

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

**Образовательный комплекс
«Параллельные численные методы»**

**Лабораторная работа
Методы прогонки и циклической редукции
в решении СЛАУ с ленточной матрицей**

Кустикова В.Д.

При поддержке компании Intel

Нижний Новгород
2011

Содержание

ВВЕДЕНИЕ	3
1. МЕТОДИЧЕСКИЕ УКАЗАНИЯ	3
1.1. Цели и задачи работы.....	3
1.2. СТРУКТУРА РАБОТЫ.....	4
1.3. ТЕСТОВАЯ ИНФРАСТРУКТУРА	5
1.4. РЕКОМЕНДАЦИИ ПО ПРОВЕДЕНИЮ ЗАНЯТИЙ	5
2. ЗАДАЧА ВЫЧИСЛЕНИЯ ЦЕНЫ СОСТАВНОГО ОПЦИОНА.....	6
3. МЕТОД ПРОГОНКИ.....	7
4. МЕТОД ЦИКЛИЧЕСКОЙ РЕДУКЦИИ.....	9
5. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ	13
5.1. РЕАЛИЗАЦИЯ МЕТОДА ПРОГОНКИ	13
5.2. ПОСЛЕДОВАТЕЛЬНАЯ РЕАЛИЗАЦИЯ МЕТОДА ЦИКЛИЧЕСКОЙ РЕДУКЦИИ.....	17
5.3. ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ МЕТОДА ЦИКЛИЧЕСКОЙ РЕДУКЦИИ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ OPENMP	18
5.4. ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ МЕТОДА ЦИКЛИЧЕСКОЙ РЕДУКЦИИ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ INTEL THREADING BUILDING BLOCKS.....	19
6. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ПРИЛОЖЕНИЯ ПРИ ИСПОЛЬЗОВАНИИ ПОСЛЕДОВАТЕЛЬНЫХ РЕАЛИЗАЦИЙ МЕТОДОВ ПРОГОНКИ И ЦИКЛИЧЕСКОЙ РЕДУКЦИИ.....	24
7. АНАЛИЗ МАСШТАБИРУЕМОСТИ ПРИЛОЖЕНИЯ ПРИ ИСПОЛЬЗОВАНИИ OPENMP-РЕАЛИЗАЦИИ МЕТОДА ЦИКЛИЧЕСКОЙ РЕДУКЦИИ.....	29
8. АНАЛИЗ МАСШТАБИРУЕМОСТИ ПРИЛОЖЕНИЯ ПРИ ИСПОЛЬЗОВАНИИ ТВВ-РЕАЛИЗАЦИИ МЕТОДА ЦИКЛИЧЕСКОЙ РЕДУКЦИИ.....	35
9. ДОПОЛНИТЕЛЬНЫЕ ЗАДАНИЯ.....	36
10. ЛИТЕРАТУРА	37
10.1. ОСНОВНАЯ ЛИТЕРАТУРА	37
10.2. РЕСУРСЫ СЕТИ ИНТЕРНЕТ	37
10.3. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.....	38

Введение

Подавляющее большинство задач вычислительной математики сводится к решению систем алгебраических уравнений (СЛАУ) вида $Ax = b$, где A – матрица размерности $n \times m$, b – известный вектор размера n , а x – вектор размера m . Решением системы указанного вида является вектор x , при подстановке которого в систему получается верное тождество. Система называется *совместной*, если она имеет хотя бы одно решение.

Методы решения СЛАУ можно разделить на прямые и итерационные.

Прямые методы позволяют получить точное решение системы за конечное количество действий. Отметим, что программные реализации прямых методов не предоставляют абсолютно точное решение СЛАУ вследствие погрешности округления. Примерами прямых методов решения СЛАУ являются метод Гаусса, метод Холецкого и др. [1-3].

Итерационные методы строят последовательность приближений к решению системы уравнений [1-4]. Как правило, за конечное число итераций предел последовательности приближений не достигается. Поэтому вводятся параметры остановки таких методов. Примерами итерационных методов являются методы Якоби, Гаусса-Зейделя, верхней релаксации и ряд других [1-3].

В данной лабораторной работе рассматривается СЛАУ с трехдиагональной матрицей. Для решения таких СЛАУ известны специальные методы, в частности, рассмотренные в данной работе *метод прогонки* и *метод циклической редукции*. Метод циклической редукции является немногим более сложным в реализации, но на него меньшее влияние, по сравнению с прогонкой, оказывают погрешности округления [10]. Для метода циклической редукции в работе рассмотрены вопросы распараллеливания в системах с общей памятью.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – продемонстрировать практическое применение параллельных алгоритмов линейной алгебры на примере прикладной задачи из области финансовой математики.

Данная цель предполагает решение следующих основных задач:

1. Изучение постановки задачи вычисления цены составного опциона [6].
2. Освоение методов прогонки и циклической редукции [5] для решения СЛАУ с трехдиагональной матрицей для случая матрицы с одинаковыми элементами на каждой из диагоналей.
3. Реализация метода прогонки для решения СЛАУ [1] с трехдиагональной матрицей полученного вида.
4. Выполнение программной реализации алгоритма циклической редукции в случае трехдиагональной матрицы специального вида, возникающей в рассматриваемой прикладной задаче.
5. Распараллеливание выполненной реализации метода циклической редукции на системы с общей памятью с использованием технологии OpenMP [7, 11].
6. Распараллеливание последовательной реализации метода циклической редукции с использованием библиотеки TBV [8, 9, 12].
7. Анализ производительности реализованной схемы решения дифференциального уравнения в частных производных при использовании методов прогонки и циклической редукции для решения СЛАУ с трехдиагональной матрицей.
8. Анализ масштабируемости приложения с использованием параллельных реализаций метода циклической редукции для решения СЛАУ с трехдиагональной матрицей.

1.2. Структура работы

В работе предлагается содержательная постановка задачи вычисления цены составного опциона. Изменение цены составного опциона описывается дифференциальным уравнением в частных производных. Решение данного дифференциального уравнения сводится к многократному решению СЛАУ с трехдиагональной матрицей, у которой на каждой диагонали в отдельности стоит одинаковое число. Далее в работе предполагается реализовать метод прогонки, а также реализовать и распараллелить метод циклической редукции [5] для решения СЛАУ указанного вида. По завершении программной реализации проводится сравнение производительности приложения при использовании последовательных версий прогонки и циклической редукции и анализ масштабируемости в случае параллельной реализации редукции.

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, профилировщик, отладчик	Intel Parallel Studio XE
Библиотека TBV	Intel® Threading Building Blocks 3.0 for Windows, Update 3 (в составе Intel® Parallel Studio XE 2011)

1.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется следующая последовательность действий:

1. Напомнить вводную информацию описательного характера о методах решения СЛАУ с матрицами общего вида.
2. Рассмотреть содержательную постановку прикладной задачи.
3. Рассмотреть метод прогонки для решения СЛАУ с трехдиагональной матрицей общего вида и трехдиагональной матрицей, содержащей одинаковые элементы на диагоналях.
4. Рассмотреть метод циклической редукции для решения СЛАУ с трехдиагональной матрицей специального вида.
5. Реализовать метод прогонки для решения СЛАУ с матрицей специального вида, который возникает в рассматриваемой задаче.
6. Реализовать последовательную версию алгоритма циклической редукции для случая трехдиагональной матрицы, у которой на каждой диагонали в отдельности стоят одинаковые элементы.
7. Распараллелить реализацию алгоритма циклической редукции сначала с использованием технологии OpenMP [11], а затем с помощью средств библиотеки TBV [12].

8. Провести анализ производительности приложения с использованием последовательных версий методов прогонки и циклической редукции для решения СЛАУ.
9. Выполнить анализ масштабируемости приложения при использовании параллельных версий метода циклической редукции для решения СЛАУ.

2. Задача вычисления цены составного опциона

Опцион (лат. *optio* – выбор, желание, усмотрение) – договор, по которому потенциальный покупатель или потенциальный продавец получает право, но не обязательство, совершить покупку или продажу актива (товара, ценной бумаги, акции, валюты) по заранее оговоренной цене в определенный договором момент в будущем или на протяжении определенного отрезка времени [13].

Цена опциона – это плата, которая взимается за право осуществить покупку или продажу конкретного актива в пределах оговоренного срока в будущем.

Исполнением опциона называется сделка, которая инициируется держателем опциона в части осуществления его права на покупку (продажу) определенного актива. Цена покупки (продажи) актива в будущем – *цена исполнения опциона (страйк)*.

Составной опцион (СО) – опцион, в основе которого лежат один или более базовых активов. В некоторых случаях в основе составного опциона может лежать другой опцион.

В работе рассматривается задача вычисления оптимальной цены исполнения составного опциона на протяжении некоторого временного промежутка.

Жизненный цикл составного опциона от начального момента времени до момента исполнения опциона делится на конечное число стадий – временных отрезков. Держатель СО может реализовать свое право на покупку или продажу активов в конце каждой из этих стадий, исходя из возможной прибыли, которую он может получить от сделки (исполнения опциона). Если держатель обладает правом на покупку активов и суммарная стоимость активов больше суммы цены СО и цены исполнения СО, то держатель реализует свое право, иначе продолжает удерживать опцион. Если же держатель обладает правом на продажу активов и суммарная стоимость активов меньше суммы цены СО и цены исполнения СО, то он реализует свое право на продажу, в противном случае, продолжает удерживать опцион.

Пусть существует m^* моментов взимания платы за опцион в пределах времени жизни СО ($t_k^*, k = 1, 2, \dots, m^*$) и n моментов времени, когда держатель принимает решение об исполнении СО ($t_i, i = 0, 1, \dots, n, n \geq m^*$). Множество моментов взимания платы является подмножеством моментов принятия решений. Обозначим через $T = t_n - t_0$ время жизни СО.

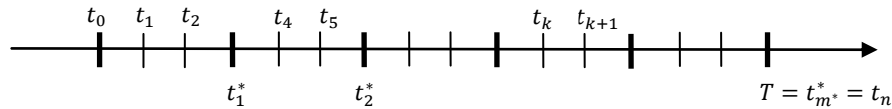


Рис. 1. Временная шкала периода жизни СО

Изменение цены СО на каждом малом интервале временной шкалы описывается дифференциальным уравнением в частных производных с набором естественных граничных условий [6]. Задача решается в обратном времени для каждой стадии $[t_k, t_{k+1}]$ между двумя последовательными моментами принятия решения об исполнении СО. Затем на всех стадиях определяется оптимальное значение СО.

