	8,232664	4,022352
Fidamp15	21,3312	11,22695	6,237193	2,825325	22,0912	20,26716	9,646812	7,10328
Bcsstk17	29,0013	14,79658	7,532805	3,737281	30,7413	27,69486	10,93996	10,31587

На рисунке 8.6 и рисунке 8.7 представлены графики, основанные на результатах таблицы 8.5.

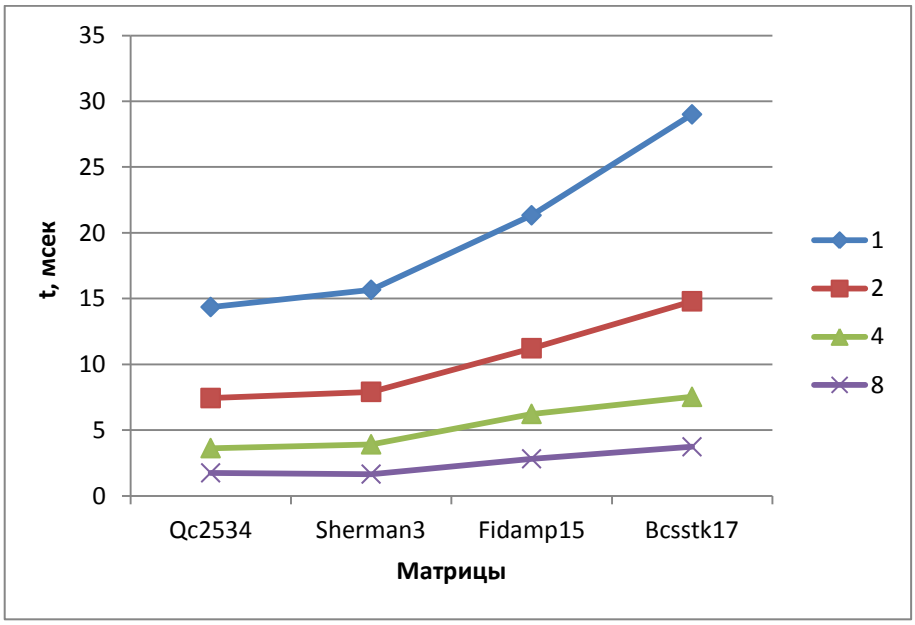

Рисунок 8.6 – Графики зависимости времени выполнения параллельного алгоритма при разделении на блоки от размера матрицы

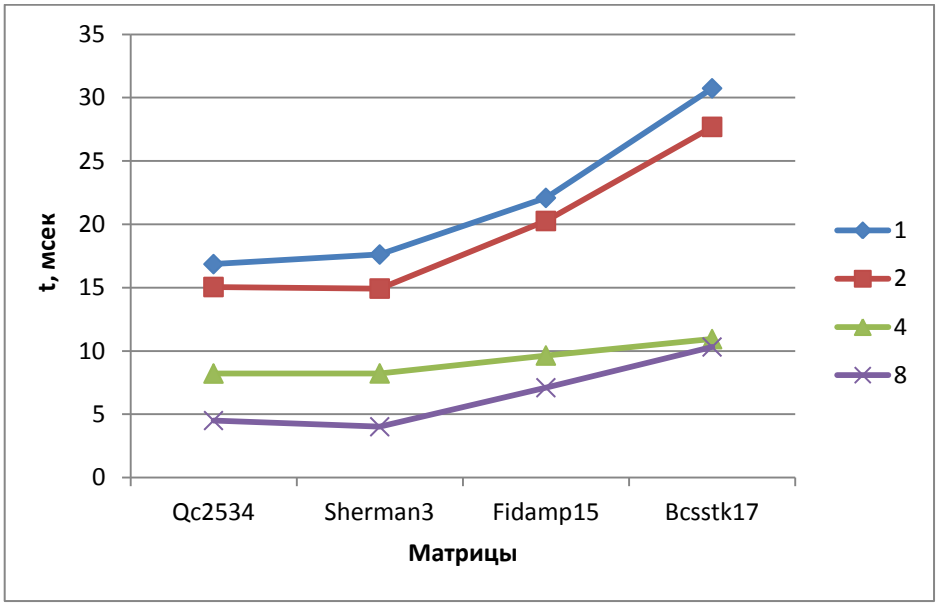

Рисунок 8.7 – Графики зависимости времени выполнения параллельного алгоритма при разделении по строкам от размера матрицы

8.4 Анализ результатов

Эффективность строчного алгоритма уменьшается при увеличении числа потоков за счет дополнительно используемой памяти. Эффективность блочного алгоритма близка к линейной, поскольку не используется дополнительной памяти в процессе распараллеливания. Масштабируемость увеличивается за счет максимальной загрузки потоков.

