Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лабораторная работа №7
Разреженное матричное умножение

Мееров И.Б., Сысоев А.В.
при участии Сафоновой Я.

При поддержке компании Intel

Нижний Новгород
2011

Содержание

ВВЕДЕНИЕ **4**

1. МЕТОДИЧЕСКИЕ УКАЗАНИЯ **5**

1.1. Цели и задачи работы 5
1.2. Структура работы 6
1.3. Тестовая инфраструктура 6
1.4. Рекомендации по проведению занятий 7

2. ПОСТАНОВКА ЗАДАЧИ МАТРИЧНОГО УМНОЖЕНИЯ **7**

3. ФОРМАТЫ ХРАНЕНИЯ РАЗРЕЖЕННЫХ МАТРИЦ **8**

3.1. Координатный формат хранения 8
3.2. Формат хранения CRS (CSR) и его модификации 9

4. ГЕНЕРАЦИЯ ТЕСТОВЫХ ЗАДАЧ **12**

5. ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ **17**

6. НАИВНАЯ ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ **24**

6.1. Общая схема решения задачи 24
6.2. Алгоритм транспонирования 26
6.3. Создание проекта 30
6.4. Алгоритм умножения 33

7. ОПТИМИЗИРОВАННАЯ ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ **36**

7.1. Алгоритмическая оптимизация. Подход 1 36
7.2. Алгоритмическая оптимизация. Подход 2 42
7.3. Программная оптимизация. Последние штрихи 50

8. ДВУХФАЗНАЯ ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ **55**

8.1. Идея разделения алгоритмов разреженной алгебры на две фазы 55
8.2. Последовательный двухфазный алгоритм 56

9. АЛГОРИТМ ГУСТАВСОНА **64**

10. ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ **65**

10.1. Параллельная реализация с использованием OpenMP 65
10.2. Балансировка нагрузки 74
10.3. Параллельная реализация с использованием TBB 74

11. ДОПОЛНИТЕЛЬНЫЕ ЗАДАНИЯ **80**

12. ЛИТЕРАТУРА 81

12.1. Использованные источники информации 81
12.2. Дополнительная литература 81
12.3. Ресурсы сети Интернет 82

Введение

Не все так просто, как кажется.

Народная мудрость

Алгебра разреженных матриц (Sparse algebra) – важный раздел математики, имеющий очевидное практическое применение. Разреженные матрицы [1, 2, 4] возникают естественным образом при постановке и решении задач из различных научных и инженерных областей. В частности, при формулировании оптимизационных задач большой размерности с линейными ограничениями матрица системы ограничений нередко оказывается разреженной. Матрицы с существенно преобладающим количеством нулевых элементов формируются при численном решении дифференциальных уравнений в частных производных. Такие матрицы возникают в теории графов. Есть и другие случаи использования (см., например, обзор в работе [4]).

Попробуем определиться с понятием *разреженная матрица*. Выясняется, что это не так просто. Один из первых источников по данной тематике, в комплексе рассматривающий проблемы работы с разреженными матрицами, дает такое определение: *разреженной называют матрицу, имеющую малый процент ненулевых элементов* [4]. В ряде других источников встречается такая формулировка: матрица размера N x N называется разреженной, если количество ее ненулевых элементов есть O(N). Известны и другие определения. Понятно, что приведенные варианты являются не вполне точными в математическом смысле. Причина затруднений с более точной формулировкой состоит в том, что на практике классификация матрицы зависит не только от количества ненулевых элементов, но и от размера матрицы, распределения ненулевых элементов, архитектуры конкретной вычислительной системы и используемых алгоритмов (см., например, более подробное обоснование в [2]). Почему так? Дело в том, что важно не просто дать матрице звучное название: разреженная или плотная. Важно определиться с выбором структуры хранения – использовать обычные двухмерные массивы или что-то более сложное, учитывающее факт разреженности. Именно в момент выбора структуры хранения вступают в дело перечисленные выше факторы. Так, в ряде случаев матрица имеет регулярную структуру (например, ленточная матрица), что позволяет разработать специализированную структуру хранения. Иногда размерность матрицы такова, что ее плотное представление попросту не убирается в память. Если и разреженное, и плотное представление допустимы,

а количество ненулевых элементов невелико, необходимо проанализировать, какая структура хранения будет более эффективна в рамках используемых алгоритмов и конкретной архитектуры. В зависимости от результатов этого анализа можно рассматривать матрицу либо как разреженную, либо как плотную.

Эффективные методы хранения и обработки разреженных матриц на протяжении последних десятилетий вызывают интерес у широкого круга исследователей. Выясняется, что для учета разреженной структуры приходится существенно усложнять как методы хранения, так и алгоритмы обработки. Многие тривиальные с точки зрения программирования алгоритмы в разреженном случае становятся весьма сложными. В данной работе рассматривается один из достаточно показательных алгоритмов – умножение двух разреженных матриц с хранением итогового результата в разреженной матрице. Учитывая сложность и многогранность проблемы, в работе не ставится цель разработать или продемонстрировать оптимальный в некотором смысле алгоритм, равно как и дать полный обзор текущего состояния дел в данной области. Речь идет о демонстрации некоторых проблем, возникающих при работе с разреженными матрицами, а также возможных подходов к их решению с учетом возможностей современных многоядерных архитектур и инструментов для разработки программ в системах с общей памятью. Используемые алгоритмы, как правило, являются классическими и опубликованы в открытой печати (см., например, [1, 2]). Работа может быть использована в рамках лабораторного практикума по курсам «Параллельные численные методы», «Анализ производительности и оптимизация программ», «Инструменты для параллельного программирования в системах с общей памятью».

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение некоторых принципов хранения и алгоритмов обработки разреженных матриц.

Данная цель предполагает решение следующих основных задач:

1. Изучение способов хранения разреженных матриц.
2. Разработка программной инфраструктуры для проведения экспериментов – генератора тестовых задач, автоматизации анализа корректности путем сравнения с эталонной версией из библиотеки Intel Math Kernel Library (MKL) и др.
3. Разработка «наивной» программной реализации (по определению) разреженного матричного умножения.

4. Демонстрация алгоритмической оптимизации в разреженном матричном умножении.
5. Ознакомление с идеологией двухфазной обработки разреженных матриц – разделением на *символическую* и *численную фазу*. Выполнение соответствующей программной реализации.
6. Распараллеливание разреженного матричного умножения в системах с общей памятью с использованием OpenMP и Intel Threading Building Blocks (TBB).
7. Изучение способов балансировки нагрузки с использованием OpenMP и TBB.

1.2. Структура работы

Работа построена следующим образом: дана постановка задачи разреженного матричного умножения, рассмотрены вопросы хранения разреженных матриц. Приведено описание генератора тестовых задач и соображения по выбору таких задач. Описана последовательная программная реализация для вычисления произведения разреженных матриц в формате CRS (CSR) по определению. Описаны алгоритмы транспонирования. Приведены результаты экспериментов и анализ эффективности. Проведена алгоритмическая оптимизация. Выполнена реализация другого алгоритма матричного умножения с лучшими временными показателями. Описана широко распространенная идея разделения алгоритмов разреженной алгебры на две фазы – символическую и численную. На примере наиболее быстрого из демонстрируемых в работе алгоритмов выполнено разделение на фазы. Далее рассмотрены параллельные реализации финального алгоритма с использованием OpenMP и TBB, включая балансировку нагрузки.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 четырехъядерных процессора Intel Xeon E5520 (2.27 GHz)
Память	16 Gb
Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2008
Компилятор, профилиров-	Intel Parallel Studio XE

щик, отладчик	
Математическая библиотека	Intel MKL v. 10.2.5.035

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий.

1. Обсудить понятие разреженной матрицы, применение таких матриц и возникающие при этом вопросы и проблемы.
2. Рассмотреть постановку задачи матричного умножения и возможные проблемы при ее решении в разреженном случае.
3. Рассказать слушателям о специфике хранения и обработки разреженных матриц. Дать форматы хранения разреженных матриц, сравнить их.
4. Предложить механизм генерации тестовых задач, обсудить выбор таких задач.
5. Рассмотреть некоторые алгоритмы умножения разреженных матриц (по определению, двухфазный в разных вариантах) и смежные вопросы (транспонирование, проверка корректности и т.д.).
6. Выполнить программную реализацию в последовательной версии, собрать и проанализировать результаты вычислительных экспериментов.
7. Обсудить параллельную реализацию алгоритма, показавшего лучшие результаты. Выполнить программную реализацию с использованием OpenMP и TBB, провести анализ масштабируемости.
8. Принять меры для балансировки нагрузки, провести анализ масштабируемости.

2. Постановка задачи матричного умножения

Будем для упрощения рассматривать задачу для случая квадратных матриц. Пусть A и B – квадратные матрицы размера N x N, в которых процент ненулевых элементов мал[1]. Будем считать, что элементы матриц A и B – вещественные числа (далее в программной реализации будем использовать числа с плавающей точкой двойной точности).

[1] Далее в разделе «генерация тестовых задач» будет уточнено, с какими именно матрицами мы будем работать в данной лабораторной работе.

Требуется найти матрицу C = A * B, где символ * соответствует матричному умножению.

В приведенном примере (рис. 1) ненулевые элементы расположены таким образом, что в результирующей матрице C большинство элементов равны нулю. Заметим, что так будет не всегда. Нередко в процессе умножения двух разреженных матриц получится плотная матрица. В рамках данной работы предполагается, что матрица C также является разреженной.

A

1				2	
		3	4		
			2		5
	7				

*

B

3					1
		1			
			3		
1		3			
	2			4	

=

C

5		6			1
			12		
	10		6	20	
		7			

Рис. 1. Умножение разреженных матриц

3. Форматы хранения разреженных матриц

Изучению способов хранения разреженных матриц посвящено немало литературы (см., например, работы [1, 2, 4]). В лабораторной работе мы изучим несколько широко распространенных форматов хранения, в том числе поддерживаемых математическими библиотеками и пакетами (Intel MKL, PETSc, Matlab и др.).

При описании мы будем ориентироваться на матрицы размера N x N, в которых NZ элементов являются ненулевыми, $NZ << N^2$.

3.1. Координатный формат хранения

Один из наиболее простых для понимания форматов хранения разреженных матриц – координатный формат. Элементы матрицы и ее структура хранятся в трех массивах, содержащих значения, их X и Y координаты (рис. 2).

Оценим объем необходимой памяти (M).

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

В координатном формате: M = 8 NZ + 4 NZ + 4 NZ = 16 NZ байт (предполагается, что $N < 2^{32}$; в противном случае необходимо использовать 64-битный вариант беззнакового целого для хранения индексов).

Ясно, что 16 NZ << 8 N^2, если NZ << N^2.

Пример:

A

1				2	
		3	4		
			8		5
	7	1			6

Структура хранения:

1	2	3	4	8	5	7	1	6

Value

0	0	1	1	3	3	5	5	5

Row

0	4	2	3	3	5	1	2	5

Col

Рис. 2. Координатный формат хранения

Координатный формат можно использовать как в упорядоченном виде, когда в массиве **`Value`** хранятся элементы матрицы построчно (например, слева направо, сверху вниз), так и в неупорядоченном. Неупорядоченное представление существенно упрощает операции вставки/удаления новых элементов, но приводит к переборным поискам. Упорядоченный вариант позволяет быстрее находить все элементы нужной строки (столбца, если упорядочивать сначала по столбцам), но приводит к перепаковкам при вставках/удалениях элементов. В целом координатный формат достаточно прост, что является его несомненным достоинством, но оказывается недостаточно эффективным с точки зрения использования памяти и основных алгоритмов обработки.

3.2. Формат хранения CRS (CSR) и его модификации

Формат хранения, широко распространенный под названием CSR (Compressed Sparse Rows) или CRS (Compressed Row Storage), призван устранить некоторые недоработки координатного представления. Так, по-прежнему используются три массива. Первый массив хранит значения элементов построчно (строки рассматриваются по порядку сверху вниз), второй – номера столбцов для каждого элемента, а третий заменяет номера строк, используемые в координатном формате, на индекс начала каждой строки (рис. 3).

Отметим, что количество элементов массива **`RowIndex`** равно N + 1. При этом элементы строки *i* в массиве **`Value`** находятся по индексам от **`RowIndex[i]`** до **`RowIndex[i + 1] – 1`** включительно (*i*-ый элемент массива **`RowIndex`** указывает на начало *i*-ой строки). Исходя из этого обрабатывается случай пустых строк, а также добавляется «лишний» элемент в массив **`RowIndex`** – устраняется особенность при доступе к элементам последней строки. Данный элемент хранит номер последнего ненулевого

элемента матрицы A плюс 1, что соответствует количеству ненулевых элементов NZ.

Оценим объем необходимой памяти.

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

В координатном формате: M = 16 NZ байт.

В формате CRS: M = 8 NZ + 4 NZ + 4 (N + 1) = 12 NZ + 4 N + 4.

В часто встречающемся случае, когда N + 1 < NZ, данный формат является более эффективным, чем координатный, с точки зрения объема используемой памяти.

Пример:

A

1				2	
		3	4		
			8		5
	7	1			6

Структура хранения:

1	2	3	4	8	5	7	1	6

Value

0	4	2	3	3	5	1	2	5

Col

0	2	4	4	6	6	9

RowIndex

Рис. 3. Координатный формат хранения

Формат имеет целый ряд модификаций. Перечислим некоторые из них.

- Возможна индексация как с нуля, так и с единицы (C/Fortran).
- Заметим, что в базовом варианте CRS строки рассматриваются по порядку, но элементы внутри каждой строки могут быть как упорядочены по номеру столбца, так и не упорядочены. Структура легко восстанавливается, поскольку номер столбца для каждого элемента хранится в массиве **`Col`**. Оба варианта имеют свои особенности с точки зрения последующей обработки. И упорядоченный, и неупорядоченный варианты востребованы на практике. Далее в работе мы будем использовать упорядоченный вариант. Вопросы, связанные с упорядоченностью, будут раскрыты при описании алгоритмов.
- Существует модификация CRS с четырьмя массивами. Четвертый массив хранит индексы элементов, идущих в конце строки. В таком варианте представления матрицы строки могут не быть упорядочены, что позволяет осуществлять перестановку строк путем выполнения элементарных операций над индексами, исключая перепаковку. При этом, разумеется, тратится дополнительная память, а также несколько замед-

ляются операции редактирования, в которых требуется обрабатывать четвертый массив.

- В некоторых источниках рассмотренный формат упоминается как Yale format, в некоторых – как AIJ. Иногда применяется модификация, состоящая в том, что массивы хранятся в памяти в другом порядке.
- Все, что было описано выше, может быть с равным успехом применено не к строкам, а к столбцам. В итоге получается столбцовый формат CSC (CCS) вместо строчного CSR (CRS). Для ряда алгоритмов более удобен строчный формат, для ряда – столбцовый.

Заметим, что доступ к элементам строки в формате CRS организуется крайне просто, что дает возможность использовать следующую реализацию алгоритма вычисления $y = Ax$ – умножение разреженной матрицы A размера N x N на плотный вектор x размера N.

```
// Цикл по строкам матрицы A
for (i = 0; i < N; i++)
{
  // Вычисляем i-ю компоненту вектора y
  sum = 0;
  j1 = RowIndex[i]; j2 = RowIndex[i + 1];
  for (j = j1; j < j2; j++)
    sum += Value[j] * x[Col[j]];
  y[i] = sum;
}
```

Внимательный читатель может обдумать данный фрагмент кода и рассмотреть вопрос его эффективности на современных вычислительных архитектурах, использующих многоуровневую организацию подсистемы памяти. В этом смысле необходимо отметить, что операция **`x[Col[j]]`** приводит к обращению в вектор x по адресам, зачастую достаточно разбросанным по памяти, что выражается в кеш-промахах и потере производительности, но это представляет собой тему отдельного разговора (интересующиеся могут обратиться к исследованиям Дж. Деммеля, изложенным в виде презентаций на его сайте [14]).

Операции вставки/удаления элементов приводят к перепаковкам, что является недостатком рассмотренного представления.

Отметим, что приведенное описание не является полным. Так, известны также диагональный формат хранения, разные варианты блочного представления – разреженная структура из плотных блоков (BCRS), плотная структура из разреженных блоков и т.д.

Далее в работе мы будем использовать формат CRS в виде трех массивов, расположенных в памяти в порядке **`Value`**, **`Col`**, **`RowIndex`**. Индексация массивов в стиле языка C – с нуля. Элементы в каждой строке упорядочи-

ваются по номеру столбца. В матрице хранятся числа с плавающей точкой двойной точности (тип данных **double** в языке C).

Будем использовать следующую структуру данных:

```
struct crsMatrix
{
  int N;  // Размер матрицы (N x N)
  int NZ; // Кол-во ненулевых элементов

  // Массив значений (размер NZ)
  double* Value;
  // Массив номеров столбцов (размер NZ)
  int* Col;
  // Массив индексов строк (размер N + 1)
  int* RowIndex;
};
```

4. Генерация тестовых задач

Как известно, к анализу «времени работы» алгоритмов можно подходить как с теоретической, так и с практической точки зрения. Теоретический подход предполагает построение и доказательство аналитических оценок трудоемкости. При этом пытаются определить, как растет время работы в зависимости от объема входных данных (оценки O, Ω, Θ, см., например, [9]). Конечно, такой подход не дает возможности точно прогнозировать время работы программы в секундах, но он теоретически обоснован, абстрагирован от конкретной вычислительной архитектуры и деталей программной реализации. Теоретический подход, основанный на построении оценок трудоемкости, является общепринятым в анализе сложности алгоритмов.

К сожалению, в некоторых случаях аналитически получить достаточно точные оценки времени работы алгоритма затруднительно. Иногда необходимо оценить время в стандартных единицах – секундах, минутах, часах, ориентируясь на конкретную вычислительную архитектуру и средства программирования (компиляторы, библиотеки и т.д.). В таких ситуациях приходится прибегать к вычислительным экспериментам. Отметим, что постановка соответствующего эксперимента – отдельная сложная задача. Необходимо позаботиться о достаточно хорошей программной реализации выбранного алгоритма, адаптировать ее для целевой вычислительной архитектуры, грамотно реализовать замеры времени, подобрать репрезентативные наборы тестовых данных, на которых будут производиться измерения и т.д. (более подробная информация может быть найдена в источниках [5-7]). В каких-то вопросах нам помогут инструменты программирования

(профилировщик, хороший оптимизирующий компилятор), но многие проблемы придется решать самостоятельно.

В данной лабораторной работе мы не ставим целью построение оптимального алгоритма умножения. Для иллюстрации проблем при работе с разреженными матрицами и подходов к их решению будем использовать достаточно специфические матрицы, не претендуя на репрезентативность в общем случае. Пусть матрицы A и B содержат не более 1% ненулевых элементов. На заполнение результирующей матрицы C в экспериментах будем обращать меньшее внимание, но позаботимся о том, чтобы оно не превышало 10% (не совсем разреженная матрица, но подойдет для иллюстрации работы алгоритма). Будем формировать матрицы A и B при помощи датчика случайных чисел. Данная задача включает два этапа – построение *портрета (шаблона) матрицы* и наполнение этого портрета конкретными значениями. Рассмотрим пример портрета матрицы (символом X обозначены позиции ненулевых элементов), а также его представление в формате CRS. CRS-представление портрета определяется парой массивов **`Col`** и **`RowIndex`**, массив **`Value`** заполняется по мере наполнения матрицы конкретными значениями (рис. 4).