Решение дифференциального уравнения выполняется с помощью вычислительной схемы Кранка-Николсона [2]. Для поиска оптимальной цены СО на стадии вводится сетка, показанная на Рис. 2. Дифференциальное уравнение аппроксимируется СЛАУ с трехдиагональной матрицей, у которой на каждой диагонали стоит одинаковое число. Размерность сетки J по вертикали определяет размер матрицы в СЛАУ $(J - 1) \times (J - 1)$.

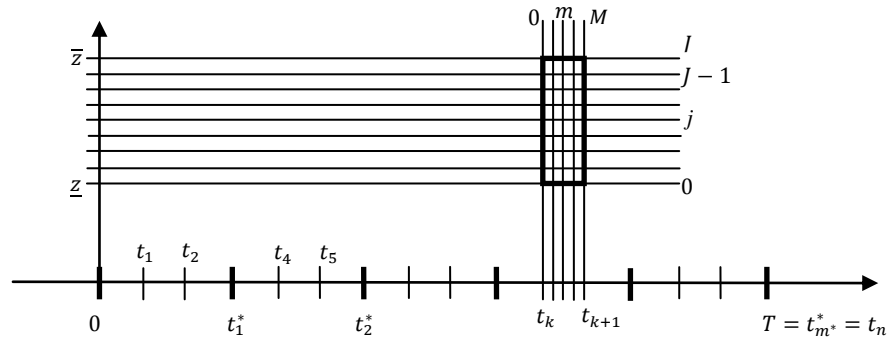


Рис. 2. Сетка для построения вычислительной схемы на каждой стадии изменения цены СО

3. Метод прогонки

Вычисление оптимальной цены СО, в конечном счете, сводится к многократному решению трехдиагональной системы уравнений вида (1).

$$\begin{bmatrix} b & c & & & & \\ a & b & c & & & \\ & a & b & c & & \\ & & \ddots & \ddots & \ddots & \\ & & & a & b & c \\ & & & & a & b \end{bmatrix}_{(N-1) \times (N-1)} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix}_{(N-1) \times 1} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{pmatrix}_{(N-1) \times 1} \quad (1)$$

Рассмотрим метод прогонки, применяемый для решения СЛАУ с трехдиагональной матрицей, для случая, когда на каждой диагонали в отдельности стоят одинаковые элементы¹.

Предположим, что имеет место соотношение (2).

$$x_i = \alpha_{i+1}x_{i+1} + \beta_{i+1} \quad (2)$$

с неопределенными коэффициентами α_{i+1} и β_{i+1} , и подставим выражение (2) в i -е уравнение системы:

$$(\alpha_i a + b)x_i + cx_{i+1} = f_i - a\beta_i$$

Сравнивая полученное выражение с (2), находим

$$\begin{aligned} \alpha_{i+1} &= -\frac{c}{a\alpha_i + b}, i = 2, \dots, N-2 \\ \beta_{i+1} &= \frac{f_i - a\beta_i}{a\alpha_i + b}, i = 2, \dots, N-2 \end{aligned} \quad (3)$$

Из первого уравнения системы $bx_1 + cx_2 = f_1$ находим $\alpha_2 = -\frac{c}{b}, \beta_2 = \frac{f_1}{b}$.

Вычислив α_2, β_2 и переходя от i к $i+1$ в формулах (2), определим $\alpha_i, \beta_i, i = 3, \dots, N-1$.

Определим x_{N-1} из последнего уравнения системы и условия (1) при $i = N-2$.

$$x_{N-2} = \alpha_{N-1}x_{N-1} + \beta_{N-1},$$

$$ax_{N-2} + bx_{N-1} = f_{N-1}.$$

Решив систему из двух уравнений с двумя неизвестными, получаем $x_{N-1} = \frac{f_{N-1} - a\beta_{N-1}}{a\alpha_{N-1} + b}$. Остальные значения x_i находим в обратном порядке, используя формулу (2).

Соберем теперь все формулы прогонки и запишем их в порядке применения.

¹ Описание метода прогонки заимствовано из [1] и преобразовано для случая трехдиагональной матрицы, содержащей одинаковые элементы на каждой из диагоналей в отдельности.

Прямой ход:

$$\alpha_2 = -\frac{c}{b}, \alpha_{i+1} = -\frac{c}{a\alpha_i + b}, i = 2, \dots, N-2,$$

$$\beta_2 = \frac{f_1}{b}, \beta_{i+1} = \frac{f_i - a\beta_i}{a\alpha_i + b}, i = 2, \dots, N-2.$$

Обратный ход:

$$x_{N-1} = \frac{f_{N-1} - a\beta_{N-1}}{a\alpha_{N-1} + b}, x_i = \alpha_{i+1}x_{i+1} + \beta_{i+1}, i = N-2, \dots, 1.$$

4. Метод циклической редукции

Для решения СЛАУ с трехдиагональной матрицей наряду с методом прогонки применяются и другие методы, которые на практике часто оказываются более эффективными. Одним из таких методов является метод циклической редукции. Основное ограничение данного метода состоит в том, что он работает только в случаях, когда матрица имеет размерность, равную степени двух.

Вернемся к СЛАУ вида (1) и рассмотрим метод циклической редукции² применительно к указанной системе.

Введем в данную систему два фиктивных уравнения с парой фиктивных переменных x_0 и x_N : $x_0 = f_0 = 0$ и $x_N = f_N = 0$. Тогда систему (1) можно записать в виде (4).

$$\begin{bmatrix} a & b & c & & & & \\ & a & b & c & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & a & b & c & \\ & & & & a & b & c \end{bmatrix}_{(N-1) \times (N+1)} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}_{(N+1) \times 1} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}_{(N+1) \times 1} \quad (4)$$

Будем считать, что размерность системы (4) является степенью двух, т.е. $N = 2^q$, а значит, для ее решения можно использовать метод циклической редукции.

Смысл метода состоит в последовательном исключении переменных с нечетными индексами (*прямой ход редукции*) и обратном восстановлении

² Описание метода циклической редукции для общего случая трехдиагональных матриц содержится в [5].

значений нечетных переменных на основании известных значений переменных с четными номерами (*обратный ход*).

На каждой итерации прямого хода редукции рассматриваются тройки уравнений, неперекрывающихся по уравнениям с четными индексами (рис. 3). Из каждой такой тройки исключаются переменные с нечетными индексами, после чего переменные перенумеровываются и на следующей итерации снова исключаются переменные с нечетными индексами.

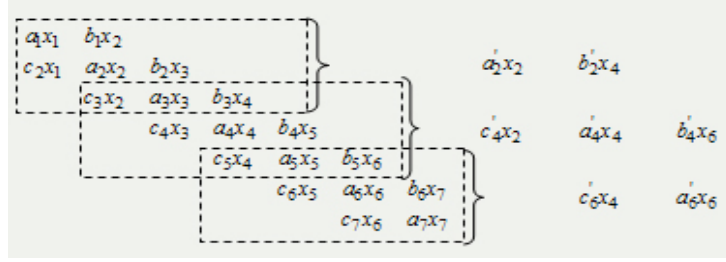


Рис. 3. Схема циклической редукции

На первой итерации алгоритма циклической редукции тройка рассматриваемых уравнений для системы (4) выглядит следующим образом:

$$\begin{cases} ax_{i-2} + bx_{i-1} + cx_i = f_{i-1} \\ ax_{i-1} + bx_i + cx_{i+1} = f_i, \text{ где } i = 2, 4, 6, \dots, N-2 \\ ax_i + bx_{i+1} + cx_{i+2} = f_{i+1} \end{cases} \quad (5)$$

В приведенной системе переменные x_{i-1} , x_{i+1} являются переменными с нечетными индексами согласно индексации. Выполним исключение этих переменных. Выразим переменную x_{i-1} из первого уравнения системы (5):

$$x_{i-1} = (f_{i-1} - ax_{i-2} - cx_i) \frac{1}{b}$$

Подставим полученное выражение во второе уравнение системы (5):

$$\begin{aligned} \frac{a}{b}(f_{i-1} - ax_{i-2} - cx_i) + bx_i + cx_{i+1} &= f_i \\ -\frac{a^2}{b}x_{i-2} + x_i\left(b - \frac{ac}{b}\right) + cx_{i+1} &= f_i - \frac{a}{b}f_{i-1} \\ -a^2x_{i-2} + x_i(b^2 - ac) + bcx_{i+1} &= bf_i - af_{i-1} \end{aligned} \quad (6)$$

Умножим уравнение (5.3) на c :

$$acx_i + bcx_{i+1} + c^2x_{i+2} = cf_{i+1} \quad (7)$$

Вычтем из (6) уравнение (7). В результате приведения подобных слагаемых получим уравнение вида:

$$-a^2x_{i-2} + (b^2 - 2ac)x_i - c^2x_{i+2} = bf_i - cf_{i+1} - af_{i-1} \quad (8)$$

Разделим обе части уравнения на b , введем обозначения $\alpha = -\frac{a}{b}, \beta = -\frac{c}{b}$. Получим систему уравнений (9).

$$\alpha a x_{i-2} + (b + 2\alpha c)x_i + \beta c x_{i+2} = \alpha f_{i-1} + f_i + \beta f_{i+1},$$

$$\text{где } i = 2, 4, 6, \dots, N - 2. \quad (9)$$

Система (9) структурно схожа с исходной системой уравнений (8), но полученная система имеет меньшую размерность. Перенумеровав переменные, можно выполнить следующую итерацию исключения переменных с нечетными номерами, повторив процедуру, аналогичную описанной. Таким образом, если получены коэффициенты, стоящие на диагонали системы на j -ой итерации, то на $(j + 1)$ -ой итерации они могут быть вычислены в соответствии с рекуррентными соотношениями, приведенными ниже.

$$a^{(0)} = a, b^{(0)} = b, c^{(0)} = c$$

$$a^{(1)} = \alpha a^{(0)}, b^{(1)} = b^{(0)} + 2\alpha c^{(0)}, c^{(1)} = -\beta c^{(0)}, \quad (10)$$

$$\text{где } \alpha = -\frac{a^{(0)}}{b^{(0)}}, \beta = -\frac{c^{(0)}}{b^{(0)}}.$$

$$a^{(j+1)} = \alpha a^{(j)}, b^{(j+1)} = b^{(j)} + 2\alpha c^{(j)}, c^{(j+1)} = -\beta c^{(j)}, \quad (11)$$

$$\text{где } \alpha = -\frac{a^{(j)}}{b^{(j)}}, \beta = -\frac{c^{(j)}}{b^{(j)}}, j = \overline{0, q-1}.$$