В процессе анализа видно явное преимущество блочного алгоритма как над последовательным, так и над строчным. Но для достижения таких высоких показателей необходимо учитывать структуру матрицы: размер и количество ненулевых элементов, т.к. самым сложным в алгоритме становится оптимальное разбиение на блоки.

9. ЗАКЛЮЧЕНИЕ

В ходе проделанной работы был создан проект умножения симметричной разреженной матрицы на заполненный вектор-столбец.

Произведено исследование не только достоинств форматов хранения, но и алгоритмов их оптимизации, а также алгоритмов параллельного умножения симметричной разреженной матрицы на вектор. Реализован алгоритм для систем с общей памятью.

Проведен анализ форматов, наиболее подходящих при решении поставленной задачи. Разобраны алгоритмы распараллеливания с учетом особенностей форматов. Разработан программный комплекс для проведения численных экспериментов, включающий в себя: чтение и конвертация матриц, умножение матрицы на вектор разными алгоритмами, валидация полученных результатов и замер времени выполнения.

Результаты работы алгоритмов сильно зависят от структуры матриц. Крайне трудно теоретически оценить эффективность таких алгоритмов. Поэтому был проведен ряд экспериментов, показавший, что блочный формат хранения является более предпочтительным за счет лучшей масштабируемости и эффективности. Данные результаты напрямую зависят от размера матрицы, количества ее ненулевых элементов, а также способа разбиения на блоки, что является еще одной достаточно сложной задачей.

ПРИЛОЖЕНИЕ А

(справочное)
Библиография

1 Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984.
2 Писсанецки С. Технология разреженных матриц. — М.: Мир,1988.
3 Д.Т. Письменный Конспект лекций по высшей математике: в 2 ч. [Текст] : курс лекций — М.: Айрис-пресс, 2006. — 608 с.
4 Тьюарсон Р. Разреженные матрицы. – М.: Мир, 1977.
5 Fast sparse matrix-vector multiplication by exploiting variable block structure URL: http://bebop.cs.berkeley.edu/pubs/vuduc2005-ubcsr-split.pdf (Дата обращения 20.05.2016)
6 Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks
URL: http://www.fftw.org/~athena/papers/csb.pdf (Дата обращения 21.05.2016)
7 Blocked-Based Sparse Matrix-Vector Multiplication on Distributed Memory Parallel Computers
URL: http://ccis2k.org/iajit/PDF/vol.8,no.2/3-766.pdf (Дата обращения 1.06.2016)
8 Гантмахер. Ф. Р. Теория матриц [Электронный ресурс]: учебное пособие – М.: ФИЗМАТЛИТ, 2010. – 560 с.

ПРИЛОЖЕНИЕ Б

(рекомендуемое)
Наиболее употребляемые текстовые сокращения

SMP - ссимметричные
мультипроцессорные системы
OpenMP - Open Multi-Processing
COO – Coordinate Storage
CRS - Compressed Row Storage
CCS - Compressed Column Storage
RR(C)O - Row-wise Representation
Complete and Ordered
CR(C)O - Column-wise Representation
Complete and Ordered
RR(L)U - Row-wise Representation
Lower Unordered Storage
BCRS – Block Compressed Row
Storage
SBCRS – Sparse Block Compressed
Row Storage

ПРИЛОЖЕНИЕ В

Листинг В.1 – Spmv.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <sys/time.h>
#include <omp.h>

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
}

struct index_block {
    int i_start;
    int j_start;
    int flag;
};

struct CRB {
    int nz;
    double *value;
    int *row;
    int *col;
    double *result;
};

struct result{
    double *pResult;
};

int Norm(double *A, double *B, int size)
{
    int i = 0;

    double max = -10000;
    for(i = 0; i < size; i++)
        if (abs(A[i]-B[i]) > max)
            max = abs(A[i]-B[i]);
    if(max > 0.000000001)
        return 0;
    else
        return 1;
}

void readfromfile_vec(FILE *fp, int sz1, double VAL[sz1], int
    I[sz1], int J[sz1])
{
    int i, k = 0;
```