Пример:

Портрет матрицы

X				X	
		X	X		
			X		X
	X	X			X

Структура хранения портрета:

0	4	2	3	3	5	1	2	5

Col

0	2	4	4	6	6	9

RowIndex

Рис. 4. Портрет матрицы и его структура в формате CRS

Пусть N = 10000. Для формирования портрета матрицы A применим следующую схему: будем генерировать случайным образом позиции 50 ненулевых элементов в каждой строке матрицы. Для формирования портрета матрицы B применим другой подход: будем наращивать количество ненулевых элементов от строки к строке по кубическому закону так, чтобы последняя строка содержала максимальное количество (50) ненулевых элементов.

Выполним программную реализацию алгоритмов генерации матриц.

```
// Флаг - был ли инициализирован генератор случайных чисел
bool isSrandCalled = false;

double next()
{
```

```
  return ((double)rand() / (double)RAND_MAX);
}

// Генерирует квадратную матрицу в формате CRS
// (3 массива, индексация с нуля)
// В каждой строке cntInRow ненулевых элементов
void GenerateRegularCRS(int seed, int N,
  int cntInRow, crsMatrix& mtx)
{
  int i, j, k, f, tmp, notNull, c;

  if (!isSrandCalled)
  {
    srand(seed);
    isSrandCalled = true;
  }

  notNull = cntInRow * N;
  InitializeMatrix(N, notNull, mtx);
  for(i = 0; i < N; i++)
  {
    // Формируем номера столбцов в строке i
    for(j = 0; j < cntInRow; j++)
    {
      do
      {
        mtx.Col[i * cntInRow + j] = rand() % N;
        f = 0;
        for (k = 0; k < j; k++)
          if (mtx.Col[i * cntInRow + j] ==
              mtx.Col[i * cntInRow + k])
            f = 1;
      }
      while (f == 1);
    }
    // Сортируем номера столцов в строке i
    for (j = 0; j < cntInRow - 1; j++)
      for (k = 0; k < cntInRow - 1; k++)
        if (mtx.Col[i * cntInRow + k] >
            mtx.Col[i * cntInRow + k + 1])
        {
          tmp = mtx.Col[i * cntInRow + k];
          mtx.Col[i * cntInRow + k] =
                                mtx.Col[i * cntInRow + k + 1];
          mtx.Col[i * cntInRow + k + 1] = tmp;
        }
  }
  // Заполняем массив значений
  for (i = 0; i < cntInRow * N; i++)
```

```
    mtx.Value[i] = next() * MAX_VAL;
  // Заполняем массив индексов строк
  c = 0;
  for (i = 0; i <= N; i++)
  {
    mtx.RowIndex[i] = c;
    c += cntInRow;
  }
}

// Генерирует квадратную матрицу в формате CRS
// (3 массива, индексация с нуля)
// Число ненулевых элементов в строках растет
// от 1 до cntInRow. Закон роста - кубическая парабола
void GenerateSpecialCRS(int seed, int N,
  int cntInRow, crsMatrix& mtx)
{
  if (!isSrandCalled)
  {
    srand(seed);
    isSrandCalled = true;
  }

  double end = pow((double)cntInRow, 1.0 / 3.0);
  double step = end / N;
  vector<int>* columns = new vector<int>[N];
  int NZ = 0;
  for (int i = 0; i < N; i++)
  {
    int rowNZ = int(pow((double(i + 1) * step), 3) + 1);
    NZ += rowNZ;
    int num1 = (rowNZ - 1) / 2;
    int num2 = rowNZ - 1 - num1;
    if (rowNZ != 0)
    {
      if (i < num1)
      {
        num2 += num1 - i;
        num1 = i;
        for(int j = 0; j < i; j++)
          columns[i].push_back(j);
        columns[i].push_back(i);
        for(int j = 0; j < num2; j++)
          columns[i].push_back(i + 1 + j);
      }
      else
      {
        if (N - i - 1 < num2)
        {
          num1 += num2 - (N - 1 - i);
```

```
          num2 = N - i - 1;
        }
        for (int j = 0; j < num1; j++)
          columns[i].push_back(i - num1 + j);
        columns[i].push_back(i);
        for (int j = 0; j < num2; j++)
          columns[i].push_back(i + j + 1);
      }
    }
  }
  InitializeMatrix(N, NZ, mtx);
  int count = 0;
  int sum = 0;
  for (int i = 0; i < N; i++)
  {
    mtx.RowIndex[i] = sum;
    sum += columns[i].size();
    for (unsigned int j = 0; j < columns[i].size(); j++)
    {
      mtx.Col[count] = columns[i][j];
      mtx.Value[count] = next();
      count++;
    }
  }
  mtx.RowIndex[N] = sum;
  delete [] columns;
}
```

Тогда имеем следующие характеристики матриц после генерации:

Матрица	Количество ненулевых элементов	Процент ненулевых элементов
A	500 000	0,5%
B	130 775	0,13%
C	5 681 531	5,68%

Портреты матриц выглядят следующим образом (рис. 5):

Разумеется, можно выбирать для экспериментов тестовые матрицы другой структуры. При проведении экспериментов с разными видами матриц обнаруживается немало интересных моментов, связанных с алгоритмической и программной оптимизацией, оптимизацией под конкретную структуру, эффективностью распараллеливания и т.д. В целом анализ изменений, происходящих при смене структуры и размеров матриц, является достаточно поучительным и может быть проведен в рамках дополнительных заданий к работе.

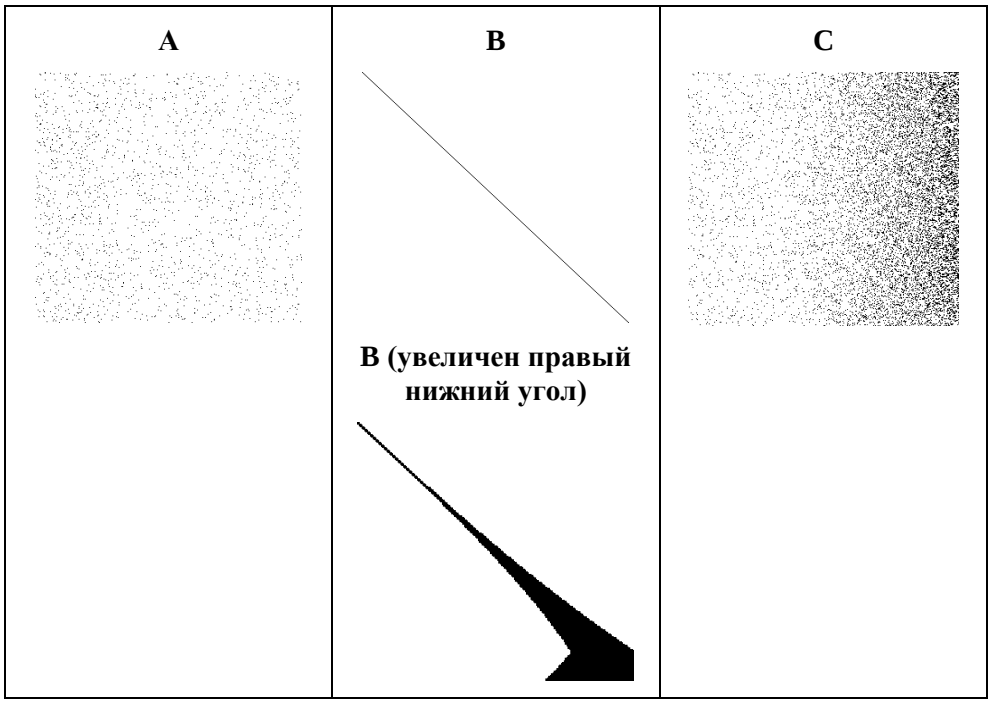

Рис. 5. Портреты матриц

5. Вспомогательные функции

Для комфортной работы, как правило, полезно разработать некоторую инфраструктуру, которая будет использоваться при проведении экспериментов. В данном случае как минимум необходимо реализовать следующие операции:

1. *Инициализация матрицы* – выделение памяти под структуру данных для хранения матрицы в формате CRS. Эта функция уже была нами использована в описанных выше генераторах матриц.

```
void InitializeMatrix(int N, int NZ, crsMatrix &mtx)
{
  mtx.N = N;
  mtx.NZ = NZ;
  mtx.Value     = new double[NZ];
  mtx.Col       = new int[NZ];
  mtx.RowIndex = new int[N + 1];
}
```

2. *Удаление матрицы* – освобождение выделенной ранее памяти.

```
void FreeMatrix(crsMatrix &mtx)
{
```

```
  delete[] mtx.Value;
  delete[] mtx.Col;
  delete[] mtx.RowIndex;
}
```

3. *Сравнение двух разреженных матриц* в формате CRS. Заметим, что данная операция является не такой простой, как кажется. Дело в том, что частичное несовпадение CRS-структур еще не означает, что матрицы отличны друг от друга. Так, какой-то из нулевых элементов может храниться, то есть присутствовать в портрете матрицы и в массиве значений (значение равно нулю).

Для корректной реализации указанной процедуры можно, например, запрограммировать алгоритм вычитания матриц и сравнивать полученный результат с нулевой матрицей (с некоторой точностью). Тривиальный алгоритм предусматривает выполнение операций сравнения через плотные матрицы.

```
int CompareMatrix(crsMatrix mtx1, crsMatrix mtx2,
  double &diff)
{
  int N;
  int i, j;
  int i1, i2;
  double **p;

  // Совпадает ли размер?
  if (mtx1.N != mtx2.N)
    return 1; // Не совпал размер

  N = mtx1.N;
  // Создание плотной матрицы
  p = new double*[N];
  for (i = 0; i < N; i++)
  {
    p[i] = new double[N];
    for (j = 0; j < N; j++)
      p[i][j] = 0.0;
  }
  // Копирование первой матрицы в плотную
  for (i = 0; i < N; i++)
  {
    i1 = mtx1.RowIndex[i];
    i2 = mtx1.RowIndex[i + 1] - 1;
    for (j = i1; j <= i2; j++)
      p[i][mtx1.Col[j]] = mtx1.Value[j];
  }
  // Вычитание второй матрицы из плотной
  for (i = 0; i < N; i++)
```

```
  {
    i1 = mtx2.RowIndex[i];
    i2 = mtx2.RowIndex[i + 1] - 1;
    for (j = i1; j <= i2; j++)
      p[i][mtx2.Col[j]] -= mtx2.Value[j];
  }
  // Вычисление максимального отклонения
  double max = 0.0;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      if (fabs(p[i][j]) > max)
        max = fabs(p[i][j]);
  diff = max;
  // Удаление плотной матрицы
  for (i = 0; i < N; i++)
    delete[] p[i];
  delete[] p;

  return 0; // Совпал размер
}
```

Разумеется, такой вариант не будет работать при достаточно больших размерах матриц из-за ограничений по памяти, также он крайне не эффективен с точки зрения производительности. Его хватит лишь для проверки некоторых алгоритмов на малых размерностях.

Для более корректной разработки подпрограммы сравнения разреженных матриц необходима реализация алгоритма вычитания разреженных матриц. Далее в работе мы будем использовать подпрограмму **mkl_dcsradd()** из библиотеки Intel MKL. Указанная подпрограмма выполняет сложение двух разреженных матриц: C = A + beta*op(B), где beta – вещественный множитель (в нашем случае -1), а параметр op позволяет указать, что матрица B должна быть транспонирована (в нашем случае это не требуется). К сожалению, функция **mkl_dcsradd()** работает только с разновидностью CRS, в которой нумерация начинается с единицы. Это требует пролога с приведением A и B к искомому формату и эпилога с возвратом A и B к исходному представлению.

Выполним программную реализацию. Замечания по настройке параметров функции приведены непосредственно в коде подпрограммы в виде комментариев.

```
// Принимает 2 квадратных матрицы в формате CRS
//   (3 массива, индексация с нуля)
// Возвращает max|Cij|, где C = A - B
// Для сравнения использует функцию из MKL
// Возвращает признак успешности операции: 0 - OK,
//   1 - не совпадают размеры (N)
int SparseDiff(crsMatrix A, crsMatrix B, double &diff)
{
```

```
  if (A.N != B.N)
    return 1;
  int n = A.N;
  // Будем вычислять C = A - B, используя MKL
  // Структуры данных в стиле MKL
  double *c = 0; // Значения
  int *jc = 0;   // Номера столбцов (нумерация с единицы)
  int *ic;       // Индексы первых элементов строк
                 // (нумерация с единицы)

  // Настроим параметры для вызова функции MKL
  // Переиндексируем матрицы A и B с единицы
  int i, j;
  for (i = 0; i < A.NZ; i++)
    A.Col[i]++;
  for (i = 0; i < B.NZ; i++)
    B.Col[i]++;
  for (j = 0; j <= A.N; j++)
  {
    A.RowIndex[j]++;
    B.RowIndex[j]++;
  }
  // Используется функция, вычисляющая C = A + beta*op(B)
  char trans = 'N'; // говорит о том, что op(B) = B -
                    // не нужно транспонировать B
// Параметр, влияющий на то, как будет выделяться память
// request = 0: память для результирующей матрицы должна
//              быть выделена заранее
// Если мы не знаем, сколько памяти необходимо для хранения
// результата, необходимо:
//  1) выделить память для массива индексов строк ic:
//       "Кол-во строк + 1" элементов;
//  2) вызвать функцию с параметром request = 1 -
//       в массиве ic будет заполнен последний элемент
//  3) выделить память для массивов c и jc:
//       кол-во элементов = ic[Кол-во строк] - 1
//  4) вызвать функцию с параметром request = 2
  int request;
// Еще один существенный момент: есть возможность
// настроить, нужно ли упорядочивать матрицы A, B и C.
// У нас предполагается, что все матрицы упорядочены,
// следовательно, выбираем вариант "No-No-Yes", который
// соответствует любому значению, кроме целых чисел
// от 1 до 7 включительно
  int sort = 8;
// beta = -1 -> C = A + (-1.0) * B;
  double beta = -1.0;
// Количество ненулевых элементов
// Используется только если request = 0
```

```
  int nzmax = -1;
// Служебная информация
  int info;
// Выделим память для индекса в матрице C
  ic = new int[n + 1];
// Сосчитаем количество ненулевых элементов в матрице C
  request = 1;
  mkl_dcsradd(&trans, &request, &sort, &n, &n,
              A.Value, A.Col, A.RowIndex, &beta,
              B.Value, B.Col, B.RowIndex,
              c, jc, ic,
              &nzmax, &info);
  int nzc = ic[n] - 1;
  c = new double[nzc];
  jc = new int[nzc];
// Сосчитаем C = A - B
  request = 2;
  mkl_dcsradd(&trans, &request, &sort, &n, &n,
              A.Value, A.Col, A.RowIndex, &beta,
              B.Value, B.Col, B.RowIndex,
              c, jc, ic,
              &nzmax, &info);
// Сосчитаем max|Cij|
  diff = 0.0;
  for (i = 0; i < nzc; i++)
  {
    double var = fabs(c[i]);
    if (var > diff)
      diff = var;
  }
// Приведем к исходному виду матрицы A и B
  for (i = 0; i < A.NZ; i++)
    A.Col[i]--;
  for (i = 0; i < B.NZ; i++)
    B.Col[i]--;
  for (j = 0; j <= n; j++)
  {
    A.RowIndex[j]--;
    B.RowIndex[j]--;
  }
// Освободим память
  delete [] c;
  delete [] ic;
  delete [] jc;

  return 0;
}
```

4. *Умножение двух разреженных матриц* в формате CRS – эталонная версия. Для проверки собственных реализаций алгоритма умножения

разреженных матрицы необходимо обеспечить контроль за правильностью результата. Будем использовать функцию **mkl_dcsrmultcsr()** в качестве эталона. Отметим, что в целом она весьма похожа по своему интерфейсу на функцию **mkl_dcsradd()** и имеет те же особенности (упорядоченный формат хранения в исходных матрицах, нумерация с единицы, двухуровневый механизм работы, связанный с необходимостью выделения памяти и др.).

Выполним программную реализацию.

```
// Принимает 2 квадратных матрицы в формате CRS
//   (3 массива, индексация с нуля)
// Возвращает C = A * B, C - в формате CRS
//   (3 массива, индексация с нуля)
// Память для C в начале считается не выделенной
// Возвращает признак успешности операции:
//   0 - OK, 1 - не совпадают размеры (N)
// Возвращает время работы
int SparseMKLMult(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;
  int n = A.N;
  // Настроим параметры для вызова функции MKL
  // Переиндексируем матрицы A и B с единицы
  int i, j;
  for (i = 0; i < A.NZ; i++)
    A.Col[i]++;
  for (i = 0; i < B.NZ; i++)
    B.Col[i]++;
  for (j = 0; j <= n; j++)
  {
    A.RowIndex[j]++;
    B.RowIndex[j]++;
  }
  // Используется функция, вычисляющая C = op(A) * B
  char trans = 'N'; // говорит о том, op(A) = A -
                    // не нужно транспонировать A
// Параметр, влияющий на то, как будет выделяться память
// request = 0: память для результирующей матрицы
//              должна быть выделена заранее
// Если мы не знаем, сколько памяти необходимо
//     для хранения результата, необходимо:
// 1) выделить память для массива индексов строк ic:
//     "Кол-во строк+1" элементов;
// 2) вызвать функцию с параметром request = 1 -
//     в массиве ic будет заполнен последний элемент
// 3) выделить память для массивов c и jc
```

```
//     (кол-во элементов = ic[Кол-во строк]-1)
// 4) вызвать функцию с параметром request = 2
  int request;
// Еще один нетривиальный момент: есть возможность
// настроить, нужно ли упорядочивать матрицы A, B и C.
// У нас предполагается, что все матрицы упорядочены,
// следовательно, выбираем вариант "No-No-Yes", который
// соответствует любому значению, кроме целых чисел
// от 1 до 7 включительно
  int sort = 8;
// Количество ненулевых элементов.
// Используется только если request = 0
  int nzmax = -1;
// Служебная информация
  int info;
  clock_t start = clock();
// Выделим память для индекса в матрице C
  C.RowIndex = new int[n + 1];
// Сосчитаем количество ненулевых элементов в матрице C
  request = 1;
  C.Value = 0;
  C.Col = 0;
  mkl_dcsrmultcsr(&trans, &request, &sort, &n, &n, &n,
                  A.Value, A.Col, A.RowIndex,
                  B.Value, B.Col, B.RowIndex,
                  C.Value, C.Col, C.RowIndex,
                  &nzmax, &info);
  int nzc = C.RowIndex[n] - 1;
  C.Value = new double[nzc];
  C.Col = new int[nzc];
// Сосчитаем C = A * B
  request = 2;
  mkl_dcsrmultcsr(&trans, &request, &sort, &n, &n, &n,
                  A.Value, A.Col, A.RowIndex,
                  B.Value, B.Col, B.RowIndex,
                  C.Value, C.Col, C.RowIndex,
                  &nzmax, &info);
  C.N = n;
  C.NZ = nzc;

  clock_t finish = clock();

  // Приведем к нормальному виду матрицы A, B и C
  for (i = 0; i < A.NZ; i++)
    A.Col[i]--;
  for (i = 0; i < B.NZ; i++)
    B.Col[i]--;
  for (i = 0; i < C.NZ; i++)
    C.Col[i]--;
  for (j = 0; j <= n; j++)
```

```
  {
    A.RowIndex[j]--;
    B.RowIndex[j]--;
    C.RowIndex[j]--;
  }
  time = double(finish - start) / double(CLOCKS_PER_SEC);
  return 0;
}
```

Тогда в функции **`main()`** в дальнейшем мы сможем выполнять проверку, вызывая свою реализацию умножения, **`SparseMKLMult()`**, а затем **`SparseDiff()`** с последующим сравнением максимума модуля разности элементов с некоторым достаточно малым **`EPSILON`**.

