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

# Образовательный комплекс «Параллельные численные методы»

# Лабораторная работа Дифференциальные уравнения в частных производных

*Кустикова В.Д.*

*При поддержке компании Intel*

Нижний Новгород
2011

# Содержание

# Введение

> Когда наши дифференциальные уравнения мы умеем интегрировать, задача эта конечно не представляет затруднений. Но важно иметь способы, которые позволяли бы решать ее независимо от выполнимости этого интегрирования.
>
> *А.М. Ляпунов*

Уравнение, связывающее неизвестную функцию $u(x_1, x_2, \dots, x_n)$, независимые переменные $x_1, x_2, \dots, x_n$ и частные производные от неизвестной функции, называется *дифференциальным уравнением в частных производных*. *Решением* такого уравнения называется любая функция $u = u(x_1, x_2, \dots, x_n)$, которая, будучи подставлена в уравнение вместо неизвестной функции и ее частных производных, обращает это уравнение в тождество по независимым переменным.

Одним из важнейших классов дифференциальных уравнений в частных производных являются уравнения второго порядка (*порядком уравнения* называют порядок старшей частной производной, входящей в него). Важность данного класса задач обусловлена тем фактом, что математическими моделями многих процессов механики (колебания струн, стержней, мембран и трехмерных объемов), физики (электромагнитные колебания, распространение тепла или диффузия частиц в среде), гидро- и газодинамики (различные виды течений жидкости и газа), а также многих других областей знаний являются уравнения именно второго порядка.

Следует отметить, что только некоторое подмножество уравнений указанного класса можно решить аналитически (например, в случае постоянных коэффициентов). Большинство же задач, которые описывают явления и процессы окружающего нас мира, допускают лишь численное решение. Множество книг посвящено методам как аналитического, так и численного решения дифференциальных уравнений в частных производных (см., например, фундаментальные работы [1, 2]). Мы же остановимся на рассмотрении классического численного метода решения таких уравнений – *метода конечных разностей*, и его распараллеливании в системах с общей памятью.

Основная идея метода конечных разностей заключается в сведении решения дифференциального уравнения к решению разностных уравнений. Для получения совокупности разностных уравнений *(разностной схемы)* вместо одного дифференциального уравнения следует:

- заменить область непрерывного изменения аргументов дискретным множеством точек – сеткой (сетка, как правило, выбирается равномерная);
- заменить (аппроксимировать на сетке) дифференциальное уравнение разностной схемой.

При этом используются известные формулы аппроксимации производных, ознакомиться с которыми можно, например, в книге [3].

В связи с построением разностной схемы возникают следующие проблемы, которые типичны для разностных методов вообще. Во-первых, необходимо убедиться, что система линейных алгебраических уравнений (СЛАУ) имеет единственное решение, и указать алгоритм, позволяющий получить это решение. И, во-вторых, надо показать, что при стремлении шага сетки к нулю решение разностной задачи будет сходиться к решению исходной дифференциальной задачи. Вопросам разрешимости и сходимости разностных схем посвящена обширная литература, среди которой можно отметить работу [5].

В данной лабораторной работе рассматривается уравнение параболического типа (подробную классификацию уравнений в частных производных можно найти, например, в книге [4]), и один из подходов к построению разностных схем для такого типа уравнений – *схема Кранка-Николсона*, называемая также *схемой с весом* 1/2. Применение данной схемы позволяет без существенных вычислительных затрат повысить (по сравнению с другими подходами) порядок аппроксимации разностной схемы.

Решение возникающей при этом совокупности разностных уравнений сводится к многократному решению СЛАУ с трехдиагональной матрицей. Для решения таких СЛАУ известны специальные методы, в частности, приведенные 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.

Изложение проводится на примере одной из прикладных задач финансовой математики – задачи вычисления цены конвертируемой облигации [6, 13].