```
    for(i = 0; !feof(fp); i++) {
        fscanf(fp, "%d %d %lf\n", &I[i], &J[i], &VAL[i]);
        k++;
    }
    printf("%d == %d\n", sz1, k);
}

void readfromfile_vec_double(FILE * fp, double * p, int sz1)
{
    int i;

    for(i = 0; i < sz1; i++)
        fscanf(fp, "%lf", &p[i]);
}

void vector_content_int(int * iptr, int * elem_i, int * elem_j, int
    size, int _diag)
{
    int k, m;
    for(k = 0; k < _diag + 1; k++)
    iptr[k] = 0;

    for(k = 0; k < size; k++)
    {
          if(elem_i[k] > elem_j[k])
          {
               for(m = elem_i[k] + 1; m < _diag + 1; m++)
                    iptr[m]++;
          }
    }

}

void vector_content(double * adiag, int *jptr, double * altr, double
    * elem, int * elem_i, int * elem_j, int _altr, int _diag, int
    size)
{
    int k , m;

    m = 0;
    for(k = 0; k < size; k++)
    {
          if((elem_i[k] == elem_j[k]) && (elem_i[k] < _diag))
               adiag[elem_i[k]] = elem[k];
          if((elem_i[k] > elem_j[k]) && (m < _altr))
          {
               altr[m] = elem[k];
               jptr[m] = elem_j[k];
               m++;
          }
    }

}
```

```
void multy_rr(double * adiag, double * altr, int * jptr, int * iptr,
    int _diag, double * x, double * z)
{
    int i, j;
    for(i = 0; i < _diag; i++)
    z[i] = 0;

    for(i = 0; i < _diag; i++)
    {
          z[i] = x[i] * adiag[i];
    }
    for(i = 0; i < _diag; i++)
    {
          for(j = iptr[i]; j < iptr[i + 1]; j++)
          {
               z[i] = z[i] + x[jptr[j]] * altr[j];
               z[jptr[j]] = z[jptr[j]] + x[i] * altr[j];
          }
    }
}

void multy_coo(double * elem, int * elem_i, int * elem_j, int _diag,
    int size, double * x, double * z)
{
    int k;

    for(k = 0; k < _diag; k++)
          z[k] = 0;

    for(k = 0; k < size; k++)
    {
          z[elem_i[k]] += elem[k] * x[elem_j[k]];
          if(elem_i[k] != elem_j[k])
          {
               z[elem_j[k]] += elem[k] * x[elem_i[k]];
          }

    }
}

void multy_by_block(int Size, double * adiag, double * altr, int *
    jptr, int * iptr, double *pVector, double * Result, int
    GridThreadsNum, int NumBlock)
{

    int BlockSize = 2;

    int ColBlocks = (Size * Size) / (BlockSize * BlockSize);
    int ColBlockPtr = sqrt(ColBlocks);

    struct index_block iBlock[ColBlocks];

    int nnz_block = 0;
    int i, j;
```

```
for(NumBlock = 0; NumBlock < ColBlocks; NumBlock++)
{
      iBlock[NumBlock].i_start = (NumBlock / ColBlockPtr) *
BlockSize;
      iBlock[NumBlock].j_start = (NumBlock % ColBlockPtr) *
BlockSize;
      iBlock[NumBlock].flag = 0;
}

for(i = 0; i < Size; i++)
{
      for(NumBlock = 0; NumBlock < ColBlocks; NumBlock++)
      {
           for(j = iptr[i]; j < iptr[i+1]; j++)
           {
                if((iBlock[NumBlock].i_start <= i) && (i <
           iBlock[NumBlock].i_start + BlockSize))
                {
                     if((iBlock[NumBlock].j_start <= jptr[j]) &&
                (jptr[j] < iBlock[NumBlock].j_start + BlockSize))
                          if(altr[iptr[i]] != 0)
                               iBlock[NumBlock].flag = 1;
                }
           }
      }
}

for(NumBlock = 0; NumBlock < ColBlocks; NumBlock++)
      if(iBlock[NumBlock].flag != 0)
           nnz_block++;

int bsrVal[nnz_block];
for(i = 0; i < nnz_block;)
{
      for(NumBlock = 0; NumBlock < ColBlocks; NumBlock++)
      {
           if(iBlock[NumBlock].flag != 0)
           {
                bsrVal[i] = NumBlock;
                i++;
           }
      }
}