При этом правые части уравнений системы будут пересчитываться по формулам:

$$f_i^{(1)} = \alpha f_{i-1}^{(0)} + f_i^{(0)} + \beta f_{i+1}^{(0)}, i = 2, 4, 6 \dots N - 2,$$

$$\text{где } \alpha = -\frac{a^{(0)}}{b^{(0)}}, \beta = -\frac{c^{(0)}}{b^{(0)}}.$$

$$f_i^{(2)} = \alpha f_{i-2}^{(1)} + f_i^{(1)} + \beta f_{i+2}^{(1)}, i = 4, 8 \dots N - 4,$$

$$\text{где } \alpha = -\frac{a^{(1)}}{b^{(1)}}, \beta = -\frac{c^{(1)}}{b^{(1)}}.$$

$$f_i^{(j+1)} = \alpha f_{i-2^j}^{(j)} + f_i^{(j)} + \beta f_{i+2^j}^{(j)}, i = 2^j, 2^{j+1} \dots N - 2^j, \quad (12)$$

$$\text{где } \alpha = -\frac{a^{(j)}}{b^{(j)}}, \beta = -\frac{c^{(j)}}{b^{(j)}}, j = \overline{0, q-1}.$$

Схема исключения переменных для случая $N = 8$ приведена на рис. 4.

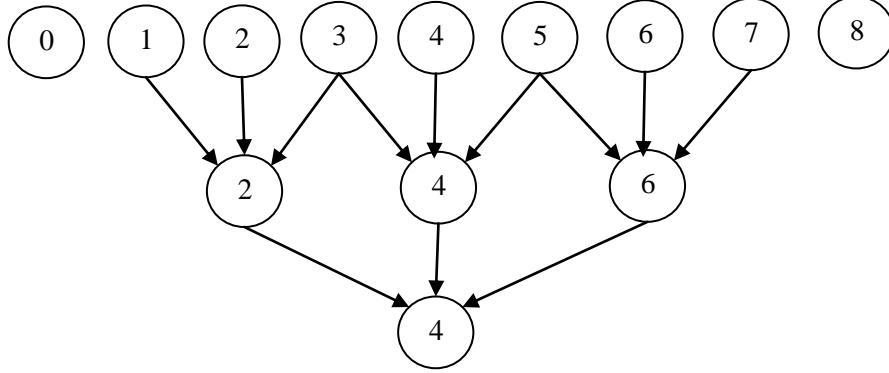


Рис. 4. Схема исключения переменных с нечетными номерами при $N = 8$

На последней $(q - 1)$ -ой итерации исключения переменных останется одно значимое уравнение с двумя фиктивными переменными.

$$x_0 = f_0 = 0$$

$$a^{(q-2)}x_0 + b^{(q-2)}x_{N/2} + c^{(q-2)}x_N = f_{N/2}^{(q-2)}$$

$$x_N = f_N = 0$$

Данное уравнение можно разрешить относительно переменной $x_{N/2}$:

$$x_{N/2} = \frac{f_{N/2}^{(q-2)} - a^{(q-2)}x_0 - c^{(q-2)}x_N}{b^{(q-2)}}$$

Таким образом, развертывается обратный ход редукции. На произвольной l -ой итерации можно восстановить переменную с нечетным индексом i , выразив ее из соответствующего уравнения (13) через известные переменные, полученные на предшествующем шаге обратного хода.

$$a^{(l)}x_{i-2^l} + b^{(l)}x_i + c^{(l)}x_{i+2^l} = f_i^{(l)} \quad (13)$$

$$x_i = \frac{f_i^{(l)} - a^{(l)}x_{i-2^l} - c^{(l)}x_{i+2^l}}{b^{(l)}}, i = 2^l, 2^{l+1} \dots N - 2^l$$

На рис. 5 приведена схема восстановления решения системы при $N = 8$. Согласно данной схеме переменные пересчитываются последовательно снизу вверх (пересчитываемые на каждом шаге переменные выделены, стрелками от них указаны переменные, значения которых используются при восстановлении согласно формуле (13)). Такая схема восстановления позволяет при пересчете правых частей уравнений в прямом ходе редукции затирать значения, полученные на предыдущем шаге.

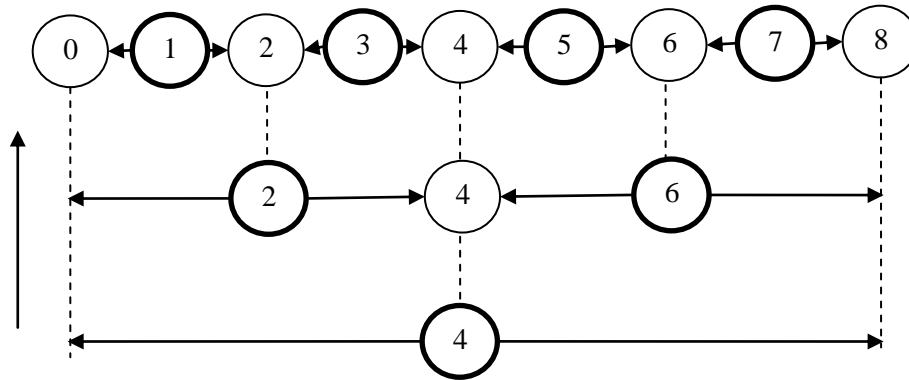


Рис. 5. Схема восстановления переменных на каждой итерации обратного хода редукции при $N = 8$

5. Программная реализация

5.1. Реализация метода прогонки

Перейдем к программной реализации рассмотренных выше методов. Прежде всего, в папке с лабораторными работами курса – `c:\ParallelCalculus` – откройте **Решение (Solution) 20_SweepAndReduction**, а в нем проект **01_Sweep**. Проект содержит файлы исходного кода **DiffEquation.cpp**, **main.cpp**, а также заголовочный файл **DiffEquation.h**. Рассмотрим имеющийся исходный код.

Начнем с файла **main.cpp**. Обратимся к функции **main()**. В основную функцию через аргумент командной строки будем передавать число разбиений J . Остальные параметры дифференциального уравнения принимаются равными некоторым константам, которые чаще всего используются при решении задачи вычисления цены составного опциона. Также для контроля правильности работы приложения все результаты решения прикладной задачи (цена составного опциона на начальной стадии для всех возможных значений цены акции и оптимальная цена составного опциона в каждый момент принятия решения о конвертации) будем записывать в файлы.

```
int main(int argc, char *argv[])
{
    // параметры дифференциального уравнения
    double T = 5.0;
    int m = 5;
    double sigma = 0.3;
    double D = 0.01;
    double K = 100.0;
```

```
double rk_s = 0.015;
double Ic = 6.0;
double rf = 0.0253;
int M = 100;
int n = 100;
int J;
// переменные для сохранения замеров времени
clock_t start, finish;
// дескрипторы файлов для сохранения результатов
FILE *fstockprice, *fcbprice;
// вспомогательная переменная, необходимая для
// перехода от переменной  $z_t$  к  $V_t$ 
double delta_z;
// переменные для сохранения результатов
double *stockprice, *cbprice;
if (argc < 2)
    return -1;
J = atoi(argv[1]);
start = clock();
// вычисление цены КО
getprices(T, m, sigma, D, K, rk_s, Ic, rf, M, J, n,
          &stockprice, &cbprice);
finish = clock();
// вычисление времени поиска оптимальной цены КО
duration = ((double)(finish-start)) /
            ((double)CLOCKS_PER_SEC);
printf("Time = %.5f\n", duration);
// создание файла для сохранения оптимальной цены КО
fstockprice = fopen("stockprice.csv", "w+");
if (fstockprice == NULL)
{
    printf("File wasn't created\n");
    free(stockprice);
    free(cbprice);
    return -1;
}
// создание файла для сохранения цены КО на начальной
// стадии при всех значениях стоимости акции
fcbprice = fopen("cbprice.csv", "w+");
if (fcbprice == NULL)
{
    printf("File wasn't created\n");
    free(stockprice);
    free(cbprice);
    return -1;
}
// запись оптимальной цены КО
for (i = 0; i < n; i++)
    fprintf(fstockprice, "%lf\n", stockprice[i]);
```

```

// запись стоимости акции и соответствующей цены КО
delta_z = (log(MAXPRICE) - log(MINPRICE)) / J;
for (i = 0; i < J + 1; i++)
    fprintf(fcbprice, "%lf;%lf\n",
        exp(log(MINPRICE) + i * delta_z), cbprice[i]);
// закрытие файлов
fclose(fstockprice);
fclose(fcbprice);
// освобождение памяти, выделенной под результаты
free(stockprice);
free(cbprice);
return 0;
}

```

Отметим, что в коде основной функции отсутствуют подключения необходимых библиотек – предоставляем читателю добавить их самостоятельно. Полужирным начертанием в коде выделена основная вычислительная функция **getprices**.

Перейдем к реализации метода прогонки. Поместим соответствующую функцию в файл **DiffEquation.cpp**.

Реализация метода прогонки для решения СЛАУ с трехдиагональной матрицей, элементы каждой диагонали которой одинаковы, **sweepmethod()**, показана ниже. В реализации выделены прямой и обратный ход. Описание метода было рассмотрено в § 3.

```

// a_0 - число на нижней побочной диагонали
// a_1 - число на главной диагонали
// a_2 - число на верхней побочной диагонали
int sweepmethod(double a_0, double a_1, double a_2,
    double *cb_next, int J, double *x)
{
    int size, i;
    double *alpha, *beta, denominator;
    size = J - 1;
    alpha = (double *)malloc(sizeof(double) * size);
    beta = (double *)malloc(sizeof(double) * size);
    // прямой ход метода прогонки
    alpha[1] = -a_2 / a_1;
    beta[1] = cb_next[0] / a_1;
    for (i = 1; i < size - 1; i++)
    {
        denominator = a_0 * alpha[i] + a_1;
        alpha[i+1] = -a_2 / denominator;
        beta[i+1] = (cb_next[i] - a_0 * beta[i]) / denominator;
    }
    // обратный ход метода прогонки
    x[size-1] = (-a_0 * beta[size-1] + cb_next[size-1]) /
        (a_1 + a_0 * alpha[size - 1]);
    for (i = size - 2; i >= 0; i--)

```

```

    x[i] = alpha[i + 1] * x[i + 1] + beta[i + 1];
    free(alpha);
    free(beta);
    return 0;
}

```

Наконец, осталось собрать проект, выполнив команду **Build→Rebuild 01_Sweep**. В случае отсутствия в программе синтаксических ошибок и ошибок линковки можно выполнить запуск приложения, передав в качестве параметра командной строки (**Project → Properties** **дерево Configuration Properties → Debugging→Command Arguments**) размерность разбиения по вертикальному измерению сетки.