# 1. Методические указания

## 1.1. Цели и задачи работы

*Цель данной работы – продемонстрировать практическое применение параллельных алгоритмов линейной алгебры при решении дифференциального уравнения в частных производных на примере прикладной задачи из области финансовой математики.*

Данная цель предполагает решение следующих основных задач:

1. Изучение постановки задачи вычисления цены конвертируемой облигации [6].
2. Рассмотрение вычислительной схемы Кранка-Николсона в применении к решению дифференциального уравнения в частных производных, которым описывается изменение цены конвертируемой облигации. Приведение исходной задачи к многократному решению системы линейных алгебраических уравнений (СЛАУ) с трехдиагональной матрицей специального вида (каждая диагональ в отдельности содержит одинаковые элементы).
3. Освоение методов прогонки и циклической редукции [5] для решения СЛАУ с трехдиагональной матрицей для случая матрицы с одинаковыми элементами на каждой из диагоналей.
4. Реализация вычислительной схемы Кранка-Николсона применительно к рассматриваемой задаче с использованием метода прогонки для решения СЛАУ [1] с трехдиагональной матрицей полученного вида.
5. Выполнение программной реализации алгоритма циклической редукции в случае трехдиагональной матрицы специального вида, возникающей в рассматриваемой прикладной задаче.
6. Интеграция реализации алгоритма циклической редукции в реализацию схемы Кранка-Николсона.
7. Распараллеливание выполненной реализации метода циклической редукции на системы с общей памятью с использованием технологии OpenMP [7, 11].
8. Распараллеливание последовательной реализации метода циклической редукции с использованием библиотеки TBB [8, 9, 12].
9. Анализ производительности реализованной схемы решения дифференциального уравнения в частных производных при использовании методов прогонки и циклической редукции для решения СЛАУ с трехдиагональной матрицей.

10. Анализ масштабируемости приложения с использованием параллельных реализаций метода циклической редукции для решения СЛАУ с трехдиагональной матрицей.

## 1.2. Структура работы

В работе предлагается краткое описание прикладной задачи вычисления цены конвертируемой облигации, изменение которой описывается дифференциальным уравнением в частных производных. Далее рассматривается вычислительная схема Кранка-Николсона для решения указанного дифференциального уравнения. Фактически вычислительная схема включает в себя решение последовательности СЛАУ с трехдиагональной матрицей специального вида (каждая из диагоналей, по сути, задается одним числом). Далее в работе предлагается 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 | MicrosoftWindows 7 |
| Среда разработки | Microsoft Visual Studio 2008 |
| Компилятор, профилировщик, отладчик | IntelParallelStudioXE |
| Библиотека TBB | 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 дифференциальных уравнениях в частных производных, их применении и методах решения.
2. Рассмотреть постановку прикладной задачи. Пояснить «природу» дифференциального уравнения в частных производных, описывающего изменение цены конвертируемой облигации.
3. Рассмотреть вычислительную схему Кранка-Николсона на примере простого дифференциального уравнения в частных производных параболического типа, например, уравнения теплопроводности [1].
4. Продемонстрировать применение схемы Кранка-Николсона к решению дифференциального уравнения, которым описывается изменение цены конвертируемой облигации, сведение исходной задачи к решению СЛАУ с трехдиагональной матрицей специального вида, содержащей одинаковые элементы на каждой диагонали в отдельности.
5. Рассмотреть метод прогонки для решения СЛАУ с трехдиагональной матрицей специального вида.
6. Рассмотреть метод циклической редукции для решения СЛАУ с трехдиагональной матрицей специального вида.
7. Выполнить программную реализацию вычислительной схемы Кранка-Николсона применительно к рассматриваемой задаче с использованием метода прогонки.
8. Реализовать последовательную версию алгоритма циклической редукции для случая трехдиагональной матрицы, у которой на каждой диагонали в отдельности стоят одинаковые элементы.
9. Встроить последовательную реализацию циклической редукции в схему Кранка-Николсона.
10. Распараллелить реализацию алгоритма циклической редукции сначала с использованием технологии OpenMP [11], а затем с помощью средств библиотеки TBB [12].
11. Провести анализ производительности реализаций схемы Кранка-Николсона с использованием последовательных версий методов прогонки и циклической редукции для решения СЛАУ.
12. Выполнить анализ масштабируемости реализации схемы Кранка-Николсона при использовании параллельных версий метода циклической редукции для решения СЛАУ.