6. Наивная последовательная реализация

6.1. Общая схема решения задачи

Приступим к обсуждению алгоритмов умножения разреженных матриц. Прежде всего, попробуем понять, как выполнить умножение непосредственно по определению.

A

X				X	
		X	X		
			X		X
	X	X			X

B

X		X		X	
		X			
	X				
				X	
					X

C

X		X		X	
	X				
					X
	X	X			X

Рис. 6. Умножение матриц

Определение предполагает, что элемент в строке i и столбце j матрицы C вычисляется как скалярное произведение i-ой строки матрицы A и j-го столбца матрицы B. Какие особенности вносит разреженное представление?

Во-первых, используемая структура данных, построенная на основе формата CRS, предполагает хранение только ненулевых элементов, что несколько усложняет программирование вычисления скалярного произведения, но одновременно уменьшает количество арифметических операций. При вычислении скалярных произведений нет необходимости умножать нули и накапливать полученный нуль в частичную сумму, что положи-

тельно влияет на сокращение времени счета. Пусть, например, в первом и втором векторах находится по 1% ненулевых элементов, при этом только десятая часть этих элементов расположена на соответствующих друг другу позициях. В этом случае расчет с использованием информации о структуре векторов может использовать в 1000 раз меньшее число умножений и сложений. Учитывая, что таких пар векторов N^2, получается существенное сокращение объема вычислений. К сожалению, не все так просто. Учет структуры векторов тоже требует машинного времени. Необходимо выполнить сопоставление номеров ненулевых элементов с целью обнаружения пар значений, которые необходимо перемножить и накопить в частичную сумму. В алгоритме появятся ветвления.

Во-вторых, необходимо научиться выделять вектора в матрицах A и B. В соответствии с определением, речь идет о строках матрицы A и столбцах матрицы B. Выделить строку матрицы в формате CRS не представляет труда: *i*-я строка может быть легко найдена, так как ссылки на первый элемент (**`RowIndex[i]`**) и последний элемент (**`RowIndex[i+1]-1`**) известны, что позволяет получить доступ к значениям элементов и номерам столбцов, хранящихся в массивах **`Value`** и **`Col`** соответственно. Таким образом, проход по строке выполняется за время, пропорциональное числу ненулевых элементов в указанной строке, а проход по всем строкам[2] – за время, пропорциональное NZ.

Проблема возникает с выделением столбца. Чтобы найти элементы столбца *j* необходимо просмотреть массив **`Col`** (~NZ операций) и выделить все элементы, у которых в соответствующей ячейке массива **`Col`** записано число *j*. Если это нужно проделать для каждого столбца, необходимо ~NZ*N операций, что выглядит неэффективным.

Возможное, но не единственное решение проблемы состоит *в транспонировании матрицы B*. Если удастся сделать это достаточно быстро, появится возможность эффективно работать со столбцами исходной матрицы B. Другой вариант – для каждой CRS-матрицы, которая может понадобиться в столбцовом представлении, дополнительно хранить транспонированный портрет. Сэкономив время на транспонировании, придется смириться с расходами времени на поддержание дополнительного портрета в актуальном состоянии. Есть и другие варианты решения проблемы, никак не связанные с транспонированием матрицы B. Так, например, алгоритм Густавсона [2, 10] упоминается во многих источниках как один из самых быстрых. Краткое описание алгоритма и численные результаты, характеризующие его работу, приведены в разделе 9. Реализацию этого алгоритма предлагается выполнить в рамках одного из дополнитель-

---

[2] Предполагается, что элементы используются в теле цикла для выполнения каких-либо операций.

ных заданий. В основной части данной лабораторной работе будем использовать подход, основанный на транспонировании матрицы B.

В-третьих, необходимо научиться записывать посчитанные элементы в матрицу C. Учитывая, что C хранится в формате CRS, важно избежать перепаковок. Для этого нужно обеспечить пополнение матрицы C ненулевыми элементами последовательно, по строкам – слева направо, сверху вниз. Это означает, что часто используемый в вычислениях на бумаге метод «возьмем первый столбец матрицы B, запишем его над матрицей A и перемножим на все строки…» не подходит. Нужно делать наоборот – брать первую строку матрицы A и умножать ее по очереди на все столбцы матрицы B (строки матрицы B^T). В этом случае обеспечивается последовательное пополнение матрицы C, позволяющее дописывать элементы в массивы **`Value`** и **`Col`**, а также «налету» формировать массив **`RowIndex`**. Вопрос организации работы с динамической памятью для структуры хранения матрицы C обсудим позже.

Подведем промежуточные итоги. Необходимо сделать следующее:

1. Реализовать транспонирование разреженной матрицы и применить его к матрице B.
2. Инициализировать структуру данных для матрицы C, обеспечить возможность ее пополнения элементами.
3. Последовательно перемножить каждую строку матрицы A на каждую из строк матрицы B^T, записывая в C полученные результаты и формируя ее структуру. При умножении строк реализовать сопоставление с целью выделения пар ненулевых элементов.

Еще один непростой момент: в процессе вычисления скалярного произведения на бумаге (в точной арифметике) может получиться нуль, причем не только в том случае, когда при сопоставлении векторов соответствующих друг другу пар не обнаружилось, но и просто как естественный результат. В арифметике с плавающей точкой нуль может получиться еще и в связи с ограничениями на представление чисел и погрешностью вычислений (см. стандарт IEEE-754). Нули, получившиеся в процессе вычислений, можно как хранить в матрице C (в векторе **`Value`** в соответствующей позиции), так и не хранить. Оба подхода имеют право на существование. Далее мы еще не раз вернемся к этой теме, обсуждая алгоритмы умножения.

6.2. Алгоритм транспонирования

Начнем с обсуждения алгоритма транспонирования разреженной матрицы. Для плотных матриц алгоритм достаточно прост. У тех, кто начинает работать с разреженными матрицами, иногда возникает иллюзия, что и там все очень просто. Конечно, если цель состоит в том, чтобы как-нибудь пра-

вильно транспонировать матрицу, не обращая внимания на время работы программы, – задача сильно упрощается. Если же производительность важна, приходится немного подумать, в противном случае есть риск серьезно замедлить все приложение.

В чем суть операции транспонирования? Если создать нулевую матрицу, а далее добавлять туда по одному элементу, выбирая их из CRS-структуры исходной матрицы, придется столкнуться с необходимостью большого количества перепаковок. Чтобы этого избежать, нужно формировать транспонированную матрицу построчно. Для этого можно брать столбцы исходной матрицы и создавать из них строки результирующей матрицы. Однако выделить из CRS-матрицы столбец *i* не так просто. Необходимо другое решение. Обсуждение проблемы транспонирования разреженной матрицы изложено в книге [2]. Рассмотрим основные идеи описанного в [2] алгоритма, предложенного Густавсоном.

1. Сформируем N одномерных векторов для хранения целых чисел, а также N векторов для хранения вещественных чисел. N в данном случае соответствует числу столбцов исходной матрицы.
2. В цикле просмотрим все строки исходной матрицы, для каждой строки – все ее элементы. Пусть текущий элемент находится в строке *i*, столбце *j*, его значение равно *v*. Тогда добавим числа *i* и *v* в *j*-ые вектора для хранения целых и вещественных чисел (соответственно). Тем самым в векторах мы сформируем строки транспонированной матрицы.
3. Последовательно скопируем данные из векторов в CRS-структуру транспонированной матрицы (**`Col`** и **`Value`**), попутно формируя массив **`RowIndex`**.

Рассмотрим пример (рис. 7).

При обходе исходной матрицы A формируются вектора **`IntVectors`** и **`DoubleVectors`** (число 3 расположено в строке 0 и столбце 1, следовательно, в **`IntVectors[1]`** добавляется 0, в **`DoubleVectors[1]`** добавляется 3 и т.д.). Далее вектора последовательно формируют структуру A^T: обратите внимание на порядок следования элементов в массиве **`Value`** матрицы A^T и его соответствие порядку элементов в массиве **`DoubleVectors`** (аналогично **`Col`** и **`IntVectors`**). Для формирования **`RowIndex`** достаточно подсчитать количество элементов в каждом из N векторов. **`RowIndex[0]`** всегда равно нулю, **`RowIndex[i] = RowIndex[i-1] + «Количество элементов в векторе i-1»`**.

Задание.

Реализуйте и проверьте алгоритм транспонирования.

Недостатком алгоритма в том виде, как он описан выше, является использование дополнительной памяти. В книге [2] приводится реализация алго-

ритма на языке Fortran, лишенная этого недостатка. В статье [8] авторы дают более понятное описание, основанное на тех же принципах, но не привязанное к конкретному языку программирования. Основная идея состоит в использовании структур данных матрицы A^T для промежуточных результатов вычислений.

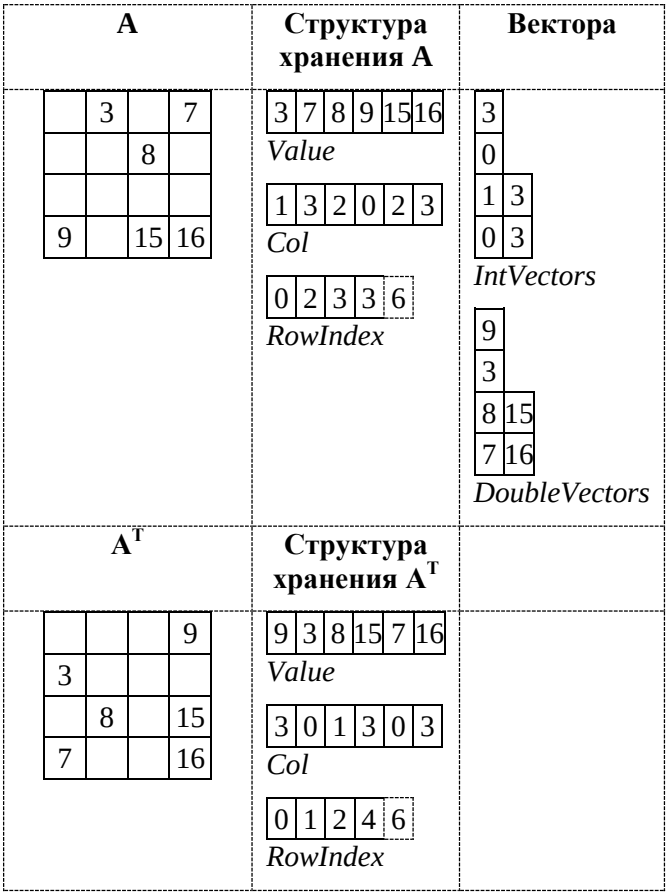

Рис. 7. Транспонирование разреженной матрицы

1. Обнулим массив **`RowIndex`** матрицы A^T. Просмотрим массив **`Col`** матрицы A и сосчитаем количество элементов в каждом столбце матрицы A (строке матрицы A^T), сохраняя результаты в массиве **`RowIndex`** матрицы A^T. Пусть при этом **`AT.RowIndex[j]`** хранит количество элементов в столбце j–1 матрицы A (j меняется от 1 до N включительно).

```
memset(AT.RowIndex, 0, (N+1) * sizeof(int));
for (i = 0; i < A.NZ; i++)
  AT.RowIndex[A.Col[i] + 1]++;
```

В нашем примере получим следующий результат (рис. 8):

A	**Структура хранения A**	**Структура хранения A^T**
	3 7 8 9 15 16 *A.Value*	*AT.Value*
	1 3 2 0 2 3 *A.Col*	*AT.Col*
	0 2 3 3 6 *A.RowIndex*	0 1 1 2 2 *AT.RowIndex*

Рис. 8. Транспонирование разреженной матрицы – этап 1

2. Подсчитаем индексы начала каждой строки в матрице A^T так, что элемент **`AT.RowIndex[i]`** хранит индекс начала (*i*–1)-ой строки.

```
S = 0;
for (i = 1; i <= A.N; i++)
{
  tmp = AT.RowIndex[i];
  AT.RowIndex[i] = S;
  S = S + tmp;
}
```

В нашем примере получим следующий результат (рис. 9):

A	**Структура хранения A**	**Структура хранения A^T**
	3 7 8 9 15 16 *A.Value*	*AT.Value*
	1 3 2 0 2 3 *A.Col*	*AT.Col*
	0 2 3 3 6 *A.RowIndex*	0 0 1 2 4 *AT.RowIndex*

Рис. 9. Транспонирование разреженной матрицы – этап 2

3. Правильно расположим в структуре результирующей матрицы значения из исходной матрицы. Сами значения и их номера строк (теперь столбцов) известны, необходимо «лишь» поместить их в корректную позицию. Для этого будем использовать массив **`AT.RowIndex`**. Заметим, что взяв очередной элемент матрицы A, имеющий значение V, координаты (*i*, *j*), мы должны добавить его в *j*-ю строку матрицы A^T. Зная, что в настоящий момент *j*-я строка матрицы A^T начинается с элемента

AT.RowIndex[j+1], будем добавлять V и *i* в массивы **AT.Value** и **AT.Col** соответственно по адресу **AT.RowIndex[j+1]**, после чего увеличим **AT.RowIndex[j+1]** на единицу. В конце элемент **AT.RowIndex[j+1]** будет хранить индекс начала (*j*+1)-ой строки, что и требуется.

```
for (i = 0; i < A.N; i++)
{
  j1 = A.RowIndex[i]; j2 = A.RowIndex[i+1];
  Col = i;              // Столбец в AT - строка в A
  for (j = j1; j < j2; j++)
  {
    V = A.Value[j];     // Значение
    RIndex = A.Col[j]; // Строка в AT
    IIndex = AT.RowIndex[RIndex + 1];
    AT.Value[IIndex] = V;
    AT.Col  [IIndex] = Col;
    AT.RowIndex[RIndex + 1]++;
  }
}
```

Итоговые результаты транспонирования представлены ниже (рис. 10).

A	**Структура хранения A**	**Структура хранения A^T**
3 7 8 9 15 16	3 7 8 9 15 16 *A.Value* 1 3 2 0 2 3 *A.Col* 0 2 3 3 6 *A.RowIndex*	9 3 8 15 7 16 *AT.Value* 3 0 1 3 0 3 *AT.Col* 0 1 2 4 6 *AT.RowIndex*

Рис. 10. Транспонирование разреженной матрицы – этап 3 (итог)

Задание.

Реализуйте алгоритм транспонирования, проведите вычислительные эксперименты, убедитесь в корректности. Сравните с предыдущей реализацией.

6.3. Создание проекта

К настоящему моменту мы выполнили всю подготовительную работу, необходимую для реализации умножения разреженных матриц. Остался по-

следний штрих – создание проекта в среде разработки **Microsoft Visual Studio 2008**.

Прежде всего, создадим новое **Решение** (**Solution**), в которое включим первый **Проект** (**Project**) данной лабораторной работы. Последовательно выполните следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2008**.
- В меню File выполните команду **New→Project…**.
- Как показано на рис. 11, в диалоговом окне **New Project** в типах проекта выберите **Win32**, в шаблонах **Win32 Console Application**, в поле **Solution** введите **07_SparseMM**, в поле **Name** – **01_Naive**, в поле **Location** укажите путь к папке с лабораторными работами курса – **c:\ParallelCalculus**. Нажмите **OK**.

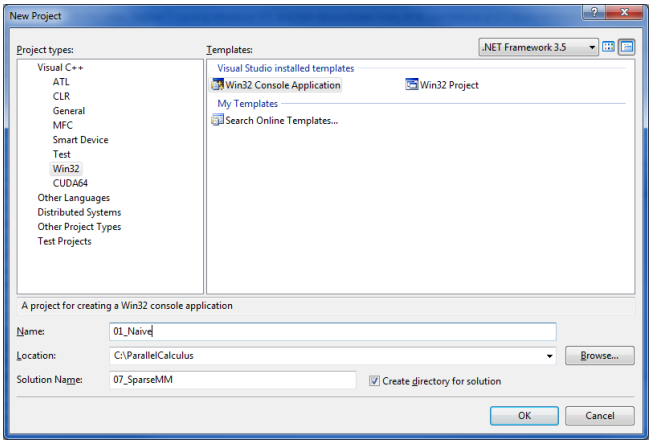

Рис. 11. Создание решения для лабораторной работы

- В диалоговом окне **Win32 Application Wizard** нажмите **Next** (или выберите **Application Settings** в дереве слева) и установите флаг **Empty Project**. Нажмите **Finish**.
- В окне **Solution Explorer** в папке **Source Files** выполните команду контекстного меню **Add→New Item…**. В дереве категорий слева выберите **Code**, в шаблонах справа – **C++ File (.cpp)**, в поле **Name** введите имя файла **main_n**. Нажмите **Add**.

В результате выполненной последовательности действий в окне редактора кода **Visual Studio** будет открыт пустой файл **main_n.cpp**.

Далее создадим заготовку функции `main()`, в которую через аргумент командной строки будем передавать порядок матриц и количество ненулевых элементов в каждой строке матрицы A. Напомним, что в § 4 мы обсуждали, что в матрице A в каждой строке будем генерировать по 50 ненулевых элементов, однако выгоднее не вводить эту константу в код, а иметь воз-

можность задать это число непосредственно в ходе проведения экспериментов.