Для проверки корректности полученной реализации возьмем $J = 8$, количество моментов принятия решения о конвертации $n = 100$ и размерность разбиения временного интервала между двумя последовательными моментами принятия решения $M = 100$. Остальные параметры дифференциального уравнения оставим равными значениям, установленным в начале функции **main()**. При таких параметрах результирующие значения цены СО (сохраняются в выходной файл с названием **cbprice.csv**) для каждого возможного значения курса акций в начальный момент времени должны совпадать в пределах погрешности с теми, что приведены в таблице 2.

Таблица 2. Цена СО при всех значениях цены акций на начальной стадии при $J = 8$

Курс акций	Цена составного опциона
0.010000	107.500000
0.056234	95.413130
0.316228	95.116675
1.778279	97.063120
10.000000	168.915385
56.234133	937.235542
316.227766	5270.462767
1778.279410	29637.990167
10000.000000	166672.666667

Для проверки корректности вычисления оптимальной цены СО на каждой стадии задайте размерности сетки, равными $J = 1024$, $n = 5$, $M = 100$. В результате выполнения программы с указанным набором параметров значения оптимальной цены СО в пределах машинной погрешности должны

быть равными 14.788423, 15.820447, 14.590242, 13.097473, 10.990454 (stockprice.csv).

5.2. Последовательная реализация метода циклической редукции

В рамках решения **20_SweepAndReduction** откройте проект с названием **02_Cycle**. Наряду с файлами **DiffEquation.cpp**, **DiffEquation.h**, **main.cpp**, в проекте имеются файлы **CycleReduction.h** и **CycleReduction.cpp**. В этих файлах будут размещаться объявление и реализация метода циклической редукции.

Реализацию метода циклической редукции, описанного в § 4, рассмотрим в виде псевдокода. Выделяются два этапа – прямой и обратный ход редукции. На первом этапе (строки 1 – 22) осуществляется исключение переменных с нечетными номерами и пересчет диагональных коэффициентов. На втором этапе (строки 23 – 34) выполняется последовательное восстановление решения трехдиагональной системы. Необходимо отметить, что предлагаемый псевдокод, как и описанный метод, ориентирован на случай матрицы, у которой каждая диагональ в отдельности содержит одинаковые элементы.

```
// прямой ход циклической редукции
1. a[0] ← a_0;
2. b[0] ← a_1;
3. c[0] ← a_2;
4. f[0] ← 0;
5. f[n] ← 0;
6. x[0] ← 0;
7. x[n] ← 0;
8. start ← 2;
9. elementsNum ← n;
10. step ← 1;
11. for j = 0 to q - 1
12.   alpha ← -a[j] / b[j];
13.   beta ← -c[j] / b[j];
14.   a[j + 1] ← alpha * a[j];
15.   b[j + 1] ← b[j] + 2 * alpha * c[j];
16.   c[j + 1] ← beta * c[j];
17.   elementsNum ← (elementsNum - 1) / 2;
18.   for i = 0 to elementsNum
19.     k ← start * (i + 1);
20.     f[k] ← alpha*f[k-step] + f[k] + beta*f[k+step];
21.   start ← 2 * start;
22.   step ← 2 * step;

// обратный ход циклической редукции
23. start ← n / 2;
24. step ← start;
```

```

25.elementsNum ← 1;
26.for j = q - 1 to 0
27.  alpha ← -a[j] / b[j];
28.  beta ← -c[j] / b[j];
29.  for i = 0 to elementsNum
30.    k ← start * (2 * i + 1);
31.    x[k] ← f[k]/b[j]+alpha*x[k-step]+beta*x[k+step];
32.  start ← start / 2;
33.  step ← start;
34.  elementsNum ← elementsNum * 2;

```

Предлагаем читателю разработать на основании представленного псевдокода функцию, содержащую реализацию алгоритма циклической редукции для решения трехдиагональной системы вида $Ax = f$, рассмотренной в процессе описания метода. Возможный прототип данной функции представлен ниже.

```

// x - вектор, в который сохраняется решение системы
// a_0, a_1, a_2 - коэффициенты, стоящие на диагоналях
//               в порядке нижняя побочная, главная,
//               верхняя побочная диагонали
// a, b, c - массивы обновленных диагональных элементов
//           матрицы на каждом шаге исключения переменных
//           (используются при восстановлении решения)
// f - правая часть системы уравнений
// n - индекс последней переменной в системе
// q - степень двойки  $n = 2^q$ 
int CycleReductionMethod(double *x, double a_0, double a_1,
    double a_2, double *a, double *b, double *c, double *f,
    int n, int q);

```

После того, как разработана программная реализация алгоритма циклической редукции, скомпилируйте проект, выполнив команду **Build→Rebuild 02_Cycle**. Убедитесь в корректности результатов работы программы. Для этого воспользуйтесь данными, приведенными в § 5.1 (в частности, в табл. 2).

5.3. Параллельная реализация метода циклической редукции с использованием технологии OpenMP

Прежде, чем переходить к параллельной реализации алгоритма циклической редукции для систем с общей памятью с использованием технологии OpenMP [7], создадим в рамках решения **20_SweepAndReduction** новый проект с названием **03_CycleOmp**. Скопируйте в созданный проект все заголовочные файлы и файлы исходного кода, содержащиеся в проекте **02_Cycle**.

Теперь настроим в свойствах проекта возможность использования технологии OpenMP. Откройте свойства проекта, выполнив команду **Project**→**Properties**. В дереве **Configuration Properties** перейдите к разделу **C/C++**→**Language** и в поле **OpenMP Support** справа выберите вариант: **Generate Parallel Code (/openmp, equiv. to /Qopenmp)**.

Рассмотрим схемы исключения и восстановления переменных, показанные на рис. 4 и рис. 5. Очевидно, что каждая следующая итерация прямого и обратного хода редукции зависит от предыдущей. С другой стороны, исключение или восстановление каждой конкретной переменной на отдельной итерации можно проводить независимо, то есть, можно распараллелить выполнение вложенных циклов (строки 18 и 29 псевдокода алгоритма циклической редукции). Таким образом, для получения OpenMP-версии достаточно вставить директиву библиотеки OpenMP **pragma omp parallel for** [7] перед соответствующими циклами. Ниже приведен фрагмент кода для вложенного цикла прямого хода редукции. В нашем случае итерации циклов не имеют зависимостей по данным, поэтому эффектов, связанных с доступом потоков к одинаковым областям памяти не возникнет.

```
// THREADSNUM - константа, определяющая количество потоков
#pragma omp parallel for num_threads(THREADSNUM)
for (i = 0; i < elementsNum; i++)
{
    int k = start * (i + 1);
    f[k] = alpha*f[k-step] + f[k] + beta*f[k+step];
}
```

Осталось убедиться в корректности полученной параллельной реализации алгоритма циклической редукции. Задайте количество потоков (THREADSNUM), равное двум. Далее скомпилируйте проект, выполнив команду **Build**→**Rebuild 03_CycleParallel**, и воспользуйтесь данными, приведенными в § 5.1 (в частности, в табл. 2). Если получены результаты, отличные от корректных, то убедитесь в отсутствии «гонок» данных. Для этого можно воспользоваться, например, инструментом Intel Parallel Inspector (или Intel Parallel Inspector XE) в режиме Threading Errors (Threading Error Analysis в Intel Parallel Inspector XE).

5.4. Параллельная реализация метода циклической редукции с использованием библиотеки Intel Threading Building Blocks

Как и ранее, перед выполнением параллельной реализации алгоритма циклической редукции для систем с общей памятью с помощью библиотеки Intel TBB [8, 9] создадим в рамках решения **20_SweepAndReduction** новый проект с названием **04_CycleTBB**. Скопируйте файлы **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**, **CycleReduction.h**, **CycleReduction.cpp**

из проекта **02_Cycle**. Создайте дополнительно заголовочный файл **tbb.h** и файл исходного кода **tbb.cpp**. В дальнейшем в этих файлах будут размещены объявление и реализация функций и классов, необходимых для организации параллелизма с помощью TBB.

Чтобы подключить библиотеку TBB к проекту, требуется изменить настройки проекта:

1. указать путь до заголовочных файлов библиотеки (**Configuration Properties**→**C/C++**→**General**→**Additional Include Directories**),
2. указать путь до **.lib** файлов библиотеки (**Configuration Properties**→**Linker**→**General**→**Additional Library Directories**),
3. указать библиотеку **tbb.lib** (**Configuration Properties**→**Linker**→**Input**→**Additional Dependencies**), с которой должен собираться проект.

Для использования возможностей TBB по распараллеливанию вычислений необходимо иметь хотя бы один активный (инициализированный) экземпляр класса **tbb::task_scheduler_init**. Этот класс предназначен для создания потоков и внутренних структур, необходимых планировщику потоков. Внесем инициализацию объекта данного класса в основную функцию текущего проекта. Необходимые модификации выделены полужирным начертанием.

```
...
#include "tbb/task_scheduler_init.h"
#define TBB_NUM_THREADS 2
int main(int argc, char **argv)
{
    ...
    tbb::task_scheduler_init
        init(tbb::task_scheduler_init::deferred);
    init.initialize(TBB_NUM_THREADS);
    start = clock();
    // compute CB and stock prices
    getprices(T, m, sigma, D, K, rk_s, Ic, rf, M,
              J, n, &stockprice, &cbprice);
    finish = clock();
    init.terminate();
    duration = ((double)(finish - start)) /
               ((double)CLOCKS_PER_SEC);
    ...
}
```

Приступим к разработке параллельной реализации. Как было отмечено в предыдущем разделе, распараллеливание можно проводить только на уровне вложенных циклов редукции, в которых происходит исключение и

восстановление переменных. В библиотеке ТВВ для распараллеливания циклов с известным числом повторений, используется функция **tbb::parallel_for()**, которая в качестве входных параметров принимает итерационное пространство цикла и объект класса-функтора. В данной задаче можно использовать встроенное одномерное итерационное пространство **tbb::blocked_range**. Класс-функтор фактически представляет собой развернутое тело цикла.