## 2. Задача вычисления цены конвертируемой облигации

*Конвертируемая облигация (КО, convertible bond)* – финансовый инструмент, дающий право держателю в любой момент времени перевести облигацию в заранее установленное количество акций или продолжить удерживать облигацию, получая фиксированный процент [6].

КО имеет два важных параметра – *номинальная стоимость* $K$ и *цена конвертации в одну акцию* $I_c$. Таким образом $\frac{K}{I_c}$ – заранее установленное количество акций, которые могут быть получены, если держатель захочет исполнить свое право на конвертацию.

Жизненный цикл КО делится на конечное число стадий – временных отрезков. Держатель КО имеет право конвертировать облигацию в заранее определенное количество акций $\frac{K}{I_c}$ в конце каждой из этих стадии, исходя из возможной прибыли, которую он может получить от конвертации. Если суммарная стоимость $\frac{K}{I_c}$ акций больше суммы процента по облигации и номинальной стоимости облигации $K$, то держатель переводит облигацию в акции, иначе продолжает удерживать ее, получая проценты.

Пусть существует $m^*$ моментов выплат процентов в пределах времени жизни КО ($t^*_k$ , $k$=1,2,...,$m^*$) и $n$ моментов времени, когда держатель принимает решение о конвертации ($t_i$,$i$=0,1,2,...,$n$; $n \geq m^*$). Множество моментов выплат является подмножеством моментов принятия решений. Обозначим через $T = t_n - t_0$ время жизни КО.

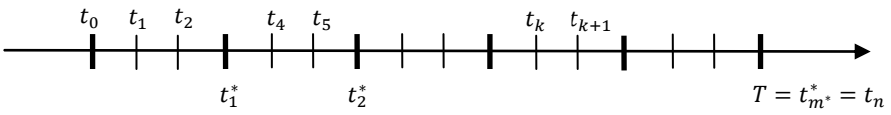


**Рис. 1.** Временная шкала периода жизни КО

Пусть $r_i$ – процент, выплачиваемый держателю облигации в момент времени $t^*_i$. С математической точки зрения $r_i$ скалярная величина, которая определяется в соответствии с формулой (1). Формула показывает, что если момент принятия решения о конвертации совпадает с моментом выплаты процента, то держатель может получить фиксированный процент, в противном случае, процент равен нулю.

$$r_i = \begin{cases} r_k^*, & \text{при } t_i = t_k^* \\ 0, & \text{иначе} \end{cases} \quad (i = 0,1,..,n;\ k = 1,2,..,m^*), \tag{1}$$

где $r_k^*$ – заданный параметр для каждого момента времени (далее в задаче считается одинаковым для всех $k$).

Ценой КО считается сумма, которую держатель может получить за продажу облигации на рынке. Цена КО $C_k = C_k(V_t, t)$ на каждом временном промежутке между двумя последовательными моментами принятия решения о конвертации $t_k$ и $t_{k+1}$ удовлетворяет дифференциальному уравнению в частных производных:

$$\frac{\partial C_k}{\partial t} + \frac{1}{2}\sigma^2 V_t^2 \frac{\partial^2 C_k}{\partial V_t^2} + \big(r_f - D\big)V_t \frac{\partial C_k}{\partial V_t} - r_f C_k = 0,$$

где$V_t$ – курс акций (неотрицательная скалярная величина, которая изменяется в известных пределах с некоторым фиксированным шагом), $r_f, D, \sigma$ – известные константные параметры задачи[1].