Кроме того, используем уже разработанные функции генерации матриц, транспонирования и сравнения с функцией умножения из MKL. В результате получим следующий код.

```
// ***** Наивная версия *****

#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include "sparse.h"

const double EPSILON = 0.000001;

// argv[1] - порядок матрицы
// argv[2] - количество ненулей в строках матрицы A
int main(int argc, char *argv[])
{
  if (argc < 3)
  {
    printf("Invalid input parameters\n");
    return 1;
  }
  int N = atoi(argv[1]);
  int NZ = atoi(argv[2]);

  if ((NZ > N) || (N <= 0) || (NZ <= 0))
  {
    printf("Incorrect arguments of main\n");
    return 1;
  }

  crsMatrix A, B, BT, C;

  GenerateRegularCRS(1, N, NZ, A);
  GenerateSpecialCRS(2, N, NZ, B);

  double timeT = Transpose2(B, BT);
  double timeM, timeM1;
  Multiplicate(A, BT, C, timeM);

  crsMatrix CM;
  double diff;

  SparseMKLMult(A, B, CM, timeM1);
  SparseDiff(C, CM, diff);
  if (diff < EPSILON)
```

```
    printf("OK\n");
  else
    printf("not OK\n");
  printf("%d %d %d %d\n", A.N, A.NZ, B.NZ, C.NZ);
  printf("%.3f %.3f %.3f\n", timeT, timeM, timeM1);

  FreeMatrix(A);
  FreeMatrix(B);
  FreeMatrix(BT);
  FreeMatrix(C);

  return 0;
}
```

В представленном коде полужирным начертанием выделены фрагменты, по которым необходимы некоторые пояснения.

Во-первых, мы подключили к главному файлу проекта два вспомогательных **util.h** и **sparse.h**. Считаем, что в файле **util.h** размещены прототипы функций, описанных в §§ 4-5 (и в файле **util.cpp** их реализации соответственно), а в файле **sparse.h** – объявление структуры данных **`crsMatrix`** и прототипы функций транспонирования (две версии), а также функции умножения (реализации в **sparse_n.cpp**). Указанные 4 файла необходимо добавить в проект и поместить в них рассмотренные ранее функции.

Константа **`EPSILON`** введена для выполнения сравнения результатов умножения с использованием разработанной нами функции и эталонной версии из библиотеки MKL.

Функции **`Transpose1()`** и **`Transpose2()`** содержат реализации двух алгоритмов транспонирования, разобранных в § 6.2. В данном случае мы используем **`Transpose2()`** как более быструю и экономную в смысле памяти.

Наконец, функция **`Multiplicate()`** выполняет умножение разреженных матриц: A на B^T. Перейдем к обсуждению ее реализации.

6.4. Алгоритм умножения

Обсудим сначала общую схему реализации алгоритма умножения разреженных матриц A и B.

1. Создайте 2 вектора (**`Value`**, **`Col`**) и массив **`RowIndex`** длины N + 1 для хранения матрицы C. Размер векторов **`Value`** и **`Col`** одинаков, но пока

не известен. Можно использовать динамически распределяемые вектора из библиотеки STL[3].

2. Транспонируйте матрицу B.
3. В цикле по i от 0 до N – 1 перебирайте все строки матрицы A. Для каждого i в цикле по j от 0 до N – 1 перебирайте все строки матрицы B^T. Вычислите скалярное произведение векторов-строк A_i и B_j, пусть оно равно V. Если V отлично от нуля[4], добавьте в вектор **Value** элемент V, в вектор **Col** – элемент j. При окончании цикла по j, скорректируйте значение **RowIndex[i+1]**, записав туда текущее значение числа ненулевых элементов V.

Поскольку матрица B транспонирована до вызова функции **Multiplicate()**, при выполнении реализации этот пункт нужно пропустить.

```
const double ZERO_IN_CRS = 0.000001;

int Multiplicate(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;
  vector<int> columns;
  vector<double> values;
  vector<int> row_index;

  clock_t start = clock();
  int rowNZ;
  row_index.push_back(0);
  for (int i = 0; i < N; i++)
  {
    rowNZ = 0;
    for (int j = 0; j < N; j++)
    {
```

[3] Возможный вариант – использовать один вектор, элементом которого является структура (Value, Col). В этом случае на каждой итерации понадобится один вызов push_back() вместо двух. См. дополнительные задания.

[4] Есть два подхода: 1) хранить элемент, даже если он отличается от нуля не более чем на малую величину EPSILON или в точности равен нулю, если при вычислении скалярного произведения были обнаружены ненулевые элементы, которые перемножались с накоплением результата в сумму; 2) хранить только те элементы, которые отличаются от нуля больше чем на EPSILON. Как упоминалось ранее, оба подхода имеют право на существование.

```
      double sum = 0;

      // Считаем скалярное произведение строк A и BT
      // ...

      if (fabs(sum) > ZERO_IN_CRS)
      {
        columns.push_back(j);
        values.push_back(sum);
        rowNZ++;
      }
    }
    row_index.push_back(rowNZ + row_index[i]);
  }

  InitializeMatrix(N, columns.size(), C);

  for (unsigned int j = 0; j < columns.size(); j++)
  {
    C.Col[j] = columns[j];
    C.Value[j] = values[j];
  }
  for(int i = 0; i <= N; i++)
    C.RowIndex[i] = row_index[i];

  clock_t finish = clock();

  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

В особых пояснениях представленный код не нуждается. Отметим лишь, что мы добавляем элемент в матрицу C, только если он отличен от нуля на величину большую, чем специально введенная константа **ZERO_IN_CRS**.

Осталось научиться считать скалярное произведение строк матриц A и B^T – разреженных векторов, заданных последовательными элементами в массивах **Value** и **Col**.

Для начала реализуйте простейший вариант, который для каждого элемента строки матрицы A перебирает все элементы строки матрицы B^T, пока не будет найден элемент с таким же значением в массиве **Col** или не кончится строка. Проведите вычислительные эксперименты.

Возможная реализация скалярного произведения строк матриц A и B^T может выглядеть следующим образом.

```
  for (int k = A.RowIndex[i]; k < A.RowIndex[i + 1]; k++)
  {
    for (int l = B.RowIndex[j]; l < B.RowIndex[j + 1]; l++)
```

```
    {
      if (A.Col[k] == B.Col[l])
      {
        sum += A.Value[k] * B.Value[l];
        break;
      }
    }
  }
```

Задание.

Внесите все необходимые изменения в код. Перейдите к использованию оптимизирующего компилятора Intel C++ Compiler из пакета инструментов Intel Parallel Studio XE. Добейтесь того, чтобы код компилировался, собирался и выдавал корректные результаты.

Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 12). Напомним, что здесь и далее перемножаются матрицы 10000 x 10000, матрица A имеет 50 ненулевых элементов в каждой строке, матрица B – от 1 в первых строках до 50 в последних. Программа печатает размер матриц, статистику по количеству ненулевых элементов в A, B и C, время транспонирования (0 сек – не хватило разрешающей способности датчика), умножения текущей наивной реализации (56,878 сек) и эталонной версии из Intel MKL (0,109 сек). «OK» в первой строке вывода говорит о том, что результат прошел автоматизированный тест на корректность.

Рис. 12. Результаты работы (наивная версия)

7. Оптимизированная последовательная реализация

7.1. Алгоритмическая оптимизация. Подход 1

Итак, к настоящему моменту мы реализовали наивную версию алгоритма умножения разреженных матриц. Эксперименты показывают, что отставание по скорости от оптимизированной реализации из библиотеки MKL составляет более двух порядков.

Задействуем программные инструменты, чтобы оптимизировать наше приложение по скорости. В данной работе будем использовать некоторые компоненты пакета инструментов Intel Parallel Studio XE, а именно:

- Intel Parallel Composer XE, имеющий в составе один из лучших компиляторов для C/C++ – Intel C/C++ Compiler, библиотеку Intel MKL, содержащую высокопроизводительные оптимизированные реализации различных математических алгоритмов (базовые подпрограммы линейной алгебры, генераторы случайных чисел, БПФ, решатели СЛАУ и многое другое), библиотеку для распараллеливания Intel TBB[5].
- Intel VTune Amplifier XE – профилировщик, новое звено в цепочке профилировщиков Интел (Intel VTune Analyzer, Intel Thread Profiler, Intel Parallel Amplifier).

В работе предполагается, что слушатели уже знакомы с пакетами Parallel Studio и/или Parallel Studio XE, тем не менее, в ходе изложения будем обращать внимание на некоторые наиболее принципиальные моменты.

Так, Intel VTune Amplifier XE (далее Amplifier) позволяет получить первичные результаты профилировки в двух режимах – Lightweight Hotspots и Hotspots. Отличие состоит в том, что в первом режиме не собирается информация о стеке вызовов, что в данном случае (фактически одна основная вычислительная функция) не является для нас препятствием.

Постройте профиль приложения в режиме Lightweight Hotspots (рис. 13). Для этого выберете в контекстном меню проекта **Intel VTune Amplifier XE 2011/New Analysis**. В появившемся окне можно настроить параметры профилирования (выбор из нескольких возможностей) и свойства профилируемого проекта (по умолчанию берутся из свойств проекта в Visual Studio – убедитесь, что используется 10000 50). Выберете вариант анализа **Lightweight Hotspots** и нажмите **Start**.

Остановимся подробнее на результатах профилирования. В столбце Function приводится список тех функций, которые потребовали сколько-нибудь значимого процессорного времени. В столбце CPU Time отображено время работы каждой функции в секундах. В столбце Instruction Retired приведено количество выполненных инструкций, столбец Module отображает информацию о том, какому процессу «принадлежит» функция. Мы отфильтровали данный столбец по строке 01_naive.exe, что соответствует нашему приложению. Особый интерес вызывает столбец CPI (cycles per instruction) – количество тактов процессора, приходящееся на одну инструкцию. В документации к Amplifier указано, что для современных процессоров пиковое значение этого параметра составляет 0,25. При этом в реальной ситуации проблемы производительности, такие как плохо прогнозируемые ветвления, кеш-промахи, ограничения, связанные с пропускной способностью шины данных, высокая латентность инструкций и др. приводят к увеличению показателя CPI. Понятно, что «приемлемое значение CPI» – не более

[5] Composer включает и другие компоненты, но в данной работе они не используются.

чем условность, но в той же документации говориться о том, что CPI = 1 считается хорошим показателем для HPC-приложений из разных областей.

Lightweight Hotspots - Hotspots

Analysis Target | Analysis Type | Summary | Bottom-up | sparse_n.cpp

/Function	CPU Time	Instructions Retired	CPI	Module
Multiplicate	71.981s	395,182,000,000	0.453	01_naive.exe
GenerateRegularCRS	0.111s	376,000,000	0.713	01_naive.exe
mkl_spblas_p4m3_dmcsr_notr	0.081s	232,000,000	0.991	01_naive.exe
mkl_spblas_p4m3_dmcsradd_notr	0.056s	302,000,000	0.450	01_naive.exe
SparseDiff	0.022s	44,000,000	1.273	01_naive.exe
mkl_spblas_p4m3_dsortrow	0.014s	80,000,000	0.250	01_naive.exe
SparseMKLMult	0.009s	30,000,000	0.667	01_naive.exe
GenerateSpecialCRS	0.002s	22,000,000	0.364	01_naive.exe
std::vector<int,class std::allocator<int> >::_Insert_n	0.002s	4,000,000	1.500	01_naive.exe
Transpose2	0.002s	2,000,000	1.000	01_naive.exe
[Import thunk new]	0s	0	0.000	01_naive.exe
`eh vector constructor iterator'	0s	2,000,000	0.000	01_naive.exe
Selected 1 row(s):	71.981s	395,182,000,000		

Рис. 13. Профиль приложения (наивная версия). Вид Bottom-up

Вооружившись этими нехитрыми знаниями, перейдем к интерпретации результатов профилирования. Так, мы получаем еще одно подтверждение уже обнаруженного вручную факта: транспонирование матрицы занимает пренебрежимо малое время, следовательно, не является кандидатом для оптимизации. Подавляющее время тратится на работу функции умножения матриц **`Miltiplicate()`**. При этом CPI = 0,453 является очень неплохим показателем, тогда как CPI для аналогичной функции MKL составляет около единицы (скорее всего, необходимо учесть данные для функции mkl_spblas_p4m3_dmcsr_notr и mkl_spblas_p4m3_dsortrow_notr). При этом наша функция работает медленнее более чем в 500 раз. На самом деле никакого противоречия нет. Да, количество тактов на одну инструкцию у нас меньше, но посмотрите, насколько больше инструкций по сравнению с MKL выполняется в нашей функции! Видимо, необходимо работать над упрощением расчетов.

Для этого попробуем понять, на что тратится основное время в функции **`Miltiplicate()`**, нажав два раза на соответствующей строчке в отчете профилировщика. Результаты[6], сформированные инструментом, показаны на рис. 14.

Окно с исходными кодами разделено на две части. Слева приводится код на языке C/C++, справа – ассемблерный листинг.

---

[6] В некоторых случаях детализация времени работы по строкам кода не отображается. Как удалось заметить, это может быть связано как с тем, что функция работает слишком мало времени, так и с тем, что Amplifier не смог найти отладочную информацию и сопоставить строки кода с результатами профилировки. В первом случае не имеет смысла проводить дальнейший анализ, а во втором можно добавить в опции компиляции параметр /debug:inline-debug-info

Рис. 14. Профиль приложения (наивная версия). Исходный код

При выделении строки кода слева автоматически выделяются соответствующие инструкции на ассемблере справа. В некоторых случаях оптимизирующий компилятор умудряется преобразовать программу до неузнаваемости, в результате чего сопоставление строки кода на языке программирования высокого уровня и ассемблерных инструкций может быть затруднено. Для тех, кто понимает ассемблер, правая часть выдачи профилировщика может дать дополнительную ценную информацию о том, что на самом деле происходит. Теперь обратим внимание на результаты профилировки. Каждой строке программы на C/C++, равно как и каждой инструкции на ассемблере, поставлено в соответствие два параметра: время работы и количество выполненных инструкций – весьма полезная информация, которой часто оказывается достаточно для первоначального изучения. Сразу скажем, что Amplifier позволяет собирать и другие важные данные. В работе мы увидим применение некоторых других режимов профилировки. Понимание многих из них требует высокой квалификации в области низкоуровневого программирования и архитектур ЭВМ. В работе в основном мы будем ориентироваться на базовые знания, присутствующие у студентов, аспирантов, прикладных программистов.

Изучим полученные результаты. Как следует из профиля, основное время тратится в строках 131 и 133. При этом собственно вычисления в строке 135 занимают пренебрежимо малое время. Таким образом, программа преимущественно вхолостую проверяет, совпали ли номера столбцов у элементов перемножаемых векторов. При этом возникает огромное количество выполненных инструкций сравнения, что не способствует быстрой работе.

Таким образом, необходимо ускорить сопоставление элементов при умножении строк матриц A и B^T, представляющих собой разреженные вектора. Попробуем понять, как это можно сделать. Заметим, что каждый умножаемый вектор является упорядоченным в соответствии с выбранной структурой хранения матрицы. Это означает, что может быть применен алгоритм, весьма похожий на слияние двух отсортированных массивов в один с сохранением упорядочивания. Алгоритм выглядит следующим образом:

1. Встать на начало обоих векторов (**`ks =`** …, **`ls =`** …).
2. Сравнить текущие элементы **`A.Col[ks]`** и **`B.Col[ls]`**. Если значения совпадают, накопить в сумму произведение **`A.Value[ks] * B.Value[ls]`** и увеличить оба индекса, в противном случае – увеличить один из индексов, в зависимости от того, какое значение больше (например, **`A.Col[ks] > B.Col[ls]`** → **`ls++`**).

Шаг 2 выполняется до тех пор, пока не кончатся элементы хотя бы в одном из векторов.

Выполним программную реализацию. С учетом подготовительной работы, описанной в предыдущих разделах, фактически необходимо лишь заменить реализацию функции **`Multiplicate()`**. Тем не менее, чтобы не смешивать реализации, создадим в рамках решения **07_SparseMM** новый проект с названием **02_Optim1**. Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_o1**.

После получения пустого файла **main_o1.cpp** скопируем в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_o1.cpp**, скопировав в него код из файла **sparse_n.cpp** проекта **01_Naive**.

Теперь изменим реализацию функции **`Multiplicate()`** в файле **sparse_o1.cpp**.

```
int Multiplicate(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
...
  clock_t start = clock();
  int NZ = 0;

  row_index.push_back(0);
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      // Умножаем строку i матрицы A и столбец j матрицы B
      double sum = 0;
      int ks = A.RowIndex[i];
      int ls = B.RowIndex[j];
      int kf = A.RowIndex[i + 1] - 1;
      int lf = B.RowIndex[j + 1] - 1;
      while ((ks <= kf) && (ls <= lf))
      {
        if (A.Col[ks] < B.Col[ls])
          ks++;
        else
          if (A.Col[ks] > B.Col[ls])
            ls++;
          else
          {
            sum += A.Value[ks] * B.Value[ls];
            ks++;
            ls++;
          }
      }
```

```
      if (fabs(sum) > ZERO_IN_CRS)
      {
        columns.push_back(j);
        values.push_back(sum);
        NZ++;
      }
    }
    row_index.push_back(NZ);
  }

  InitializeMatrix(N, NZ, C);
...