Выделим две функции, в которых будет организован вызов функции **tbb::parallel_for()**. Эти функции в качестве параметров принимают переменные, используемые в теле вложенного цикла. Объявление указанных функций и подключение необходимых заголовочных файлов библиотеки ТВВ разместим в файле **tbb.h**.

```
#ifndef _TBB_H
#define _TBB_H

#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
#include "tbb/partitioner.h"

void fcomp(double *f, int elementsNum, double alpha,
           double beta, int step, int start);
void fcompreverse(double *x, int elementsNum, double *f,
                  double bj, double alpha, double beta, int step,
                  int start);

#endif
```

Осталось разработать классы-функторы для прямого и обратного хода редукции, а также реализовать функции **fcomp()** и **fcompreverse()**. Реализацию классов и функций поместим в файл **tbb.cpp**.

Сначала рассмотрим класс-функтор для распараллеливания прямого хода редукции, **FFCompFuncutor**, и функцию **fcomp()**, в которой вызывается **tbb::parallel_for()** для параллельного исключения переменных системы. Поля класса-функтора названы в соответствии с названиями переменных, объявленных в реализации метода циклической редукции, поэтому здесь мы не будем останавливаться на объяснении их смысла.

```
class FFCompFuncutor
{
private:
    double *f;
    double alpha;
    double beta;
    int step;
    int start;
public:
    FFCompFuncutor(double *_f, const double _alpha,
```

```

    const double _beta, const int _step, const int _start)
    {
        f = _f;
        alpha = _alpha;
        beta = _beta;
        step = _step;
        start = _start;
    }

    void operator() (const tbb::blocked_range<int>&r) const
    {
        int i, k;
        for (i = r.begin(); i < r.end(); i++)
        {
            k = start * (i + 1);
            f[k] = alpha * f[k - step] + f[k] +
                    beta * f[k + step];
        }
    }
};

void fcomp(double *f, int elementsNum, double alpha,
           double beta, int step, int start)
{
    tbb::parallel_for<tbb::blocked_range<int>, FFCompFunctor>
        (tbb::blocked_range<int>(0, elementsNum),
         FFCompFunctor(f, alpha, beta, step, start));
}

```

Теперь приведем реализации класса-функтора для расплеливания обратного хода редукции, **FFCompreverseFunctor**, и функции **fcompreverse()** параллельного восстановления переменных.

```

class FFCompreverseFunctor
{
private:
    double *x;
    double *f;
    double bj;
    double alpha;
    double beta;
    int step;
    int start;
public:
    FFCompreverseFunctor(double *_x, double *_f,
        const double _bj, const double _alpha,
        const double _beta, const int _step, const int _start)
    {
        x = _x;
    }
}

```

```

    f = _f;
    bj = _bj;
    alpha = _alpha;
    beta = _beta;
    step = _step;
    start = _start;
}
void operator()(tbb::blocked_range<int>&r) const
{
    int i, k;
    for (i = r.begin(); i < r.end(); i++)
    {
        k = start * (2 * i + 1);
        x[k] = f[k] / bj + alpha * x[k - step] +
              beta * x[k + step];
    }
}
};

void fcompreverse(double *x, int elementsNum, double *f,
    double bj, double alpha, double beta, int step,
    int start)
{
    tbb::parallel_for<tbb::blocked_range<int>,
        FFCompreverseFunctor>
        (tbb::blocked_range<int>(0, elementsNum),
        FFCompreverseFunctor(x, f, bj, alpha, beta, step, start));
}

```

Перейдем к интеграции разработанной параллельной реализации в функцию, содержащую метод циклической редукции. Предварительно требуется подключить заголовочный файл **tbb.h** в заголовочный файл с объявлением метода циклической редукции **CycleReduction.h**. Затем необходимо заменить внутренние циклы прямого и обратного хода редукции вызовом функций **fcomp** и **fcompreverse** соответственно (в программном коде изменения выделены полужирным).

```

int CycleReductionMethod(double *x, double a_0, double a_1,
    double a_2, double *a, double *b, double *c,
    double *f, int n, int q)
{
    ...
    for (j = 0; j < q - 1; j++)
    {
        alpha = -a[j] / b[j];
        beta = -c[j] / b[j];
        a[j + 1] = alpha * a[j];
        b[j + 1] = b[j] + 2 * alpha * c[j];
        c[j + 1] = beta * c[j];
        elementsNum = (elementsNum - 1) / 2;
    }
}

```

```

    fcomp(f, elementsNum, alpha, beta, step, start);
    start *= 2;
    step  *= 2;
}
start = n / 2;
step  = start;
elementsNum = 1;
for (j = q - 1; j >= 0; j--)
{
    alpha = -a[j] / b[j];
    beta  = -c[j] / b[j];
    fcompreverse(x, elementsNum, f, b[j],
                 alpha, beta, step, start);

    start /= 2;
    step  = start;
    elementsNum *= 2;
}
...
}

```

Теперь, как и в случае OpenMP-версии, необходимо убедиться в корректности полученной параллельной реализации алгоритма циклической редукции. Задайте количество потоков, равным двум. Далее скомпилируйте проект, выполнив команду **Build→Rebuild 04_CycleTBB**, и воспользуйтесь данными, приведенными в § 5.1 (в частности, в табл. 2).

6. Анализ производительности приложения при использовании последовательных реализаций методов прогонки и циклической редукции

Задача вычисления цены СО была сведена к решению последовательности СЛАУ с трехдиагональной матрицей. Поэтому основная вычислительная сложность реализованной схемы определяется трудоемкостью используемых алгоритмов для решения таких СЛАУ. Выполним сравнение эффективности методов прогонки и циклической редукции при реализации вычислительной схемы Кранка-Николсона.

Будем считать, что трехдиагональная матрица, полученная при решении уравнения в частных производных, имеет размерность $J * J$.

Рассмотрим теоретическую оценку трудоемкости метода прогонки [1]. Сложность данного метода складывается из количества операций при выполнении прямого и обратного хода. Прямой ход предполагает определение значений коэффициентов в соответствии с формулами (2). Суммарное количество операций прямого хода составляет $(1 + 3(J - 2)) +$

$(1 + 5(J - 1)) = 8J - 14$. В процессе выполнения обратного хода вычисляется решение СЛАУ согласно (3). Подсчитав количество операций, получаем $5 + 2(J - 1) = 2J + 3$. Таким образом, построение решения СЛАУ с трехдиагональной матрицей методом прогонки требует $10J + O(1)$ операций. Трудоемкость метода не зависит от вида трехдиагональной матрицы.

При реализации метода циклической редукции для общего случая трехдиагональной матрицы размерности $J * J$, где $J = 2^q$, требуется $12J$ сложений, $8J$ умножений и $3J$ делений [10]. Определим количество выполняемых операций для частного случая трехдиагональной матрицы, рассматриваемого в данной лабораторной работе. Очевидно, что асимптотика должна сохраниться, а постоянный коэффициент перед размерностью матрицы – уменьшиться. Аналогично методу прогонки будем вычислять трудоемкость прямого и обратного хода в отдельности.

Прямой ход включает в себя q итераций, на каждой из которых выполняется вычисление обновленных коэффициентов матрицы согласно формулам (10), (11) и пересчет правых частей системы в соответствии с выражениями (12). Таким образом, обновление коэффициентов системы требует $q(2 + 4 + 2) = 8q = 8 \log_2 J$ операций (2 операции на вычисление $a^{(j)}$, 4 – на вычисление $b^{(j)}$ и 2 для $c^{(j)}$, $j = 0, 1, \dots, q - 1$). Определение одного нового значения правой части выполняется за 6 операций. На первой итерации прямого хода редукции осуществляется пересчет $\frac{J-1}{2}$ правых частей, на второй – $\frac{J-1}{4}$ правых частей, на третьей – $\frac{J-1}{8}$, и т.д. На последней итерации пересчитывается $\frac{J-1}{2^q}$ правых частей. Суммируя количество вычисляемых правых частей системы и умножая результат на число операций, требуемых для пересчета одной правой части, получаем оценку $6(J - 1) \left(\frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^q} \right) = \frac{6(J-1) \frac{1}{2} \left(1 - \frac{1}{2^q} \right)}{1 - \frac{1}{2}} = \frac{6(J-1)^2}{J}$, в асимптотике $6J + O(1)$. В итоге при реализации прямого хода редукции выполняется $6J + O(\log J)$ операций.

Обратный ход предполагает восстановление переменных из уравнения (13). Восстановление одной переменной требует 5 операций. В системе таких переменных J . В целом на реализацию обратного хода затрачивается $5J$ операций.

Таким образом, оценка суммарного числа операций метода циклической редукции в случае трехдиагональной матрицы, каждая диагональ которой представляется одним числом, составляет $11J + O(\log J)$, что несколько хуже, чем в случае с методом прогонки (получаем большее значение константы).

Теперь перейдем к результатам экспериментов. В табл. 3 приведены данные по методам прогонки и циклической редукции на разных размерностях

матрицы, а на рис. 6 показана зависимость времени поиска оптимальной цены СО от размерности разбиения по вертикали, которое и определяет размерность трехдиагональной матрицы в СЛАУ.