struct CRB CRB_block[nnz_block];

for(i = 0; i < nnz_block; i++)
CRB_block[i].nz = 0;

int p, k;
for(p = 0; p < nnz_block; p++)
{
      for(i = 0; i < Size; i++)
```

```
      {
           for(j = iptr[i]; j < iptr[i + 1]; j++)
           {
                if((iBlock[bsrVal[p]].i_start <= i) && (i <
           iBlock[bsrVal[p]].i_start + BlockSize))
                {
                     if((iBlock[bsrVal[p]].j_start <= jptr[j]) &&
                (jptr[j] < iBlock[bsrVal[p]].j_start +
                BlockSize))
                     {
                          CRB_block[p].nz++;
                     }
                }
           }
      }
}

for(i = 0; i < nnz_block; i++)
{
      CRB_block[i].value = (double *) malloc(CRB_block[i].nz *
sizeof(double));
      CRB_block[i].row = (int *) malloc(CRB_block[i].nz *
sizeof(int));
      CRB_block[i].col = (int *) malloc(CRB_block[i].nz *
sizeof(int));
      CRB_block[i].result = (double *) malloc(Size *
sizeof(double));
}

for(i = 0; i < nnz_block; i++)
for(j = 0; j < CRB_block[i].nz; j++)
{
    CRB_block[i].value[j] = 0;
    CRB_block[i].row[j] = 0;
    CRB_block[i].col[j] = 0;
}


for(p = 0; p < nnz_block; p++)
{
      for(k = 0; k < CRB_block[p].nz;)
      {
           for(i = 0; i < Size; i++)
           {
                for(j = iptr[i]; j < iptr[i + 1]; j++)
                {

                     if((iBlock[bsrVal[p]].i_start <= i) && (i <
                iBlock[bsrVal[p]].i_start + BlockSize))
                     {
                          if((iBlock[bsrVal[p]].j_start <=
                     jptr[j]) && (jptr[j] <
                     iBlock[bsrVal[p]].j_start + BlockSize))
                          {
```

```
                                    CRB_block[p].value[k] = altr[j];
                                    CRB_block[p].col[k] = jptr[j];
                                    CRB_block[p].row[k] = i;
                                    k++;
                                }
                            }
                        }
                    }
                }
            }

    for(p = 0; p < nnz_block; p++)
    for(i = 0; i < Size; i++)
    {
        CRB_block[p].result[i] = 0;
    }

    omp_set_num_threads(GridThreadsNum);
    double t = wtime();
    #pragma omp parallel for private(k)
    for(p = 0; p < nnz_block; p++)
    {
        for(k = 0; k < CRB_block[p].nz; k++)
        {
                CRB_block[p].result[CRB_block[p].row[k]] +=
          pVector[CRB_block[p].col[k]] * CRB_block[p].value[k];
                CRB_block[p].result[CRB_block[p].col[k]] +=
          pVector[CRB_block[p].row[k]] * CRB_block[p].value[k];
        }
    }
    t = wtime() - t;
    printf("elapsed time(serial) block: %.16f sec \n", t);

    for(p = 0; p < nnz_block; p++)
          for(i = 0; i < Size; i++)
                Result[i] += CRB_block[p].result[i];

    for(i = 0; i < Size; i++)
          Result[i] += pVector[i] * adiag[i];

}

void func2(double * adiag, double * altr, int * jptr, int * iptr,
    int Size, int size_altr, double * pVector, double * Result, int
    GridThreadsNum)
{
    int i, j, m, p, k;

    for(i = 0; i < Size; i++)
    Result[i] = pVector[i] * adiag[i];
    omp_set_num_threads(GridThreadsNum);
    double t = wtime();
    #pragma omp parallel for private(j,k,p)
    for(i = 0; i < Size; i++)
```

```
    {
          for(j = iptr[i]; j < iptr[i + 1]; j++)
          {
               Result[i] += pVector[jptr[j]] * altr[j];
          }
          for(k = i; k < Size; k++)
          {
               for(p = iptr[k]; p < iptr[k + 1]; p++)
               {
                    if(jptr[p] == i)
                    {
                         Result[jptr[p]] += pVector[k] * altr[p];

                    }
               }
          }
    }
    t = wtime() - t;
    printf("elapsed time(serial) CЃC,CЂPsPєPё: %.16f sec \n", t);

}
```