  return 0;
}
```

Большая часть кода осталась прежней. Новые строки выделены полужирным начертанием. Фрагменты кода до первого замера времени и после функции **`InitializeMatrix()`** заменены многоточием, поскольку полностью совпадают с кодом из предыдущей версии.

Проведите вычислительные эксперименты для новой реализации. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 15).

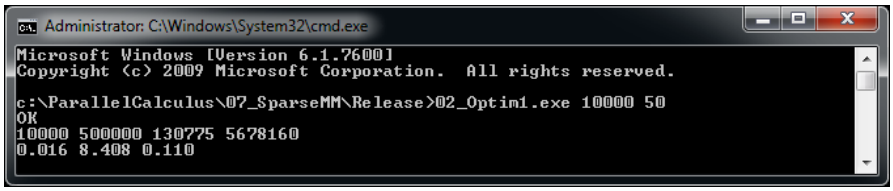

Рис. 15. Результаты работы (алг. опт., подход 1)

Результат достаточно нагляден. Ускорение по отношению к предыдущей версии составило почти 7 раз.

7.2. Алгоритмическая оптимизация. Подход 2

Итак, получен неплохой прирост производительности в результате достаточно очевидной оптимизации. Нам удалось эффективно использовать тот факт, что структура данных для хранения матрицы является упорядоченной, в противном случае напрямую использовать алгоритм слияния нельзя. Заметим, что функция умножения из библиотеки Intel MKL принимает в качестве параметра информацию о том, являются ли исходные матрицы упорядоченными. Исходя из документации, в неупорядоченном случае внутри функции сначала производится сортировка. MKL – коммерческая библиотека, поэтому алгоритм является закрытым. В литературе описаны

достаточно эффективные методы, которые, к примеру, состоят в двукратном транспонировании исходной неупорядоченной матрицы при помощи рассмотренного выше алгоритма. В результате получается упорядоченная матрица.

Попробуем улучшить имеющуюся реализацию. Построим профиль при помощи Intel VTune Amplifier XE (рис. 16).

Lightweight Hotspots - Hotspots Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

/Function	CPU Time	Instructions Retired	CPI
Multiplicate	8.413s	35,646,000,000	0.588
mkl_spblas_p4m3_dmcsr_notr	0.105s	232,000,000	1.112
main	0.097s	298,000,000	0.799
mkl_spblas_p4m3_dmcsradd_notr	0.068s	294,000,000	0.565
std::::push_back	0.026s	80,000,000	0.650
SparseDiff	0.023s	46,000,000	1.087
std::::push_back	0.018s	66,000,000	1.091
mkl_spblas_p4m3_dsortrow	0.011s	78,000,000	0.410
SparseMKLMult	0.008s	10,000,000	1.800
GenerateSpecialCRS	0.002s	8,000,000	0.500
std::::_Insert_n	0.001s	2,000,000	3.000
`eh vector constructor iterator'	0s	2,000,000	0.000
Selected 1 row(s):	8.413s	35,646,000,000	

Рис. 16. Профиль приложения (алг. опт., подход 1). Вид Bottom-up

Заметим, что число выполненных инструкций в функциях MKL и время работы несколько поменялось, что свидетельствует о наличии неизбежной погрешности при измерениях. Погрешность может быть обусловлена как способом измерения, так и несколько разным состоянием программно-аппаратного окружения, в котором происходят запуски, даже с учетом того, что исходные данные постоянны.

Обратимся к результатам профилировки. По-прежнему, практически все время тратится на умножение в функции **`Multiplicate()`**. Количество выполненных инструкций в этой функции, существенно уменьшилось, но все еще осталось очень большим по сравнению с аналогом из MKL. Посмотрим на подробный профиль функции **`Multiplicate()`** (рис. 17).

Lightweight Hotspots - Hotspots

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | sparse_o...

Line	Source	CPU Time	Instructions Retired
121			
122	row_index.push_back(0);		
123	for (int i = 0; i < N; i++)		
124	{		
125	for (int j = 0; j < N; j++)	0.091s	344,000,000
126	{		
127	// Умножаем строку i матрицы A и столбец j матрицы B		
128	double sum = 0;	0.001s	2,000,000
129	int ks = A.RowIndex[i]; int ls = B.RowIndex[j];	0.090s	286,000,000
130	int kf = A.RowIndex[i + 1] - 1; int lf = B.RowIndex[j + 1] - 1	0.134s	648,000,000
131	while ((ks <= kf) && (ls <= lf))	0.724s	2,182,000,000
132	{		
133	if (A.Col[ks] < B.Col[ls])	2.366s	9,088,000,000
134	ks++;	1.922s	8,704,000,000
135	else		
136	if (A.Col[ks] > B.Col[ls])	1.788s	8,110,000,000
137	ls++;	1.007s	5,404,000,000
138	else		
139	{		
140	sum += A.Value[ks] * B.Value[ls];	0.032s	138,000,000
141	ks++;	0.006s	24,000,000
142	ls++;	0.047s	92,000,000
143	}		
144	}		
145	if (fabs(sum) > ZERO_IN_CRS)	0.092s	352,000,000
146	{		
	Selected 1 row(s):		

Filter: 97.4% is shown | Module: [97.4%] 02_optim1.e: | Thread: [All] | Process: [All]

Рис. 17. Профиль приложения (алг. опт., подход 1). Исходный код

Нетрудно видеть, что основное время по-прежнему тратится на слияние векторов. Причем, доля «реальных вычислений» все еще пренебрежимо мала, тогда как проверки во внутреннем цикле занимают практически все время. В данном случае выигрыш в производительности достигнут за счет того, что мы заменили квадратичный по сложности алгоритм умножения разреженных векторов на его линейный по сложности аналог. При этом реализация алгоритма не очень хорошо соответствует особенностям архитектуры: действительно, факт того, какой из элементов (A.Col[ks] или B.Col[ls]) окажется больше, невозможно прогнозировать. Также сложно с проверками условия в цикле while, поскольку количество итераций зачастую оказывается небольшим! В итоге предсказатель ветвлений часто ошибается. Для него должно стать очевидным лишь то, что указанные элементы крайне редко бывают равны. Наличие ошибок в предсказании условных переходов приводит к неэффективной работе.

General Exploration - Hardware Issues

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

/Function	PMU Event Count							
	CPU_CLK_UNHALTED.THREAD	INST_RETIRED.ANY	CPI	Retire Stalls	LLC Miss	Instruction Starvation	Branch Mispredict	Execution Stalls
Multiplicate	28,000,000,000	45,596,000,000	0.614	0.260	0.000	0.149	0.065	0.057
mkl_spblas_p4m3_dmcsr_notr	492,000,000	452,000,000	1.088	1.691	0.106	0.000	0.000	0.000
mkl_spblas_p4m3_dmcsradd_notr	272,000,000	528,000,000	0.515	0.000	0.000	0.000	0.000	0.000
main	236,000,000	274,000,000	0.861	0.000	0.000	0.000	0.000	0.992
SparseDiff	102,000,000	64,000,000	1.594	4.588	0.000	0.000	0.000	0.000
mkl_spblas_p4m3_dsortrow	96,000,000	196,000,000	0.490	0.000	0.000	0.000	0.000	0.000
std::::push_back	80,000,000	78,000,000	1.026	1.300	0.000	0.000	0.683	0.000
std::::push_back	72,000,000	68,000,000	1.059	0.000	0.000	0.000	0.054	0.000
SparseMKLMult	36,000,000	20,000,000	1.800	8.667	0.000	0.000	0.000	0.000
std::::_Insert_n	8,000,000	6,000,000	1.333	0.000	0.000	0.000	0.000	0.000
GenerateSpecialCRS	6,000,000	8,000,000	0.750	0.000	0.000	0.000	0.000	0.000
[Import thunk new]	0	2,000,000	0.000	0.000	0.000	0.000	0.000	0.000
Selected 1 row(s):	28,000,000,000	45,596,000,000						

Рис. 18. Профиль приложения (алг. опт., подход 1). Режим General Exploration

Желающие могут убедиться в выводах, запустив профилировку в режиме General Exploration (столбец Branch Mispredict в результатах анализа, рис. 18). Выделенное красным значение 0,065 – посредственный результат (желательно значение, близкое к нулю). При двойном нажатии на строку с именем функции можно вновь увидеть исходный код. Обратите внимание на числа в столбце напротив оператора цикла while и условных операторов if, которые мы обсудили выше.

Итак, мы существенно ускорили приложение, но количество выполняемых инструкций остается слишком большим, а реализация плохо соответствует архитектуре. Обе грани общей проблемы сводятся к поиску соответствующих элементов разреженных векторов. Нельзя ли предложить более эффективный вариант? Обратимся к литературе за поиском идей для оптимизации. Так, например, в книге [2] предлагается остроумный вариант вычисления скалярного произведения разреженных векторов.

Рассмотрим следующий алгоритм:

1. Создадим дополнительный целочисленный массив X длины N. Инициализируем его числом -1. Обнулим вещественную переменную S.
2. Просмотрим в цикле все ненулевые элементы первого вектора V1. Пусть такой элемент с порядковым номером *i* расположен в столбце с номером **`j = V1.Col[i]`**. В этом случае запишем *i* в *j*-ю ячейку дополнительного массива.
3. Просмотрим в цикле все ненулевые элементы второго вектора V2. Пусть элемент с порядковым номером *k* расположен в столбце с номером **`z = V2.Col[k]`**. Проверим значение X[z]. Если оно равно -1, в первом векторе нет соответствующего элемента, т.е. умножение выполнять не нужно. Иначе умножаем **`V2.Value[k]`** и **`V1.Value[X[z]]`** и накапливаем в S.

Выполним программную реализацию.

Как и в предыдущем разделе создадим в рамках решения **07_SparseMM** новый проект с названием **03_Optim2**. Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_o2**.

После получения пустого файла **main_o2.cpp** скопируем в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_o2.cpp**, скопировав в него код из файла **sparse_n.cpp** проекта **01_Naive**.

Теперь изменим реализацию функции **`Multiplicate()`** в файле **sparse_o2.cpp**.

```
int Multiplicate(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;
  int i, j, k;

  vector<int> columns;
  vector<double> values;
  vector<int> row_index;

  clock_t start = clock();
  int NZ = 0;

  int *temp = new int[N];
  row_index.push_back(0);

  for (i = 0; i < N; i++)
  {
    // i-я строка матрицы A
    // Обнуляем массив указателей на элементы
    memset(temp, -1, N * sizeof(int));
    // Идем по ненулевым элементам строки и заполняем
    // массив указателей
    int ind1 = A.RowIndex[i], ind2 = A.RowIndex[i + 1];
    for (j = ind1; j < ind2; j++)
    {
      int col = A.Col[j];
      temp[col] = j; // Значит, что a[i, НОМЕР] лежит
                     // в ячейке массива Value с номером
                     // temp[НОМЕР]
    }
    // Построен индекс строки i матрицы A
    // Теперь необходимо умножить ее на каждую из строк
    // матрицы BT
    for (j = 0; j < N; j++)
    {
      // j-я строка матрицы B
      double sum = 0;
      int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];
      // Все ненулевые элементы строки j матрицы B
      for (k = ind3; k < ind4; k++)
      {
        int bcol = B.Col[k];
        int aind = temp[bcol];
```

```
        if (aind != -1)
          sum += A.Value[aind] * B.Value[k];
      }
      if (fabs(sum) > ZERO_IN_CRS)
      {
        columns.push_back(j);
        values.push_back(sum);
        NZ++;
      }
    }
    row_index.push_back(NZ);
  }

  InitializeMatrix(N, NZ, C);

  for (j = 0; j < NZ; j++)
  {
    C.Col[j] = columns[j];
    C.Value[j] = values[j];
  }
  for(i = 0; i <= N; i++)
    C.RowIndex[i] = row_index[i];

  delete [] temp;

  clock_t finish = clock();
  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

Код функции изменился достаточно сильно, поэтому мы привели его целиком, как и прежде выделив новое полужирным начертанием.

Необходимо сделать отдельное замечание по поводу использованной нами инициализации массива **temp** с помощью функции **memset()**. Вообще говоря, то, что мы сделали, в некотором смысле является «трюком» и работает по «счастливой» случайности. Согласно описанию функция **memset()** инициализирует указанное третьим параметром число символов, расположенных в памяти по адресу, начиная с первого параметра. Инициализация каждого символа выполняется значением байта, переданным во втором параметре. Типичным использованием данной функции является быстрое обнуление элементов некоторого массива. В случае же попытки инициализировать все элементы массива, например, целого типа (скажем, для определенности типа **int**), другим значением, мы сталкиваемся с проблемой. Если попытаться указать в качестве значения второго параметра функции **memset()**, например, 2, ожидая, что в результате в

каждом элементе и окажется это число, можно сильно удивиться впоследствии. В действительности, если представить содержимое байта со значением 2 (00000010) и «размножить» его 4 раза (по размеру типа **`int`**), нетрудно понять, что получившееся целое число вовсе не будет равняться двум. В нашем случае «трюк» работает в силу представления числа -1 в дополнительном коде, которое состоит из одних единиц, так что после четырехкратного «размножения» байта мы по-прежнему получаем число -1 в дополнительном коде.

Проведите вычислительные эксперименты для новой реализации. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 19).

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

c:\ParallelCalculus\07_SparseMM\Release>03_Optim2.exe 10000 50
OK
10000 500000 130775 5678160
0.000 2.512 0.109
```

Рис. 19. Результаты работы (подход 2)

И снова мы имеем неплохое (более чем трехкратное) ускорение по отношению к предыдущей реализации. Теперь отставание от MKL составляет чуть больше одного порядка. А по сравнению с исходной наивной версией мы ускорили реализацию более чем в 22 раза.

Приведем профили, полученные в результате работы Amplifier (Рис. 20, рис. 21).

Lightweight Hotspots - Hotspots — Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

/Function	CPU Time	Instructions Retired	CPI
Multiplicate	2.444s	11,428,000,000	0.532
mkl_spblas_p4m3_dmcsr_notr	0.102s	234,000,000	1.137
main	0.096s	294,000,000	0.830
mkl_spblas_p4m3_dmcsradd_notr	0.072s	296,000,000	0.628
SparseDiff	0.032s	46,000,000	1.783
std::::push_back	0.026s	80,000,000	0.925
std::::push_back	0.023s	76,000,000	0.947
mkl_spblas_p4m3_dsortrow	0.012s	86,000,000	0.419
__intel_memset	0.011s	36,000,000	1.000
SparseMKLMult	0.008s	10,000,000	1.800
_intel_fast_memset	0.001s	0	0.000
std::::_Insert_n	0.001s	6,000,000	0.333
GenerateSpecialCRS	0.001s	2,000,000	2.000
mkl_spblas_mkl_dcsradd	0s	2,000,000	0.000
[Import thunk new]	0s	2,000,000	0.000
Selected 1 row(s):	2.444s	11,428,000,000	

Рис. 20. Профиль приложения (алг. опт., подход 2). Вид Bottom-up

Обращает на себя внимание существенное уменьшение числа инструкций, которое все еще остается большим, и незначительное улучшение параметра CPI. В исходном коде можно заметить уже известные проблемы произ-

водительности – проверки в строках 148 и 152, а также необычное замедление в строке 150.

Lightweight Hotspots - Hotspots

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | sparse_o...

Line	Source	CPU Time	Instructions Retired
140	// Построен индекс строки i матрицы A		
141	// Теперь необходимо умножить ее на каждую из строк матрицы BT		
142	for (j = 0; j < N; j++)	88.722ms	290,000,000
143	{		
144	// j-я строка матрицы B		
145	volatile double sum = 0;	6.015ms	40,000,000
146	int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];	133.835ms	686,000,000
147	// Все ненулевые элементы строки j матрицы B		
148	for (k = ind3; k < ind4; k++)	561.654ms	3,374,000,000
149	{		
150	int bcol = B.Col[k];	557.895ms	2,040,000,000
151	int aind = temp[bcol];	96.241ms	410,000,000
152	if (aind != -1)	751.880ms	3,708,000,000
153	sum += A.Value[aind] * B.Value[k];	66.165ms	298,000,000
154	}		
155	if (fabs(sum) > ZERO_IN_CRS)	109.023ms	404,000,000
156	{		
157	columns.push_back(j);		
158	values.push_back(sum);		
159	NZ++;		
160	}		
161	}		
162	row_index.push_back(NZ);		2,000,000
163	}		
164			
165	InitializeMatrix(N, NZ, C);		

Selected 1 row(s):

Filter: 92.3% is shown | Module: [92.3%] 03_optim2.e: | Thread: [All] | Process: [All]

Рис. 21. Профиль приложения (алг. опт., подход 2). Исходный код

На этом с дальнейшей оптимизацией последовательного алгоритма мы остановимся. Желающие могут попытать счастья как с алгоритмической (реализация принципиально другого подхода), так и с программной точки зрения. Обсудим вопрос зависимости времени работы от исходных данных.

7.3. Программная оптимизация. Последние штрихи

До сих пор все эксперименты, которые мы проводили, выполнялись в условиях, когда матрица A имеет «регулярную» структуру – в каждой ее строке по 50 ненулевых элементов. Проведем теперь эксперимент, когда «регулярной» является матрица B, то есть после генерации переставим их местами в функции умножения **`Multiplicate()`**.

```
crsMatrix A, B, BT, C;

GenerateRegularCRS(1, N, NZ, A);
GenerateSpecialCRS(2, N, NZ, B);
```

```
double timeT = Transpose2(A, BT);
double timeM, timeM1;
Multiplicate(B, BT, C, timeM);

crsMatrix CM;
double diff;

SparseMKLMult(B, A, CM, timeM1);
SparseDiff(C, CM, diff);
```

Выше представлен фрагмент кода функции **`main()`**, в котором изменены три строки (выделены полужирным начертанием). Транспонируется матрица A вместо B, а также изменен порядок аргументов в функциях **`Multiplate()`** и **`SparseMKLMult()`**.

Для проведения эксперимента создадим в рамках решения **07_SparseMM** новый проект с названием **04_Optim3** (как мы увидим в дальнейшем, одной перестановкой матриц дело не ограничится). Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_o3**.

После получения пустого файла **main_o3.cpp** скопируйте в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_o3.cpp**, добавив в него код из файла **sparse_o2.cpp** проекта **03_Optim2**.

Внесите описанные выше изменения в код функции **`main()`** в файле **main_o3.cpp** и проведите вычислительные эксперименты.

На тестовой инфраструктуре, описанной в § 1.3, авторы получили следующие результаты (рис. 22).

Рис. 22. Результаты работы (оптимизированная версия 2, умножается матрица B на матрицу A)

На первый взгляд результаты выглядят обескураживающе. Во-первых, время умножения нашим алгоритмом выросло с двух с половиной секунд до восьми с половиной, во-вторых, кардинально, почти в 40 раз ухудшилось время работы функции из библиотеки MKL. Учитывая закрытый код MKL, мы не можем сказать, чем вызвана настолько большая разница, а вот

почему наша реализация существенно замедлилась от перестановки матриц местами, попробуем разобраться.

Соберите профиль при помощи Amplifier (рис. 23).

Lightweight Hotspots - Hotspots

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | main_o3.cpp | sparse_o...

Line	Source	CPU Time	Instructions Retired
140	// Построен индекс строки i матрицы A		
141	// Теперь необходимо умножить ее на каждую из строк матрицы BT		
142	for (j = 0; j < N; j++)	0.120s	588,000,000
143	{		
144	// j-я строка матрицы B		
145	double sum = 0;	0.021s	62,000,000
146	int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];	0.104s	570,000,000
147	// Все ненулевые элементы строки j матрицы B		
148	for (k = ind3; k < ind4; k++)	3.083s	15,358,000,000
149	{		
150	int bcol = B.Col[k];	3.005s	15,722,000,000
151	int aind = temp[bcol];	0.476s	2,258,000,000
152	if (aind != -1)	5.114s	25,980,000,000
153	sum += A.Value[aind] * B.Value[k];	0.347s	664,000,000
154	}		
155	if (fabs(sum) > ZERO_IN_CRS)	0.278s	1,176,000,000
156	{		
157	columns.push_back(j);		
158	values.push_back(sum);		
159	NZ++;		
160	}		
161	}		
162	row_index.push_back(NZ);		
163	}		
164			
165	InitializeMatrix(N, NZ, C);		

Selected 1 row(s):

Filter: 98.2% is shown | Module: [98.2%] 04_optim3.e: | Thread: [All] | Process: [All]

Рис. 23. Профиль приложения (оптимизированная версия 2, умножается матрица B на матрицу A). Исходный код

Обратите внимание на гигантское количество выполненных инструкций в строках 148, 150, 152.

Посмотрите еще раз на код функции **`Multiplicate()`** в файле **sparse_o3.cpp** проекта **04_Optim3**. Первое, что в ней делается, – для каждой строки матрицы A строится индексный массив. Затем строка матрицы A умножается на все строки матрицы B с использованием индексного массива. В случае, когда матрица A имеет «регулярную» структуру, индексный массив для каждой ее строки содержит по 50 элементов, отличных от -1. А внутренний цикл по *k*, в котором перебираются элементы очередной *j*-ой строки матрицы B, состоит из небольшого числа итераций значительную часть времени. Напротив, когда матрицы меняются местами, внутренний цикл по *k* начинает содержать по 50 итераций во всех случаях. Конеч-

но, при этом затраты на построение индекса для матрицы A с нерегулярной структурой снижаются, однако это не оказывает существенного влияния на результат.

Подумаем, что можно сделать, чтобы исправить ситуацию.

Используем известный факт, что $(AB)^T = B^T A^T$. Учитывая, что транспонирование разреженной матрицы выполняется пренебрежимо малое время по сравнению с умножением, можно вместо умножения «нерегулярной» матрицы A на «регулярную» матрицу B, умножать их транспонированные версии в обратном порядке и транспонировать результат.

Напишем дополнительную функцию **`Multiply()`**, реализующую эту идею. Для обобщения, предварительно выясним, какая ситуация на самом деле имеет место, то есть, какая из матриц имеет «регулярную» структуру. Сделать это в нашем случае довольно просто, поскольку в «регулярной» матрице заведомо большее число ненулевых элементов. Получим следующий код.