Таблица 3. Время вычисления цены СО и оптимальной цены СО с использованием методов прогонки и циклической редукции для решения трехдиагональной системы

J	Время работы реализации, использующей метод прогонки (сек)	Время работы реализации, использующей метод циклической редукции (сек)
256	0.202	0.031
512	0.405	0.078
1024	0.795	0.14
2048	0.53	0.265
4096	1.045	0.514
8192	2.074	1.092
16384	4.227	2.169
32768	8.47	4.399
65536	19.11	9.111

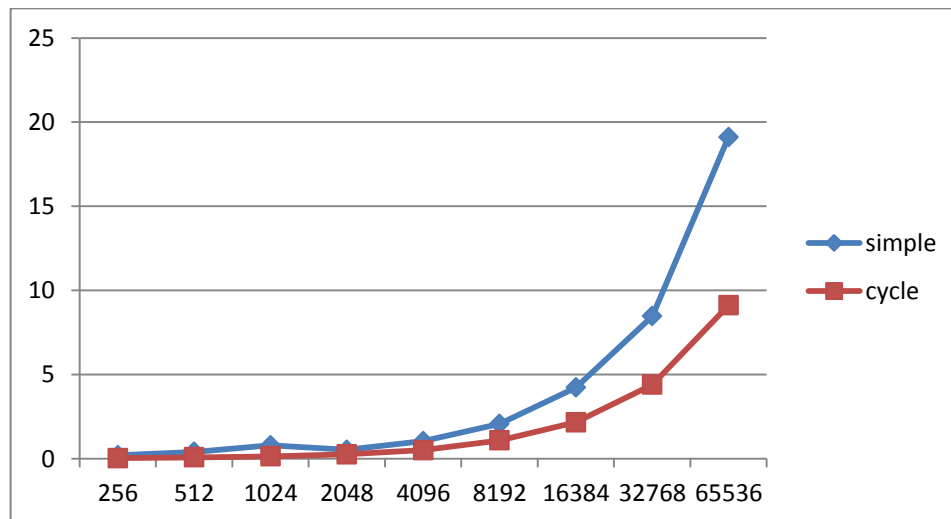


Рис. 6. Время поиска цены СО с использованием последовательных реализаций метода прогонки (simple) и метода циклической редукции (cycle)

Как видим, численные результаты противоречат теоретическим. Время решения задачи с помощью метода прогонки на всех размерностях матрицы более чем в 2 раза превышает время решения с использованием циклической редукции. Одно из возможных объяснений данного факта связано с тем, что, несмотря на константу, в асимптотике число операций для обоих методов пропорционально $O(J)$. В то же время в зависимости от способа реализации коэффициенты при размерности матрицы в оценках числа операций могут изменяться. Например, для метода прогонки значение коэффициента можно уменьшить, если знаменатель для α_{i+1} и β_{i+1} вычислять только один раз. Поэтому при сравнении результатов правильнее рассматривать количество операций, выполняемых в программной реализации. Подсчитаем число операций для разработанных реализаций метода прогонки и редукции. Такое несложное задание предлагаем выполнить самостоятельно. Здесь мы приведем только результат. Для метода прогонки оценка составляет примерно $8J$, а для редукции – $9J$. Таким образом, исходное предположение не объясняет полученных результатов. Еще одна гипотеза связана с тем, что существенное влияние на результат оказывает архитектура, на которой проводились эксперименты. Воспользуемся инструментом Intel Parallel Amplifier XE. Определим значение числа тактов, приходящихся на исполнение одной инструкции, для функций прогонки и циклической редукции. Для этого необходимо гарантировать, что компилятор не сделает их inline-функциями. Укажем компилятору явно при вызове функций **sweepmethod** и **CycleReduction** прагму **pragma noline**. Запустим Amplifier в режиме **LightWeight Hostspots** (слева в поддереве режимов анализа **Algorithm Analysis** – см. рис. 7). Из полученных результатов (рис. 8 и рис. 9) видно, что в среднем операция метода прогонки выполняется почти за 2 такта, операция метода циклической редукции примерно за 1 такт, что для высокопроизводительных приложений является нормой. Отметим, что при этом количество инструкций в редукции приблизительно в 1.3 раза больше. Поскольку СЛАУ решается многократно, то время определения оптимальной цены СО при использовании редукции меньше, чем при использовании метода прогонки.

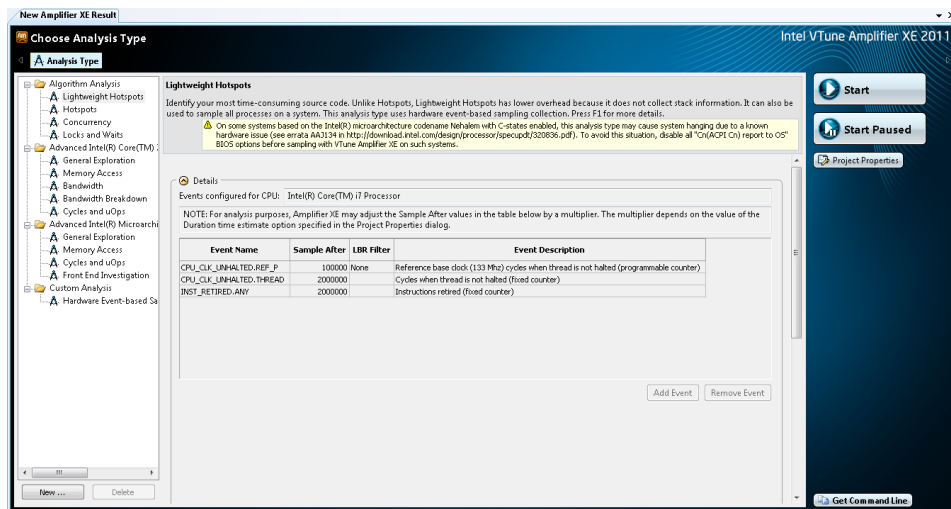


Рис. 7. Главное окно инструмента Intel Parallel Amplifier XE

/Function	CPU Time	Instructions Retired	CPI	Module	Function (Full)
sweepmethod	1.738s	1,788,000,000	2.377	01_sweep.exe	sweepmethod
crancknikolson	0.096s	498,000,000	0.514	01_sweep.exe	crancknikolson
__svml_exp2	0.011s	34,000,000	0.647	01_sweep.exe	__svml_exp2
__svml_exp2	0.002s	2,000,000	1.000	01_sweep.exe	__svml_exp2

Рис. 8. Результаты Lightweight Hotspots анализа при использовании метода прогонки для решения СЛАУ

/Function	CPU Time	Instructions Retired	CPI	Module	Function (Full)
CycleReductionMethod	975.940ms	2,312,000,000	1.036	02_cycle.exe	CycleReductionMethod
crancknikolson	114.286ms	514,000,000	0.595	02_cycle.exe	crancknikolson
__svml_exp2	9.023ms	34,000,000	1.118	02_cycle.exe	__svml_exp2
__svml_exp2	0.752ms	2,000,000	0.000	02_cycle.exe	__svml_exp2

Рис. 9. Результаты Lightweight Hotspots анализа при использовании метода циклической редукции для решения СЛАУ

Подводя итог, можно сделать вывод, что правильный выбор алгоритма для решения задачи позволяет в разы уменьшить общее время работы приложения.

7. Анализ масштабируемости приложения при использовании OpenMP-реализации метода циклической редукции

Теперь выполним анализ масштабируемости вычислительной схемы, в которой в качестве решателя СЛАУ используется OpenMP-реализация метода циклической редукции.

На рис. 10 показаны зависимости времени работы приложения с использованием циклической редукции в 1 (cycle_1), 2 (cycle_2), 4 (cycle_4) и 8 (cycle_8) потоков. В табл. 4 приведены численные результаты эксперимента. Первый столбец содержит время работы однопоточной версии, второй и последующие – время работы соответствующей многопоточной реализации и ускорение относительно однопоточной.

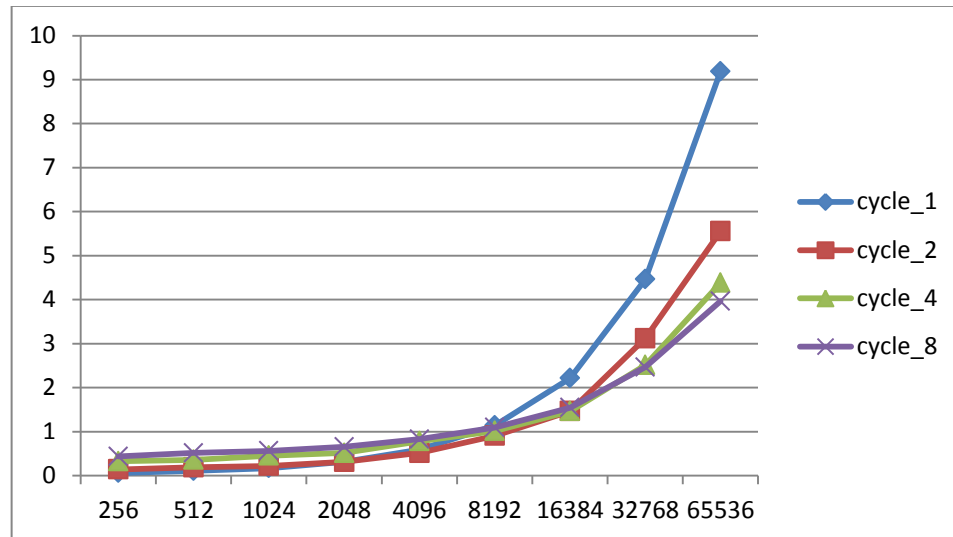


Рис. 10. Время решения задачи поиска оптимальной цены СО с использованием параллельной версии циклической редукции (cycle_1 – в 1 поток, cycle_2 – в 2 потока, cycle_4 – в 4 потока, cycle_8 – в 8 потоков)

Таблица 4. Результаты экспериментов с использованием параллельной реализации метода циклической редукции в разное количество потоков

J	1 поток	2 потока		4 потока		8 потоков	
	t,сек	t,сек	S	t,сек	S	t,сек	S
256	0.062	0.141	0.439	0.327	0.189	0.437	0.141
512	0.109	0.187	0.582	0.358	0.304	0.514	0.212

1024	0.171	0.291	0,587	0.453	0.377	0.561	0.304
2048	0.312	0.312	1	0.515	0.605	0.655	0.476
4096	0.592	0.514	1.151	0.78	0.758	0.827	0.715
8192	1.138	0.905	1.257	1.014	1.122	1.092	1.042
16384	2.215	1.466	1.51	1.466	1.51	1.545	1.433
32768	4.461	3.12	1.429	2.512	1.775	2.465	1.809
65536	9.188	5.554	1.654	4.383	2.096	3.962	2.319

Представленные результаты экспериментов свидетельствуют о плохой масштабируемости приложения, т.к. на 4 и на 8 потоках в лучшем случае ускорение составляет немногим более двух.

Для объяснения полученных результатов сначала оценим степень параллелизма разработанного приложения с помощью инструмента Intel Parallel Amplifier XE. Обращаем внимание, что при проведении анализа собирать необходимо Release-версию проекта **03_CycleParallel**. Задайте число потоков, равным 8 (максимальное количество ядер на узле тестовой инфраструктуры). Для этого необходимо изменить значение константы **THREADSNUM**, объявленной в файле **CycleReduction.h**. Установите число разбиений **J=256**. В настройках проекта **Configuration Properties**→**Debugging**→**Command Arguments** (рис. 11) задайте параметр командной строки 256.

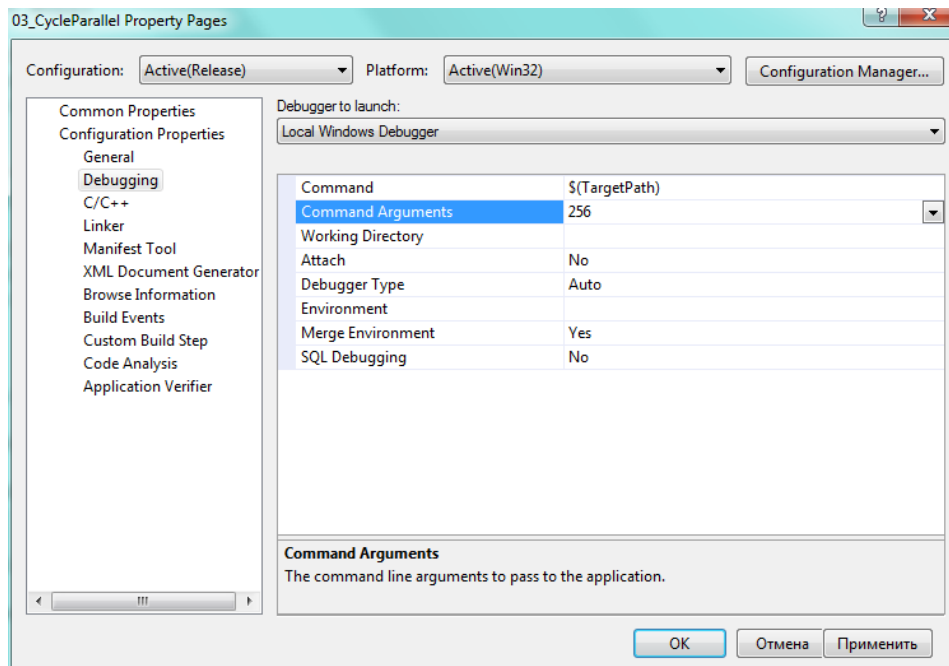


Рис. 11. Окно задания параметров командной строки в настройках проекта Visual Studio

Далее выберите режим анализа в дереве в левой части окна **Algorithm Analysis**→**Concurrency** и нажмите кнопку **Start** (рис. 7). Процедура сбора статистики занимает некоторое время.

На рис. 12 показаны результаты запуска инструмента на тестовой инфраструктуре, описанной в § 1.3.

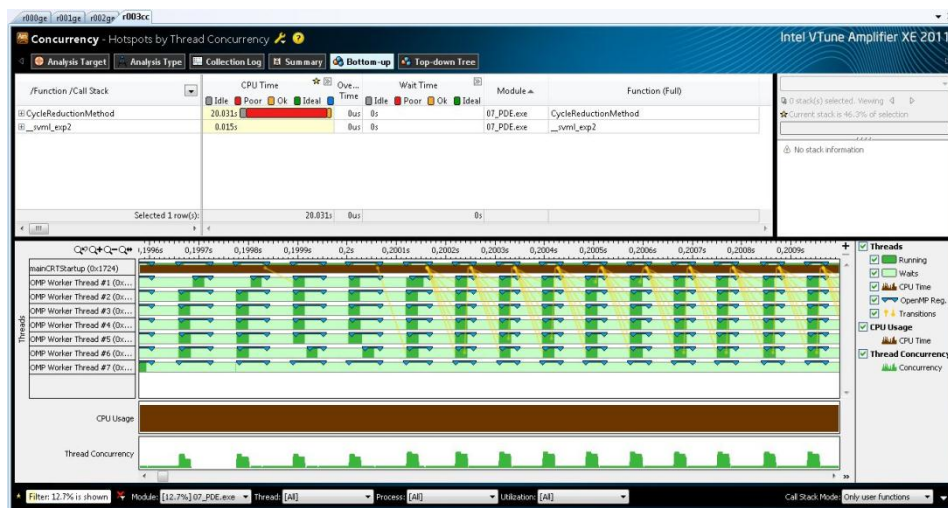


Рис. 12. Результаты запуска Intel Parallel Amplifier XE

Для каждой распараллеленной функции построена временная шкала, которая отражает полноту использования процессорных ресурсов при выполнении программы. В разработанном приложении единственной такой функцией является реализация метода циклической редукции (**CycleReductionMethod**). Из рисунка можно сделать вывод, что практически все время работы функции используются не все предоставляемые ресурсы, т.е. значительную часть времени программа работает в 1 поток (об этом свидетельствует наличие красного цвета на шкале). Если посмотреть на диаграмму активности потоков в процессе исполнения программы, то можно видеть наличие большого количества коротких параллельных секций (участки зеленого цвета). Они, очевидно, возникают из-за того, что распараллеливание выполнено на уровне внутреннего цикла прямого и обратного хода редукции, т.е. на каждой итерации редукции порождается или возобновляется (желтые стрелки на диаграмме активности) несколько потоков, выполняется ожидание их завершения (фактически, точка синхронизации), после чего главный поток продолжает последовательные вычисления. Таким образом, значительное влияние на время работы программы оказывают накладные расходы, связанные с организацией параллелизма.

Если обратиться к параллельной реализации метода циклической редукции и посмотреть на нее с точки зрения работы с данными, то можно предположить, что отсутствие масштабируемости также связано с неэффективной организацией работы с памятью. Т.к. пересчет правых частей СЛАУ и восстановление решения в методе осуществляется не последовательно, а с некоторым регулярным шагом на каждой итерации прямого и обратного хода редукции, это может приводить к многочисленным кэш-промахам при увеличении числа потоков. Для подтверждения данной гипотезы необходимо определить количество кэш-промахов, возникающих в однопоточной и многопоточной версиях при фиксированных параметрах задачи. Для этого снова воспользуемся инструментом Intel Parallel Amplifier XE. Сначала создадим тип анализа, включающий необходимые счетчики, выполнив в папке **Custom Analysis** команду **New Hardware Event-based Sampling Analysis**. В результате получим новый тип анализа, не содержащий никаких событий (рис. 13).

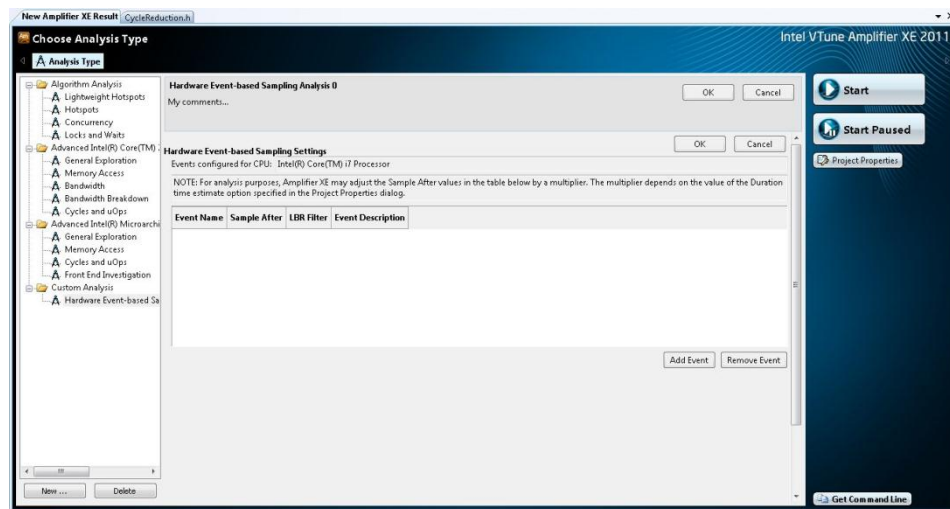


Рис. 13. Окно создания нового типа анализа программного приложения

Далее добавим интересные события. После нажатия кнопки **Add Event** будет сформирован список допустимых событий, из которых необходимо выбрать L2_RQSTS.MISS и MEM_LOAD_RETIREDD.LLC_MISS (рис. 14).

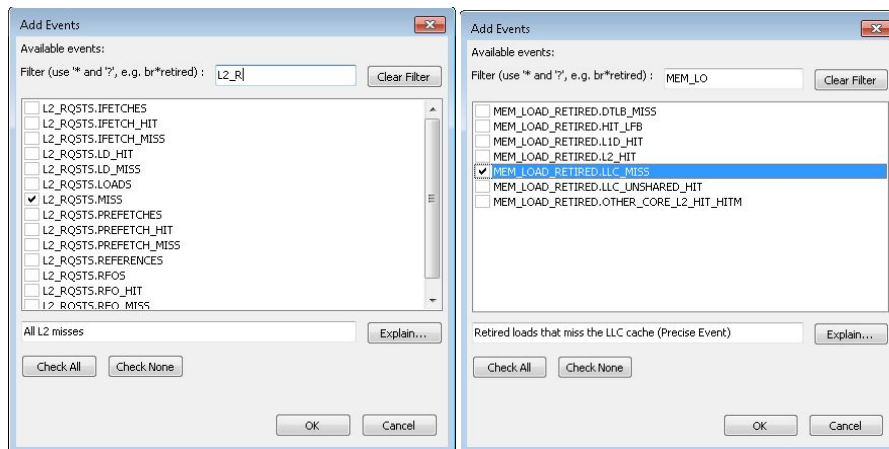


Рис. 14. Выбор интересных событий

В результате, в таблице событий (см. рис. 13) можно будет увидеть выбранные события и краткое их описание. Первый тип событий позволяет в процессе анализа определить количество L2 кэш-промахов, второй – количество загрузок, которые приводят к промахам в кэш последнего уровня. Обратите внимание на сформированную таблицу событий (рис. 15), а именно на второй столбец (**Sample After**). Данный столбец содержит пороговые значения счетчиков. Если в процессе профилировки приложения значение счетчика меньше указанного порога, то это значение принимается

равным нулю, в противном случае, отображается разница полученного значения счетчика и соответствующего порога. При необходимости пороговые значения можно изменить, редактируя ячейку таблицы.