```
int Multiply(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;

  clock_t start = clock();

  if (A.NZ > B.NZ)
  {
    Multiplicate(A, B, C, time);
  }
  else
  {
    crsMatrix CT;

    Multiplicate(B, A, C, time);

    Transpose2(C, CT);

    int i;
    for (i = 0; i < CT.NZ; i++)
    {
      C.Col[i] = CT.Col[i];
      C.Value[i] = CT.Value[i];
    }
    for(i = 0; i <= CT.N; i++)
      C.RowIndex[i] = CT.RowIndex[i];
```

```
    FreeMatrix(CT);
  }

  clock_t finish = clock();
  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

Отметим, что в коде мы не вызывали функции транспонирования ни для матрицы A, ни для матрицы B. В правильности такого варианта нетрудно убедиться, учитывая, что функция **`Multiplicate()`** возвращает матрицу C как результат операции AB^T. Таким образом, чтобы получить матрицу C^T, необходимо (в силу $C^T=(AB^T)^T=(B^T)^TA^T=BA^T$) вызвать **`Multiplicate()`**, передав в нее транспонированную матрицу A и нетранспонированную матрицу B.

Проведите вычислительные эксперименты для полученной реализации. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 24).

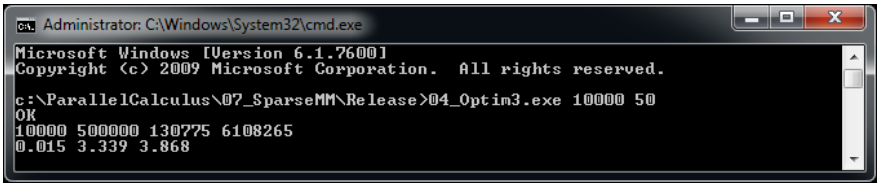

Рис. 24. Результаты работы (оптимизированная версия 3, умножается матрица B на матрицу A)

Как видим, в результате нам удалось не только существенно сократить время умножения (и почти догнать исходный вариант, см. рис. 19), но и выиграть у реализации из библиотеки MKL. Конечно, не нужно делать из этого факта далеко идущих выводов. В отличие от реализации из Intel MKL наша версия рассчитана на достаточно специфические входные данные. Используемая для определения способа умножения эвристика является элементарной, и далеко не всегда будет давать хорошие результаты. Легко проверить, что в большинстве случаев наша текущая реализация будет значительно уступать MKL по скорости.

Для полноты картины приведем профиль ускоренной реализации (рис. 25).

Отметим, что соотношение затрат времени между различными строками программы в целом не претерпело существенных изменений, но сократилось количество выполненных инструкций, что и дало выигрыш в скорости.

Lightweight Hotspots - Hotspots

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | sparse_o...

Line	Source	CPU Time	Instructions Retired
140	// Построен индекс строки i матрицы A		
141	// Теперь необходимо умножить ее на каждую из строк матрицы BT		
142	for (j = 0; j < N; j++)	89.474ms	476,000,000
143	{		
144	// j-я строка матрицы B		
145	double sum = 0;	3.008ms	14,000,000
146	int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];	44.361ms	268,000,000
147	// Все ненулевые элементы строки j матрицы B		
148	for (k = ind3; k < ind4; k++)	594.737ms	3,456,000,000
149	{		
150	int bcol = B.Col[k];	493.985ms	2,540,000,000
151	int aind = temp[bcol];	136.842ms	690,000,000
152	if (aind != -1)	801.504ms	4,080,000,000
153	sum += A.Value[aind] * B.Value[k];	79.699ms	364,000,000
154	}		
155	if (fabs(sum) > ZERO_IN_CRS)	79.699ms	340,000,000
156	{		
157	columns.push_back(j);		
158	values.push_back(sum);		
159	NZ++;		
160	}		
161	}		
162	row_index.push_back(NZ);		
163	}		
164			
165	InitializeMatrix(N, NZ, C);		
	Selected 1 row(s):		

No filters are applied. Module: [All] Thread: [All] Process: [All]

Рис. 25. Профиль приложения (умножается матрица B на матрицу A). Исходный код

Почему мы посчитали необходимым включить этот параграф в лабораторную работу? Дело совсем не в том, что в результате долгих поисков был найден вариант исходных данных, на которых наша реализация обгоняет MKL (разработкой и оптимизацией которой специалисты занимаются на протяжении многих лет). Мы хотели подчеркнуть важный момент: знание особенностей исходных данных часто позволяет выбрать подходящую структуру данных и сконструировать алгоритм, значительно опережающий алгоритмы, рассчитанные на «общий случай».

8. Двухфазная последовательная реализация

8.1. Идея разделения алгоритмов разреженной алгебры на две фазы

Из практики работы с разреженными матрицами известно, что в ряде задач требуется многократно выполнять однотипные операции над матрицами, имеющими одинаковые портреты, но разные значения элементов. Предста-

вим, что требуется 100 раз выполнить умножение матриц A и B, причем в каждом умножении портреты A и B одинаковы. Можно ли адаптировать имеющуюся реализацию к этому случаю? Нетрудно видеть, что при сохранении портрета матриц A и B сохраняется также портрет результирующей матрицы C[7]. Это означает, что мы можем разделить вычисления на 2 фазы: *символическую* и *численную* (см., например, [2]). В рамках символической части, которая выполняется однократно, производится построение портрета результата (относится не только к умножению матриц, но и к другим алгоритмам разреженной алгебры), а численная часть состоит в заполнении портрета значениями. При многократных однотипных операциях с одинаковыми портретами достигается огромный выигрыш производительности по сравнению с обычным подходом[8]. Иногда разделение на символическую и численную фазу полезно и из других соображений. Так, символическая часть вычислений решает все вопросы, связанные с выделением памяти, что часто делает программу понятнее, а иногда и быстрее.

8.2. Последовательный двухфазный алгоритм

8.2.1. Описание алгоритма

Попробуем продумать реализацию идеи с разделением вычислений на две фазы – символическую и численную.

Задача символической фазы – построить портрет матрицы C. Это можно сделать, исключив вычисление скалярного произведения из предыдущего алгоритма, оставив остальное без изменений. Так, после транспонирования матрицы B (честно говоря, для этой фазы достаточно символического транспонирования, массив **`Value`** будет не нужен, но транспонирование работает настолько быстро, что пока нет смысла делить его на фазы) необходимо при каждом «умножении» векторов (строк матриц A и B^T) определить, есть ли хотя бы одна пара элементов, находящихся в одном и том же столбце. Если такая пара найдется, нужно учесть это при формировании портрета матрицы C.

---

[7] Теперь настало время расставить точки над *i* в вопросе, который мы несколько раз поднимали ранее. На самом деле, сохранение портрета вынуждает нас в некоторых случаях хранить нули в структуре матрицы. Это происходит, когда в результате выполнения математических операций получается нуль (с некоторой точностью EPSILON). Однако это происходит не так часто, а выигрыш от такого решения велик – есть возможность выделить построение портрета в отдельную фазу, после чего многократно его заполнять разными значениями.

[8] Один из примеров – итерационные методы оптимизации. На каждой итерации требуется решать систему линейных уравнений, нередко – с разреженной матрицей. Портрет матрицы часто постоянен, от итерации к итерации меняются конкретные значения и правая часть. Для решения системы часть вычислений можно провести в рамках символьной фазы – построить портреты, впоследствии многократно заполняя их значениями.

Реализуем символическую фазу алгоритма, описанного выше. Учтем при этом, что значения вычислять не нужно, достаточно установить факт наличия соответствующих элементов.

8.2.2. Программная реализация

Как и ранее создадим в рамках решения **07_SparseMM** новый проект с названием **05_Two_phase**. Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_tp**.

После получения пустого файла **main_tp.cpp** скопируем в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_tp.cpp**, скопировав в него код из файла **sparse_o2.cpp** проекта **03_Optim2**.

Прежде всего, добавим в файл **sparse.h** прототипы двух новых функций для выполнения символической и численной фаз умножения.

```
int SymbolicMult(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time);
int NumericMult(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time);
```

Теперь внесем изменения в код функции **`main()`**. Фрагмент кода до вызова функции **`Transpose2()`** заменен многоточием, поскольку в нем изменений не потребуется.

```
// ***** Двухфазная версия *****

#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include "sparse.h"

const double EPSILON = 0.000001;

// argv[1] - порядок матрицы
// argv[2] - количество ненулей в строках матрицы A
int main(int argc, char *argv[])
{
...

  double timeT = Transpose2(B, BT);
  double timeS, timeM, timeM1;
  SymbolicMult(A, BT, C, timeS);
  NumericMult(A, BT, C, timeM);

  crsMatrix CM;
```

```
  double diff;

  SparseMKLMult(A, B, CM, timeM1);
  SparseDiff(C, CM, diff);
  if (diff < EPSILON)
    printf("OK\n");
  else
    printf("not OK\n");
  printf("%d %d %d %d\n", A.N, A.NZ, B.NZ, C.NZ);
  printf("%.3f %.3f %.3f\n", timeT, timeS, timeM, timeM1);

  FreeMatrix(A);
  FreeMatrix(B);
  FreeMatrix(BT);
  FreeMatrix(C);

  return 0;
}
```

Фактически, мы заменили вызов функции **`Multiplicate()`** на два вызова, выполняющих расчет по двухфазной схеме и добавили еще одну переменную для замера времени.

Реализация символической фазы будет чрезвычайно простой. Скопируем в функцию **`SymbolicMult()`** код функции **`Multiplicate()`** из проекта **03_Optim2** и закомментируем (удалим) строки, связанные с созданием и обработкой массива **`Value`**. Также необходимо избавиться от выполнения расчетов, что, конечно, приведет к некоторому изменению портрета матрицы C, по сравнению с предыдущими реализациями, но является верным, поскольку символическая часть не должна зависеть от конкретных значений элементов матриц A и B.

```
//  vector<double> values;
...
    for (j = 0; j < N; j++)
    {
      // j-я строка матрицы B
//      double sum = 0;
      int IsFound = 0;
      int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];
      // Все ненулевые элементы строки j матрицы B
      for (k = ind3; k < ind4; k++)
      {
        int bcol = B.Col[k];
        int aind = temp[bcol];
        if (aind != -1)
        {
//          sum += A.Value[aind] * B.Value[k];
          IsFound = 1;
```

```
          break;
        }
      }
//      if (fabs(sum) > ZERO_IN_CRS)
      if (IsFound)
      {
        columns.push_back(j);
//        values.push_back(sum);
        NZ++;
      }
...
  for (j = 0; j < NZ; j++)
  {
    C.Col[j] = columns[j];
//    C.Value[j] = values[j];
  }
```

В представленном коде полужирным начертанием выделены новые строки, работа с вектором **`values`** и переменной **`sum`** удалена.

Отметим, что представленный вариант не слишком хорош с точки зрения алгоритмизации и рассмотрен нами лишь в силу очень простого перехода от проекта **03_Optim2**. Улучшить код можно, например, следующим образом:

```
  ...
  int aind = -1; k = ind3;
  while ((aind == -1) && (k < ind4))
  {
    int bcol = B.Col[k];
    int aind = temp[bcol];
    k++;
  }
  if (aind != -1)
    ...
```

Выполнить соответствующую реализацию предоставляем читателю самостоятельно.

Перейдем к обсуждению реализации численной фазы.

Имея сформированный портрет матрицы C, можно поступить очень просто – реализация функции **`NumericMult()`** может состоять в проходе по матрице C и вычислении каждого ее элемента как скалярного произведения строки A на строку B^T. Весь вопрос в том, как именно считать скалярное произведение. Используем для этого уже написанные нами фрагменты кода из проектов **01_Naive** и **02_Optim1**. В первом случае получим:

```
int NumericMult(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
```

```
    return 1;

  int N = A.N;

  clock_t start = clock();
  memset(C.Value, 0, C.NZ * sizeof(double));

  int index = 0;
  for (i = 0; i < N; i++)
  {
    for (j = C.RowIndex[i]; j < C.RowIndex[i + 1]; j++)
    {
      for (k = A.RowIndex[i]; k < A.RowIndex[i + 1]; k++)
      {
        int colJ = C.Col[j];
        for (l = B.RowIndex[colJ];
             l < B.RowIndex[colJ + 1]; l++)
        {
          if (A.Col[k] == B.Col[l])
          {
            C.Value[index] += A.Value[k] * B.Value[l];
            break;
          }
        }
      }
      index++;
    }
  }

  clock_t finish = clock();

  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

Проведите вычислительные эксперименты с первой версией функции **`NumericMult()`**. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 26).

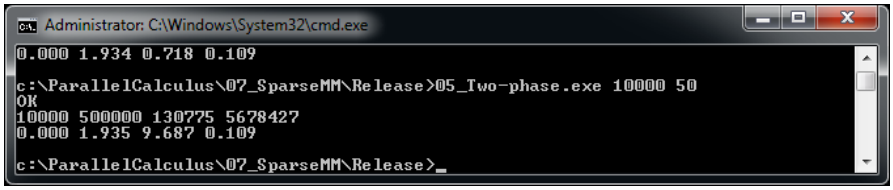

Рис. 26. Результаты работы (двухфазная версия, численная фаза, версия 1)

Как видим, символическая фаза работает быстрее, чем функция **`Multiplicate()`** из проекта **03_Optim2** (что и следовало ожидать, учитывая, что они отличаются работой с массивом **`Value`**, точнее ее отсутствием в функции **`SymbolicMult()`**, а также досрочным завершением внутреннего цикла). Что касается численной фазы, то она работает кардинально медленнее, чем символическая, что тоже не удивительно, учитывая ее неоптимальность.

Теперь заменим скалярное произведение на вариант из проекта **02_Optim1**. Получим следующее.

```
int NumericMult(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;

  clock_t start = clock();
  memset(C.Value, 0, C.NZ * sizeof(double));

  int index = 0;
  for (int i = 0; i < N; i++)
  {
    for (int j = C.RowIndex[i]; j < C.RowIndex[i + 1]; j++)
    {
      // Умножаем строку i матрицы A и столбец j матрицы B
      double sum = 0;
      int colJ = C.Col[j];
      int ks = A.RowIndex[i];
      int ls = B.RowIndex[colJ];
      int kf = A.RowIndex[i + 1] - 1;
      int lf = B.RowIndex[colJ + 1] - 1;
      while ((ks <= kf) && (ls <= lf))
      {
        if (A.Col[ks] < B.Col[ls])
          ks++;
        else
          if (A.Col[ks] > B.Col[ls])
            ls++;
          else
          {
            sum += A.Value[ks] * B.Value[ls];
            ks++; ls++;
          }
      }
      C.Value[index] = sum;
      index++;
    }
```

```
  }

  clock_t finish = clock();

  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

Проведите вычислительные эксперименты со второй версией функции **`NumericMult()`**. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 27).

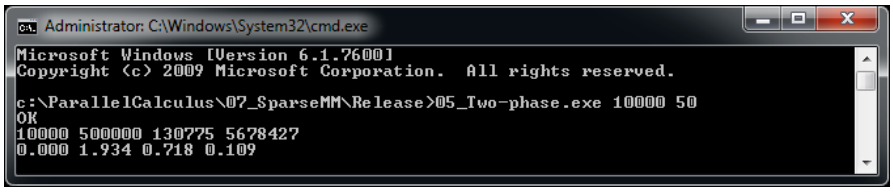

Рис. 27. Результаты работы (двухфазная версия, численная фаза, версия 2)

Как и следовало ожидать, численная фаза стала работать на порядок быстрее и существенно (более чем в два раза) обгоняет символическую.

В заключение приведем оптимизированный вариант численной фазы, построенный на идеях проекта **02_Optim2**.

```
int NumericMult2(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;

  clock_t start = clock();
  int *temp = new int[N];
  int index = 0;

  for (int i = 0; i < N; i++)
  {
    int ind1 = C.RowIndex[i];
    int ind2 = C.RowIndex[i + 1];
    if (ind2 > ind1)
    {
      memset(temp, -1, sizeof(int) * N);
      int Aind1 = A.RowIndex[i];
      int Aind2 = A.RowIndex[i + 1];
```

```
      for (int j = Aind1; j < Aind2; j++)
      {
        int col = A.Col[j];
        temp[col] = j; // Значит, что a[i, НОМЕР] лежит
                       // в ячейке массива Value
                       // с номером temp[НОМЕР]
      }
      // Индекс построен
      // Умножаем строку i матрицы A и столбцы матрицы B
      for (int j = ind1; j < ind2; j++)
      {
        int col = C.Col[j];
        // Надо перемножить i-ю строку матрицы A
        // и строку col матрицы BT
        double sum = 0;
        int Bind1 = B.RowIndex[col];
        int Bind2 = B.RowIndex[col + 1];
        for (int k = Bind1; k < Bind2; k++)
        {
          int lookup = B.Col[k];
          int aind = temp[lookup];
          if (aind != -1)
            sum += A.Value[aind] * B.Value[k];
        }
        C.Value[index] = sum;
        index++;
      }
    }
  }

  delete [] temp;

  clock_t finish = clock();

  time = (double)(finish - start) / CLOCKS_PER_SEC;

  return 0;
}
```

Проведите вычислительные эксперименты с представленной версией функции **`NumericMult()`**. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 28).

Данная версия численной фазы работает существенно быстрее, чем предыдущая (0.483 сек. против 0.718 сек.). При этом по сумме времен по сравнению с реализацией из проекта **03_Optim2** от разделения на фазы мы не проиграли (2.512 сек. и 1.935 + 0.483 = 2.418 сек. – можно считать, что время работы одинаково).

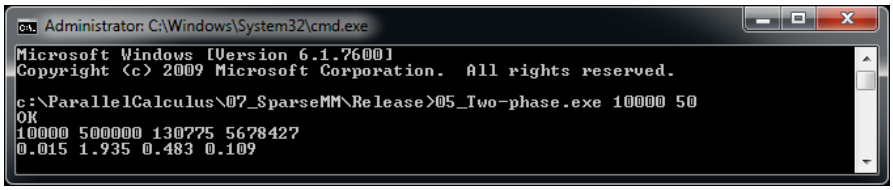

Рис. 28. Результаты работы (двухфазная версия, численная фаза, версия 3)

9. Алгоритм Густавсона

В данном разделе кратко описывается один из наиболее быстрых на практике алгоритмов матричного умножения, предложенный Густавсоном [2, 10]. В отличие от алгоритмов, рассмотренных ранее, здесь не требуется транспонировать матрицу B. Суть метода заключается в том, чтобы избежать проблемы выделения столбца в матрице B. Для этого предлагается изменить порядок вычислений: вместо того, чтобы умножать строку на столбец, – вычислять произведения каждого элемента матрицы A на все элементы соответствующей строки матрицы B, постепенно накапливая частичные суммы.