Event Name	Sample After	LBR Filter	Event Description
L2_RQSTS.LOADS	200000	None	L2 requests
L2_RQSTS.MISS	200000	None	All L2 misses
MEM_LOAD_RETIRED.LLC_MISS	10000	None	Retired loads that miss the LLC cache (Precise Event)

Рис. 15. Список добавленных событий в сценарий профилировки

Теперь необходимо выполнить анализ созданного типа для однопоточной и 8-поточной версий приложения. Запуск анализа выполняется нажатием на кнопку **Start**.

На рис. 16 и рис. 17 показаны результирующие значения выбранных счетчиков для однопоточной и 8-поточной версий. Видно, что наличие нескольких потоков, приводит к конкуренции за использование кэш-памяти, что плохо сказывается на масштабируемости приложения.

/Function	PMU Event Count			Module	Function (Full)
	CPU_CLK_UNHALTED.THREAD	L2_RQSTS.MISS	MEM_LOAD_RETIRED.LLC_MISS		
CycleReductionMethod	80,000,000	0	0	0 07_pde.exe	CycleReductionMethod
__svml_exp2	10,000,000	0	0	0 07_pde.exe	__svml_exp2
crancknikolson	4,000,000	0	0	0 07_pde.exe	crancknikolson

Рис. 16. Результаты запуска Intel Parallel Amplifier XE для однопоточной версии

/Function	PMU Event Count			Module	Function (Full)
	CPU_CLK_UNHALTED.THREAD	L2_RQSTS.MISS	MEM_LOAD_RETIRED.LLC_MISS		
CycleReductionMethod	514,000,000	13,400,000	1,220,000	07_pde.exe	CycleReductionMethod
crancknikolson	40,000,000	600,000	10,000	07_pde.exe	crancknikolson
__svml_exp2	4,000,000	0	0	07_pde.exe	__svml_exp2
[Import thunk __kmpc_push_num_threads]	2,000,000	0	0	07_pde.exe	[Import thunk __kmpc_push_num_threads]
[Import thunk __kmpc_for_static_fini]	0	200,000	0	07_pde.exe	[Import thunk __kmpc_for_static_fini]

Рис. 17. Результаты запуска Intel Parallel Amplifier XE для 8-поточной версии

Приведенные факты говорят о том, что выполненная реализация схемы Кранка-Николсона с использованием параллельного метода циклической редукции плохо масштабируется. Главным образом это объясняется алгоритмическими особенностями метода редукции. Поэтому необходимо либо разрабатывать более эффективные схемы распараллеливания циклической редукции, что нетривиально из-за зависимости итераций прямого и обратного хода, либо использовать другие методы решения СЛАУ, параллельная реализация которых имела бы лучшую масштабируемость по сравнению с циклической редукцией.

8. Анализ масштабируемости приложения при использовании ТВВ-реализации метода циклической редукции

Последнее, что мы рассмотрим в рамках данной лабораторной работы – анализ масштабируемости схемы Кранка-Николсона, в которой в качестве решателя СЛАУ используется ТВВ-реализация метода циклической редукции.

На рис. 18 показаны зависимости времени работы приложения с использованием циклической редукции в 1 (tbb_1), 2 (tbb_2), 4 (tbb_4) и 8 (tbb_8) потоков. В табл. 5 приведены численные результаты эксперимента. Первый столбец содержит время работы однопоточной версии, второй и последующие – время работы соответствующей многопоточной реализации и ускорение относительно однопоточной.

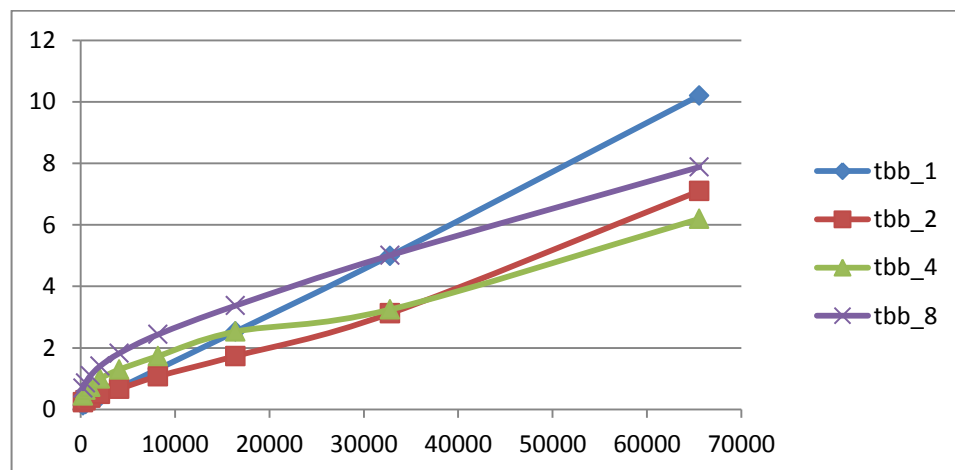


Рис. 18. Время решения задачи поиска оптимальной цены СО с использованием параллельной версии циклической редукции

Таблица 5. Результаты экспериментов с использованием параллельной реализации метода циклической редукции в разное количество потоков

J	1 поток	2 потока		4 потока		8 потоков	
	t,сек	t,сек	S	t,сек	S	t,сек	S
256	0,125	0,234	0,534	0,452	0,277	0,686	0,182
512	0,172	0,296	0,581	0,639	0,269	0,858	0,200
1024	0,25	0,374	0,668	0,733	0,341	1,092	0,229
2048	0,406	0,499	0,834	0,998	0,407	1,404	0,289
4096	0,702	0,671	1,046	1,295	0,542	1,825	0,385

8192	1,31	1,076	1,217	1,732	0,756	2,434	0,538
16384	2,527	1,732	1,549	2,527	1	3,37	0,75
32768	4,992	3,12	1,6	3,245	1,538	5,008	0,997
65536	10,202	7,098	1,437	6,193	1,647	7,878	1,295

Представленные результаты экспериментов свидетельствуют о плохой масштабируемости приложения. На 4-х потоках максимальное ускорение составляет примерно 1.7, а на 8-ми потоках – 1.3, что еще ниже, чем для OpenMP-реализации. Если просмотреть результаты Консигенсу-анализа (рис. 19), то профиль в целом будет очень похож на профиль предыдущей параллельной версии. Разница лишь в том, что участки активности потоков не являются строго параллельными. Данный факт объясняется тем, что нагрузка между потоками в ТВВ-реализации распределяется динамически.

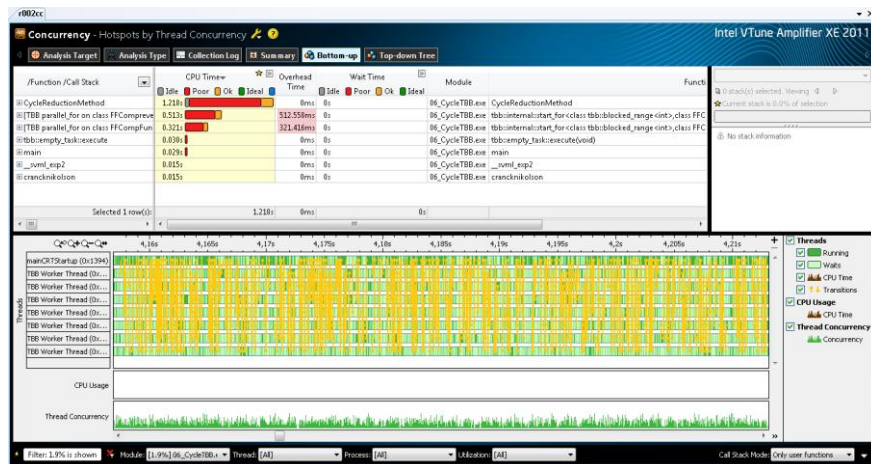


Рис. 19. Результаты запуска Intel Parallel Amplifier XE в режиме Concurrency

Узким местом, как и в OpenMP-реализации, являются накладные расходы на остановку и возобновление потоков (наличие большого количества желтых стрелок), а также организация работы с данными.

9. Дополнительные задания

1. Для решения трехдиагональной системы встройте функцию библиотеки MKL. Оцените эффективность использования библиотечных функций по сравнению с прогонкой и последовательной реализацией метода циклической редукции.

2. Реализуйте метод встречной прогонки [1]. Оцените эффективность использования встречной прогонки по сравнению с обычной прогонкой и параллельным методом циклической редукции. Объясните полученные результаты экспериментов.
3. Реализуйте алгоритм блочной прогонки [1]. Оцените эффективность использования со всеми предшествующими реализациями методов решения трехдиагональных систем. Объясните полученные результаты экспериментов. Оцените масштабируемость выполненной реализации блочной прогонки.

10. Литература

10.1. Основная литература

1. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.: Наука, 1977.
2. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989.
3. Вержбицкий В.М. Численные методы. – М.: Высшая школа, 2001.
4. Байков В.А., Жибер А.В. Уравнения математической физики. – Москва-Ижевск: Институт компьютерных исследований, 2003.
5. Самарский А.А., Николаев Е.С. Методы решения сеточных уравнений – М.: Наука, 1987. – С.130.
6. Gong. P, He. Z and Zhu. SP. Pricing convertible bonds based on a multi-stage compound option model, Physica A, 336, 2006, 449-462.
7. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.
8. Intel® Threading Building Blocks. Reference Manual. Version 1.6. Intel® Corporation, 2007.
9. Intel® Threading Building Blocks. Tutorial. Version 1.6. Intel® Corporation, 2007.
10. Самарский А.А. Введение численные методы. – СПб.: Лань, 2005.

10.2. Ресурсы сети Интернет

11. Официальный сайт OpenMP [www.openmp.org].
12. Страница библиотеки TBB на сайте корпорации Intel [<http://software.intel.com/en-us/articles/intel-tbb/>].
13. Словарь терминов Forex Euro Club [<http://enc.fxeuroclub.ru/350/>].

10.3. Дополнительная литература

14. Мееров И.Б., Никонов А.С., Русаков А.В., Шишков А.В. Параллельная реализация одного алгоритма нахождения цены конвертируемой облигации для систем с общей памятью // Технологии Microsoft в теории и практике программирования. Материалы конференции / Под ред. проф. В.П. Гергеля. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2009. – С. 287-292
15. Ярмушкин С.В., Головашкин Д.Л. Исследование параллельных алгоритмов решения трехдиагональных систем линейных алгебраических уравнений – Самара: Изд-во Самарского государственного технического университета, 2004. – №26. – С.78-82.