Рассмотрим i-ю строку матрицы A. Для всех значений j умножим элемент A[i, j] на все элементы j-ой строки матрицы B. Все произведения будем накапливать в ячейки, соответствующие i-ой строке матрицы C. По окончании обработки i-ой строки матрицы A оказывается полностью вычисленной i-я строка матрицы C [2].

В рамках дополнительных заданий к работе предлагаем выполнить программную реализацию алгоритма, распараллелить ее. При этом рекомендуем подумать о том, где и как организовать накопление частичных сумм. Представляется, что это один из важных моментов на пути достижения лучшей производительности. Также необходимо решить проблему упорядочивания полученной матрицы. Один из вариантов – два раза выполнить ее транспонирование.

Далее представлены результаты экспериментов с одной из возможных реализаций алгоритма, подготовленной студентами 3-го курса факультета ВМК ННГУ Филиппенко С. и Лебедевым С.

Таблица 2. Результаты экспериментов (алгоритм Густавсона, последовательная версия, время дано в секундах)

Порядок матриц (N)	Проект 03_Optim2 A * B	Алгоритм Густавсона A * B	MKL A * B	Проект 03_Optim2 B * A	Алгоритм Густавсона B * A	MKL B * A
10 000	2.512	0.468	0.110	3.339	0.515	3.837
15 000	5.445	0.764	0.171	6.989	0.827	6.068
20 000	9.454	0.842	0.234	12.309	1.185	8.268
25 000	14.617	1.185	0.328	18.658	1.529	10.498
30 000	20.982	1.747	0.406	26.504	1.904	12.823
35 000	34.429	1.872	0.452	35.849	2.278	15.116
40 000	44.460	1.529	0.375	46.503	2.667	17.269

Нетрудно видеть, что алгоритм Густавсона показывает отличные результаты по сравнению с последней рассмотренной реализацией и уверенно обгоняет MKL при некоторых тестовых данных.

10. Параллельная реализация

10.1. Параллельная реализация с использованием OpenMP

Как уже обсуждалось выше, еще один вариант повышения производительности умножения разреженных матриц – распараллеливание. В данном разделе будут рассмотрены параллельные реализации алгоритма умножения матриц, описанного ранее (проект **03_Optim2**). Распараллеливание двухфазного алгоритма, а также разработку последовательной и параллельной версий алгоритма Густавсона предлагается выполнить в рамках дополнительных заданий к работе.

Вернемся к коду из проекта **03_Optim2** и реализуем на его основе OpenMP-версию. Фактически нам придется распределить между созданными потоками работу в функции **`Multiplicate()`**, внеся в нее попутно необходимые изменения для обеспечения корректности обработки данных в параллельном случае.

Итак, создайте в рамках решения **07_SparseMM** новый проект с названием **06_OpenMP**. Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню

Add→New Project…. При добавлении файла в проект задайте имя **main_omp**.

После получения пустого файла **main_omp.cpp** скопируйте в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_omp.cpp**, скопировав в него код из файла **sparse_o2.cpp** проекта **03_Optim2**.

Идея распараллеливания алгоритма, описанного в § 7.2, выглядит достаточно очевидной. Во внешнем цикле мы перебираем строки матрицы A и далее работаем с ними независимо. Таким образом, естественный вариант параллелизма – распределение строк между потоками. Однако проблемы начинаются, как только мы добираемся до момента, когда необходимо записать новые значения в вектора **`columns`**, **`values`** и **`row_index`**. Нам необходимо выполнить в них операцию **`push_back()`**, соответственно потоки могут помешать друг другу сделать это корректно. Защита это участка с помощью, например, директивы **`single`** приведет лишь к существенным накладным расходам, поэтому мы используем другой вариант разрешения конфликта доступа – продублируем все рабочие структуры по числу строк, соберем в них данные по каждой строке отдельно, а в конце одним потоком все «просуммируем».

Следующее замечание касается массива **`temp`**, который мы используем для запоминания расположения ненулевых элементов матрицы A. С ним вариант с дублированием не годится, поскольку его размер равен длине строки. Таким образом, при попытке продублировать его по числу строк мы, фактически, потратим объем памяти лишь вполовину меньший, чем нужен для размещения матрицы A в плотном виде. Естественно, это неприемлемо. К счастью, для этого массива возможен простой выход, его можно продублировать по числу потоков. На самом деле мы поступим еще проще и поместим его объявление непосредственно в параллельную секцию.

Наконец, последний момент касается вектора **`row_index`**. С ним ситуация существенно проще, чем с остальными элементами разреженной матрицы, поскольку его размер равен «число строк + 1». Соответственно его можно объявить обычным массивом, работать с ним в потоках с элементами, соответствующими обрабатываемым строкам (а, значит, бесконфликтно), накапливая число элементов в каждой строке отдельно. И, наконец, по завершении параллельной секции последовательно просуммировать число элементов по строкам, получив итоговый массив.

В результате получится следующий код.

```
int Multiplicate(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
```

```
    return 1;

  int N = A.N;
  int i, j, k;

  clock_t start = clock();

  vector<int>* columns = new vector<int>[N];
  vector<double> *values = new vector<double>[N];

  int* row_index = new int[N + 1];
  memset(row_index, 0, sizeof(int) * N);

#pragma omp parallel
  {
    int *temp = new int[N];
  #pragma omp for private(j, k)
    for (i = 0; i < N; i++)
    {
      // i-я строка матрицы A
      // Обнуляем массив указателей на элементы
      memset(temp, -1, N * sizeof(int));
      // Идем по ненулевым элементам строки и заполняем
      // массив указателей
      int ind1 = A.RowIndex[i], ind2 = A.RowIndex[i + 1];
      for (j = ind1; j < ind2; j++)
      {
        int col = A.Col[j];
        temp[col] = j; // Значит, что a[i, НОМЕР] лежит
        // в ячейке массива Value с номером temp[НОМЕР]
      }
      // Построен индекс строки i матрицы A
      // Теперь необходимо умножить ее на каждую из строк
      // матрицы BT
      for (j = 0; j < N; j++)
      {
        // j-я строка матрицы B
        double sum = 0;
        int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];
        // Все ненулевые элементы строки j матрицы B
        for (k = ind3; k < ind4; k++)
        {
          int bcol = B.Col[k];
          int aind = temp[bcol];
          if (aind != -1)
            sum += A.Value[aind] * B.Value[k];
        }
        if (fabs(sum) > ZERO_IN_CRS)
        {
          columns[i].push_back(j);
```

```
            values[i].push_back(sum);
            row_index[i]++;
          }
        }
      }
      delete [] temp;
    }

    int NZ = 0;
    for(i = 0; i < N; i++)
    {
      int tmp = row_index[i];
      row_index[i] = NZ;
      NZ += tmp;
    }
    row_index[N] = NZ;

    InitializeMatrix(N, NZ, C);

    int count = 0;
    for (i = 0; i < N; i++)
    {
      int size = columns[i].size();
      memcpy(&C.Col[count], &columns[i][0],
             size * sizeof(int));
      memcpy(&C.Value[count], &values[i][0],
             size * sizeof(double));
      count += size;
    }
    memcpy(C.RowIndex, &row_index[0], (N + 1) * sizeof(int));

    delete [] row_index;
    delete [] columns;
    delete [] values;

    clock_t finish = clock();
    time = (double)(finish - start) / CLOCKS_PER_SEC;

    return 0;
}
```

Существенные для понимания моменты в представленном выше коде выделены полужирным начертанием.

Проведите вычислительные эксперименты для описанной реализации. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 29).

```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

c:\ParallelCalculus\07_SparseMM\Release>06_OpenMP.exe 10000 50
OK
10000 500000 130775 5678160
0.000 1.029 0.110
```

Рис. 29. Результаты работы (OpenMP-версия)

В вычислениях было задействовано восемь потоков на восьми ядрах. Нетрудно видеть, что ускорение по сравнению с версией из проекта **03_Optim2** составляет 2.512 / 1.029 = 2.441 и довольно далеко от восьми.

Попробуем разобраться, что происходит. Вновь воспользуемся Amplifier, но уже в режиме Concurrency (рис. 30).

Concurrency - Hotspots by Thread Concurrency — Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree

/Function /Call Stack	CPU Time (Idle Poor Ok Ideal)	Overhead Time	Wait Time (Idle Poor Ok Ideal)
Multiplicate	4.369s	0us	0.809s
__kmp_fork_barrier	0.416s	0us	0s
NtDelayExecution	0.294s	0us	0s
__kmp_x86_pause	0.129s	0us	0s
main	0.124s	0us	0.000s
mkl_spblas_p4m3_dmcsr_notr	0.119s	0us	0s
__kmp_execute_tasks	0.080s	0us	0s
mkl_spblas_p4m3_dmcsradd_notr	0.070s	0us	0s
__kmp_yield	0.040s	0us	0s
SleepEx	0.038s	0us	0s
Sleep	0.030s	0us	0s
SparseDiff	0.030s	0us	0s
Selected 0 row(s):			

Рис. 30. Профиль приложения (OpenMP-версия).
Вид Bottom-up, первая часть

Важным результатом режима профилировки Concurrency является информация о том, в какие моменты времени созданные потоки работали, а в какие – простаивали. Это позволяет сделать вывод о степени параллелизма программы. На вкладке Bottom-up как и прежде отображается список «горячих точек» на уровне функций с указанием времени их работы. При этом дополнительно отображается информация о степени параллелизма. Красный цвет соответствует последовательному выполнению, зеленый – идеально параллельному. Заметим, что функции MKL работали последовательно, но это следствие того, что мы при сборке приложения подключили библиотеку **mkl_sequential.lib** (последовательные реализации), а не **mkl_intel_thread.lib** (параллельные реализации). Наша функция **`Multiplicate()`** существенную часть времени работала в последовательном режиме (красный прямоугольник). При этом в колонке Wait Time приведено немалое на общем фоне время ожидания (18% от общего времени работы основной вычислительной функции).

Увеличим нижнюю часть экрана с более подробными результатами анализа (рис. 31):

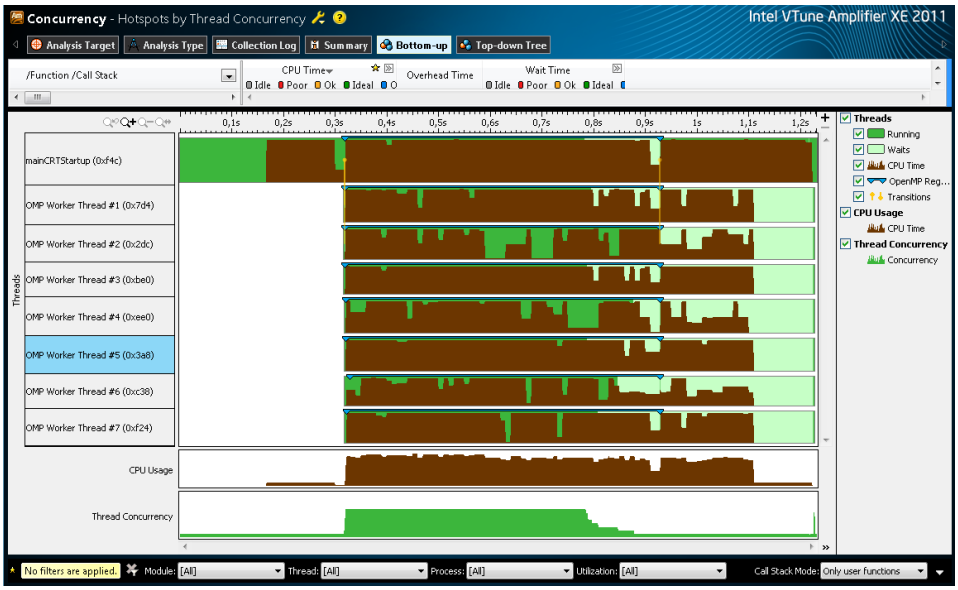

Рис. 31. Профиль приложения (OpenMP-версия).
Вид Bottom-up, вторая часть

Данная временная диаграмма отображает динамику работы профилируемого приложения. Так, видно, что работали 8 потоков. В начале первый из потоков (верхняя строка) работал последовательно, после чего инициировал создание еще 7 потоков. В течение некоторого времени 8 потоков работали в рамках параллельных секций (синие скобки). По их окончании первый поток выполнил некоторые действия в последовательном режиме. Зная, как организовано приложение, нетрудно догадаться, что часть диаграммы, показывающая параллельную работу 8 потоков, как раз соответствует функции **`Multiplicate()`**. Однако степень параллелизма оставляет желать лучшего – потоки используют процессор с разной интенсивностью, заканчивают работу в разное время и подолгу ждут завершения работы в других потоках. Посмотрим на диаграмму Thread Concurrency (зеленого цвета). Нетрудно видеть, что параллелизм сходит на нет значительно раньше того момента, как завершаются параллельные секции.

Можно заключить, что вычислительные ресурсы используются неэффективно, – результаты профилировки качественно соответствуют наблюдаемому низкому ускорению.

Посмотрим на вкладку Summary (рис. 32).

Данная вкладка предоставляет интегральную информацию о степени параллелизма. Из первой диаграммы видно, что в основном приложение работало в 1 и в 8 потоков. При этом если поток находится в режиме ожидания, считается, что он «работает». Вторая диаграмма дает информацию о степени использования CPU. Из обеих диаграмм можно сделать те же выводы: параллельная версия работает неэффективно. Отметим, что диаграммы соответствуют не только функции **`Multiplicate()`**, а всему приложению, поэтому в нашем случае (распараллелена только основная вычислительная функция) дают более пессимистичную оценку состояния дел.

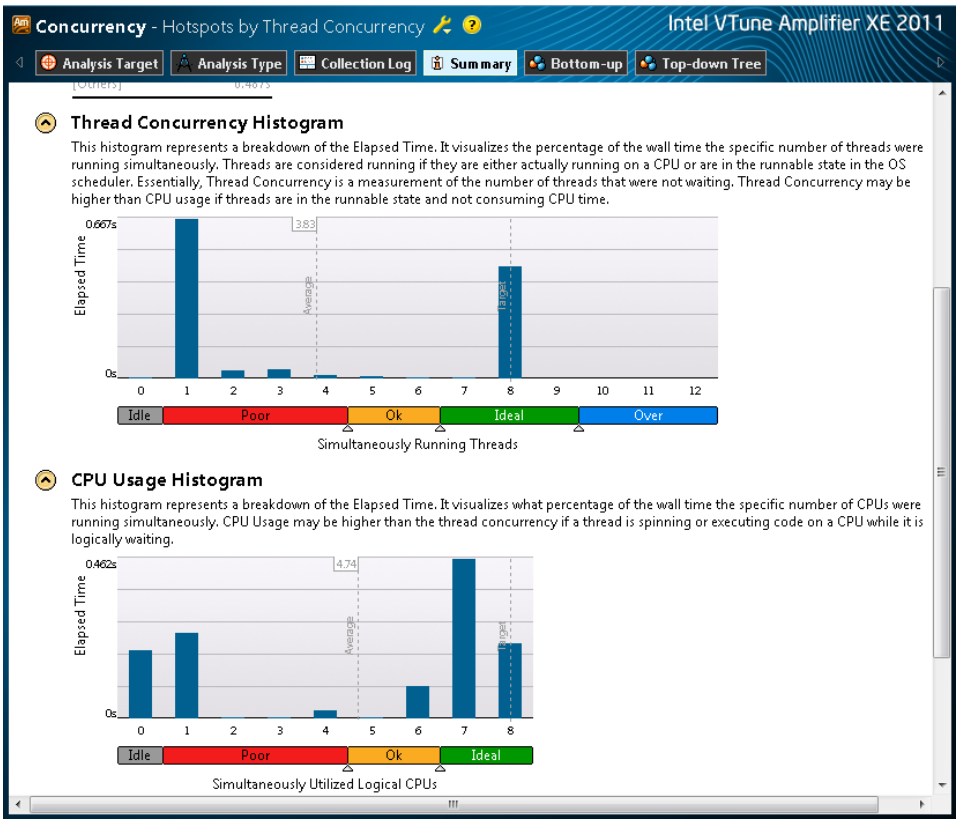

Рис. 32. Профиль приложения (OpenMP-версия). Вид Summary

Итак, потоки существенное время находятся в режиме ожидания. В чем же дело, почему это происходит? Для получения ответа на этот вопрос в данном случае можно отключить показ загрузки процессора в виде Bottom-up (рис. 33). Потоки заканчивают работу в разное время и ждут самого медленного. Временная диаграмма наводит на мысль о том, что причина низкого ускорения может быть обусловлена невыгодным соотношением времени расчета и времени ожидания, возникшим вследствие малого количества вычислений в расчете на один поток. Попробуем побольше «нагру-

зить» каждый из потоков, увеличив размерность решаемой задачи. Конечно, это не избавит нас от дисбаланса, но сократит долю времени ожидания в общем времени работы.

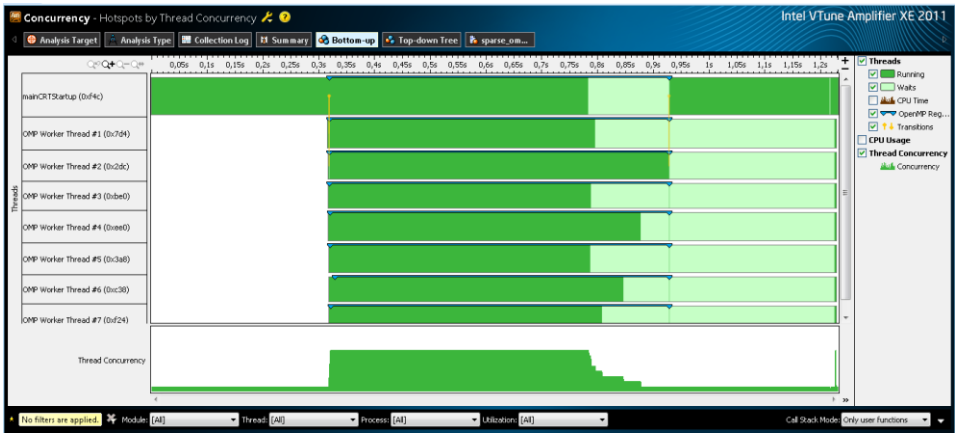

Рис. 33. Профиль приложения (OpenMP-версия).
Вид Bottom-up, вторая часть, загрузка CPU отключена

Посмотрим, как меняется ускорение с ростом вычислительной нагрузки, то есть с увеличением порядка матриц. Как и ранее сравнение будем проводить с версией из проекта **03_Optim2**.

Таблица 3. Результаты экспериментов (OpenMP vs 03_Optim2, версия 1).

Порядок матриц (N)	1 поток, t (сек)	8 потоков, t (сек)	Ускорение
10 000	2.512	1.029	2.441
15 000	5.445	1.841	2.958
20 000	9.454	2.667	3.545
25 000	14.617	3.931	3.718
30 000	20.982	5.538	3.789
35 000	34.429	7.067	4.872
40 000	44.460	8.408	5.288

Как видим, ускорение растет и добирается до величины порядка 5. Вновь спрофилируем наше приложение для матриц размера 40000 x 40000 (рис. 34).

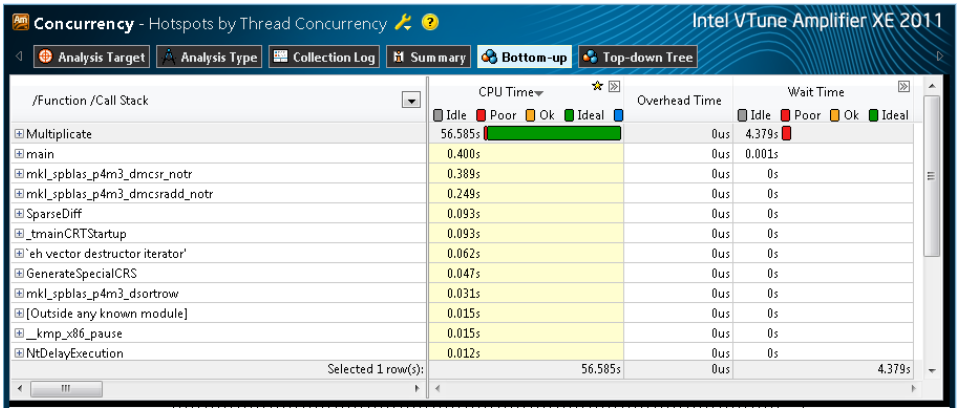

Рис. 34. Профиль приложения (OpenMP-версия).
Вид Bottom-up, первая часть, увеличена загрузка CPU

Заметим, что относительное время ожидания составляет около 7,7%, что меньше, чем для матриц 10000 x 10000. Посмотрим на временную диаграмму (рис. 35).

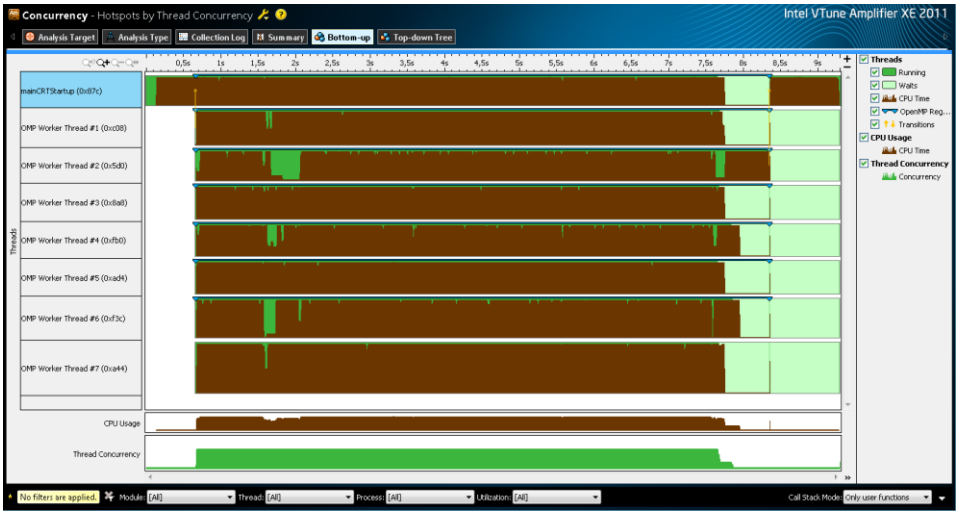

Рис. 35. Профиль приложения (OpenMP-версия).
Вид Bottom-up, вторая часть, увеличена загрузка CPU

«Пробелов» в использовании центрального процессора стало намного меньше – вычислительные ядра загружены расчетами. При этом в конце по-прежнему наблюдается разное время окончания счета – дисбаланс вычислительной нагрузки. Устранение этого дисбаланса является направлением дальнейшей работы.

10.2. Балансировка нагрузки

Для балансировки нагрузки используем возможности OpenMP по построению расписаний (schedule) при распределении итераций цикла между потоками.

Изменим использованную в предыдущем разделе прагму

```
#pragma omp for private(j, k)
```

на

```
#pragma omp for private(j, k) schedule(static, chunk)
```

Переменная **chunk** должна быть объявлена перед параллельной секцией. В качестве значения выберем 10. Предоставляем читателю провести эксперименты по определению ее оптимального размера самостоятельно.

Повторим эксперименты.

Таблица 4. Результаты экспериментов (OpenMP vs 03_Optim2, версия 2).

Порядок матриц (N)	1 поток, t (сек)	8 потоков, t (сек)	Ускорение
10 000	2.512	0.920	2.730
15 000	5.445	1.591	3.422
20 000	9.454	2.215	4.268
25 000	14.617	3.198	4.571
30 000	20.982	4.444	4.721
35 000	34.429	5.568	6.183
40 000	44.460	6.394	6.953

Как видим, ситуация существенно улучшилась и максимальное ускорение почти равно 7. Профиль приложения показывает, что доля времени ожидания уменьшилась до 4,8%, хотя полностью дисбаланс не пропал. Предоставляем читателю самостоятельно спрофилировать приложение, а также проверить, что будет, если поменять расписание со **static** на **dynamic**.

10.3. Параллельная реализация с использованием TBB

Последняя реализация алгоритма умножения, которую мы разберем в рамках данной лабораторной работы – реализация с использованием библиотеки TBB.

Цель, во-первых, состоит в том, чтобы продемонстрировать применение TBB к написанию параллельного умножения разреженных матриц, и, во-вторых, посмотреть, как алгоритм распределения нагрузки между потоками в TBB справится с такой достаточно «разбалансированной» реализацией.

Итак, создадим в рамках решения **07_SparseMM** новый проект с названием **07_TBB**. Повторите все действия, описанные в § 6.3, с той лишь разницей, что начать нужно с выбора решения **07_SparseMM** в окне **Solution Explorer** и выполнения команды контекстного меню **Add→New Project…**. При добавлении файла в проект задайте имя **main_tbb**.

После получения пустого файла **main_tbb.cpp** скопируем в него код из файла **main_n.cpp** проекта **01_Naive**. Также скопируйте в новый каталог файлы **util.h**, **util.cpp**, **sparse.h** и создайте файл **sparse_tbb.cpp**, скопировав в него код из файла **sparse_o2.cpp** проекта **03_Optim2**.

Прежде чем приступать к обсуждению реализации TBB-версии умножения, напомним, что использование библиотеки предполагает написание кода в классах. Подробные сведения могут быть найдены в [11], здесь же мы кратко опишем те элементы библиотеки, которые будут использованы нами в процессе разработки.

- Инициализация библиотеки с использованием объекта класса **`task_scheduler_init`**.
- Распараллеливание цикла с помощью шаблонной функции **`parallel_for()`**.
- Одномерное итерационное пространство **`blocked_range`**.
- Класс-функтор, который нам предстоит разработать и который, собственно, и будет реализовывать основную часть умножения в методе **`operator()`**.

Итак, прежде всего, необходимо отметить, что идейно разработка параллельной версии на основе TBB будет очень похожа на OpenMP. Мы также продублируем по числу строк служебные вектора **`columns`** и **`values`**. Также создадим массив **`row_index`** длины «число строк + 1» и точно также будем запоминать в каждом его элементе, сколько не нулей будет содержать соответствующая строка матрицы C. Фактически отличия в реализации функции **`Multiplicate()`** будут состоять в использовании шаблонной функции **`parallel_for()`**, которая «скроет» в себе весь содержательный код – то, что в OpenMP-версии составляло содержимое параллельной секции.

Сама функция **`Multiplicate()`** при этом будет выглядеть следующим образом.

```
int Multiplicate(crsMatrix A, crsMatrix B, crsMatrix &C,
  double &time)
{
  if (A.N != B.N)
    return 1;

  int N = A.N;
  int i;

  clock_t start = clock();

  task_scheduler_init init();

  vector<int>* columns = new vector<int>[N];
  vector<double> *values = new vector<double>[N];
  int* row_index = new int[N + 1];
  memset(row_index, 0, sizeof(int) * N);

  int grainsize = 10;

  parallel_for(blocked_range<int>(0, A.N, grainsize),
    Multiplicator(A, B, columns, values, row_index));

  int NZ = 0;
  for(i = 0; i < N; i++)
  {
    int tmp = row_index[i];
    row_index[i] = NZ;
    NZ += tmp;
  }
  row_index[N] = NZ;

  InitializeMatrix(N, NZ, C);

  int count = 0;
  for (i = 0; i < N; i++)
  {
    int size = columns[i].size();
    memcpy(&C.Col[count], &columns[i][0],
           size * sizeof(int));
    memcpy(&C.Value[count], &values[i][0],
           size * sizeof(double));
    count += size;
  }
  memcpy(C.RowIndex, &row_index[0], (N + 1) * sizeof(int));

  clock_t finish = clock();
  time = (double)(finish - start) / CLOCKS_PER_SEC;
```

```
  return 0;
}
```

Использованное нами итерационное пространство **blocked_range** является встроенным. Таким образом, для завершения реализации нам необходимо создать класс-функтор **Multiplicator**, принимающий на вход две матрицы и оперирующий рабочими векторами **columns**, **values** и **row_index**. Объявление этого класса разместим в файле sparse.h.

```
class Multiplicator
{
  crsMatrix A, B;
  vector<int>* columns;
  vector<double>* values;
  int *row_index;
public:
  Multiplicator(crsMatrix& _A, crsMatrix& _B,
    vector<int>* &_columns, vector<double>* &_values,
    int *_row_index) : A(_A), B(_B), columns(_columns),
    values(_values), row_index(_row_index)
  {}

  void operator()(const blocked_range<int>& r) const
  {
    ...
  }
};
```

Заметим, что согласно идеологии TBB класс-функтор должен содержать только ссылки на обрабатываемые потоками данные. Исключения могут быть сделаны для небольших по размеру переменных, копирование которых при расщеплении итерационного пространства будет требовать мало ресурсов. В данном случае мы сделали такое исключение для матриц A и B. Отметим, что размер этих структур составляет 20 байт в 32-разрядной операционной системе (2 поля типа **int** и 3 указателя). Отметим также, что в данном случае мы можем не писать необходимый для класса-функтора конструктор копирования, поскольку его автоматически созданная версия будет работать правильно. Почему это так, предоставляем читателю разобраться самостоятельно. За дополнительными сведениями можно обратиться к источнику [11].

Осталось реализовать метод класса-функтора **operator()**.

Как уже указано выше, входом оператора является итерационное пространство, и задача программиста состоит в том, чтобы получить его границы и реализовать вычислительную часть (обычно в виде цикла **for**).

В данном случае вычислительная часть соответствует параллельной секции в OpenMP-версии с незначительными изменениями.

```
void operator()(const blocked_range<int>& r) const
{
  int begin = r.begin();
  int end = r.end();
  int N = A.N;

  int i, j, k;
  int *temp = new int[N];

  for (i = begin; i < end; i++)
  {
    // i-я строка матрицы A
    // Обнуляем массив указателей на элементы
    memset(temp, -1, N * sizeof(int));
    // Идем по ненулевым элементам строки и заполняем
    // массив указателей
    int ind1 = A.RowIndex[i], ind2 = A.RowIndex[i + 1];
    for (j = ind1; j < ind2; j++)
    {
      int col = A.Col[j];
      temp[col] = j; // Значит, что a[i, НОМЕР] лежит
      // в ячейке массива Value с номером temp[НОМЕР]
    }
    // Построен индекс строки i матрицы A
    // Теперь необходимо умножить ее на каждую из строк
    // матрицы BT
    for (j = 0; j < N; j++)
    {
      // j-я строка матрицы B
      double sum = 0;
      int ind3 = B.RowIndex[j], ind4 = B.RowIndex[j + 1];
      // Все ненулевые элементы строки j матрицы B
      for (k = ind3; k < ind4; k++)
      {
        int bcol = B.Col[k];
        int aind = temp[bcol];
        if (aind != -1)
          sum += A.Value[aind] * B.Value[k];
      }
      if (fabs(sum) > ZERO_IN_CRS)
      {
        columns[i].push_back(j);
        values[i].push_back(sum);
        row_index[i]++;
      }
    }
  }
  delete [] temp;
}
```

```
};
```

Проведите вычислительные эксперименты для описанной реализации. Для сравнения представим результаты, полученные авторами на тестовой инфраструктуре, описанной в § 1.3 (рис. 36).

Рис. 36. Результаты работы (TBB-версия)

В заключение повторим эксперименты по выяснению изменения ускорения с ростом вычислительной нагрузки, то есть с увеличением порядка матриц. Как и в предыдущем разделе сравнение будем проводить с версией из проекта **03_Optim2**.

Таблица 5. Результаты экспериментов (TBB vs 03_Optim2).

Порядок матриц (N)	1 поток, t (сек)	8 потоков, t (сек)	Ускорение
10 000	2.512	0.624	4.026
15 000	5.445	1.139	4.781
20 000	9.454	1.731	5.462
25 000	14.617	2.589	5.646
30 000	20.982	3.620	5.796
35 000	34.429	4.758	7.236
40 000	44.460	5.850	7.600

Как видим, TBB в данной задаче существенно обгоняет OpenMP версию. Однако необходимо отметить, что это может быть вызвано не только более хорошей балансировкой нагрузки между потоками в TBB, но и более хорошей, чем в случае OpenMP оптимизации кода.

Как и ранее, выбранный нами размер порции при расщеплении (grainsize = 10) может не быть оптимальным. Отсылаем читателя к источнику [11] за алгоритмом его подбора.

11. Дополнительные задания

Дополнительные задания имеют разный уровень сложности. Некоторые из них являются достаточно трудоемкими и подходят в качестве зачетных тем для студентов, изучающих параллельные численные методы.

1. Провести эксперименты с умножением матриц в столбцовом формате (CCS). Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Выполнить сравнение с базовой версией, представленной в работе (матрицы в формате CRS). Разработать и настроить параллельную реализацию.
2. Провести эксперименты с умножением матриц в координатном формате. Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Выполнить сравнение с базовой версией, представленной в работе (матрицы в формате CRS). Разработать и настроить параллельную реализацию.
3. Провести эксперименты с умножением матриц другой структуры. Рассмотреть матрицы с равным числом элементов в строках. Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Разработать и настроить параллельную реализацию.
4. Провести эксперименты с умножением матриц другой структуры. Рассмотреть матрицы с нарастающим числом элементов в строках. Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Разработать и настроить параллельную реализацию.
5. Провести эксперименты с умножением матриц с сохранением результата в плотную матрицу C. Рассмотреть матрицы с равным числом элементов в строках. Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Разработать и настроить параллельную реализацию.
6. Провести эксперименты с умножением матриц с сохранением результата в плотную матрицу C. Рассмотреть матрицы с нарастающим числом элементов в строках. Выявить и объяснить эффекты, связанные с соотношением времен работы разных последовательных алгоритмов. Разработать и настроить параллельную реализацию.
7. Адаптировать использованные в работе алгоритмы для прямоугольных матриц. Выполнить программную реализацию. Провести вычислительные эксперименты.

8. В наивной версии алгоритма умножения проверить, будет ли выигрыш от использования одного вектора вместо двух (см. сноску в соответствующем разделе). Исследовать вопрос о возможности получения выигрыша в производительности при отказе от использования STL. *Указание*: для эксперимента инициализируйте матрицу C, в качестве NZ используйте заведомо большое число, чтобы изначально выделилось достаточно памяти. Это позволит оценить возможный выигрыш сверху.
9. Реализовать алгоритм Густавсона для умножения разреженных матриц. Разработать параллельную реализацию. Провести вычислительные эксперименты.
10. Ознакомьтесь с научной литературой по рассматриваемой теме. Изучите/разработайте другие алгоритмы умножения. Опробуйте их на практике, проведите вычислительные эксперименты. Убедитесь в корректности результатов. Сравните время работы.
11. Выполните распараллеливание символической и численной фазы матричного умножения. Проведите вычислительные эксперименты. Убедитесь в корректности результатов. Проведите анализ масштабируемости.
12. Разработайте другой, более быстрый алгоритм для выполнения численной фазы. Учитывая тот факт, что приведенный вариант численной фазы уступает по скорости общему времени работы функции умножения матриц из библиотеки MKL, есть потенциал для оптимизации.
13. Выясните оптимальный размер порции данных для OpenMP- и TBB-реализаций.

12. Литература

12.1. Использованные источники информации

1. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984.
2. Писсанецки С. Технология разреженных матриц. — М.: Мир,1988.

12.2. Дополнительная литература

3. Голуб Дж., Ван Лоун Ч. Матричные вычисления. – М.: Мир, 1999.
4. Тьюарсон Р. Разреженные матрицы. – М.: Мир, 1977.
5. Bik A.J.C. The software vectorization handbook. Applying Multimedia Extensions for Maximum Performance. – IntelPress, 2004.

6. Gerber R., Bik A.J.C., Smith K.B., Tian X. The Software Optimization Cookbook. High-Performance Recipes for the Intel® Architecture. – Intel Press, 2006.

7. Касперски К. Техника оптимизации программ. Эффективное использование памяти. – BHV, 2003.

8. Stathis P., Cheresiz D., Vassiliadis S., Juurlink B. Sparse Matrix Transpose Unit // 18th International Parallel and Distributed Processing Symposium (IPDPS'04) – Papers, vol. 1, 2004. [http://ce.et.tudelft.nl/publicationfiles/869_1_IPDPS2004paper.pdf]

9. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. – М.: МЦНМО, 2001.

10. Gustavson F. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition // ACM Transactions on Mathematical Software (TOMS), Volume 4 Issue 3, Sept. 1978. – Pp. 250-269.

11. Корняков К.В., Мееров И.Б., Сиднев А.А., Сысоев А.В., Шишков А.В. Инструменты параллельного программирования в системах с общей памятью. – Учебное пособие / Под ред. проф. В.П. Гергеля. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2010. – 201 с.

12. Белов С.А., Золотых Н.Ю. Численные методы линейной алгебры. Лабораторный практикум. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2005. – 264 с.

12.3. Ресурсы сети Интернет

13. Buttari A. Software Tools for Sparse Linear Algebra Computations –

[http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.7162&rep=rep1&type=pdf]

14. Официальная страница Джеймса Деммеля (James Demmel Home Page) –

[http://www.cs.berkeley.edu/~demmel/].

15. Demmel J. U.C. Berkeley Math 221 Home Page: Matrix Computations / Numerical Linear Algebra – [http://www.cs.berkeley.edu/~demmel/ma221].

16. Demmel J. U.C. Berkeley CS267/EngC233 Home Page: Applications of Parallel Computers – [http://www.cs.berkeley.edu/~demmel/cs267_Spr10/].

17. Demmel J. CS170: Efficient Algorithms and Intractable Problems – [http://www.cs.berkeley.edu/~demmel/cs170_Spr10/#starthere].

18. Справочная система к библиотеке PETSc – [http://www.geo.uu.nl/~govers/petsc/node36.html#Node36].