На цену КО $C_k$ накладываются естественные граничные условия с точки зрения рассматриваемой задачи (2), (3), (4) и (5).

$$C_{n-1}\big(V_{t_n}, t_n\big) = \max\left\{\frac{K}{I_c}V_{t_n}, K(1 + r_n)\right\} \tag{2}$$

$$C_{k-1}\big(V_{t_k}, t_k\big) = \max\left\{\frac{K}{I_c}V_{t_k}, Kr_k + C_k\big(V_{t_k}, t_k\big)\right\}, \quad k \in \{1, .., n-2\} \tag{3}$$

$$\lim_{V_t \to 0} C_k(V_t, t) = Ke^{-r_f(t_{k+1}-t)}\Big(\sum_{m=0}^{n-k-1} r_{k+m+1}e^{-r_f(t_{k+m+1}-t_{k+1})} + e^{-r_f(t_n - t_{k+1})}\Big), \tag{4}$$

где $t \in [t_k, t_{k+1}], k = 0,1, .., n-1.$

$$\lim_{V_t \to +\infty} C_k(V_t, t) = \frac{K}{I_c}V_t e^{-D(t_{k+1}-t)}, \tag{5}$$

где $t \in [t_k, t_{k+1}], k = 0,1, .., n-1.$

Задача, таким образом, состоит в том, чтобы определить оптимальную цену КО на каждой стадии, при которой держатель получает максимальную выгоду в случае конвертации.

## 3. Вычислительная схема Кранка-Николсона

Для решения дифференциального уравнения, описывающего изменение цены КО на каждой стадии, с граничными условиями (2), (3), (4) и (5) воспользуемся схемой Кранка-Николсона [6]. Данная вычислительная схема является абсолютно устойчивой, поэтому достаточно часто используется

[1] Финансовый смысл этих величин описывается в [6].

на практике при решении дифференциальных уравнений параболического типа.

Сначала преобразуем уравнение относительно цены КО $C_k = C_k(V_t, t)$ к виду, который удобен для применения указанной вычислительной схемы. Для этого выполним замену переменных $z_t = \ln V_t$ и преобразуем частные производные по переменной $V_t$ в частные производные по $z_t$:

$$\frac{\partial C_k}{\partial V_t} = \frac{\partial C_k}{\partial z_t} \cdot \frac{\partial z_t}{\partial V_t} = \frac{1}{V_t}\frac{\partial C_k}{\partial z_t} \tag{6}$$

$$\frac{\partial^2 C_k}{\partial {V_t}^2} = \frac{\partial}{\partial V_t}\left(\frac{1}{V_t}\frac{\partial C_k}{\partial z_t}\right) = -\frac{1}{{V_t}^2}\frac{\partial C_k}{\partial z_t} + \frac{1}{V_t}\frac{\partial}{\partial V_t}\left(\frac{\partial C_k}{\partial z_t}\right) = -\frac{1}{{V_t}^2}\frac{\partial C_k}{\partial z_t} + \frac{1}{V_t}\frac{\partial}{\partial z_t}\left(\frac{\partial C_k}{\partial z_t}\right)\frac{\partial z_t}{\partial V_t} =$$
$$-\frac{1}{{V_t}^2}\frac{\partial C_k}{\partial z_t} + \frac{1}{{V_t}^2}\frac{\partial^2 C_k}{\partial {z_t}^2} = \frac{1}{{V_t}^2}\left(\frac{\partial^2 C_k}{\partial {z_t}^2} - \frac{\partial C_k}{\partial z_t}\right) \tag{7}$$

Подставим полученные выражения частных производных (6) и (7) в исходное уравнение:

$$\frac{\partial C_k}{\partial t} + \frac{1}{2}\sigma^2 V_t^2 \frac{1}{{V_t}^2}\left(\frac{\partial^2 C_k}{\partial {z_t}^2} - \frac{\partial C_k}{\partial z_t}\right) + \left(r_f - D\right)V_t \frac{1}{V_t}\frac{\partial C_k}{\partial z_t} - r_f C_k = 0 \tag{8}$$

$$\frac{\partial C_k}{\partial t} + \frac{1}{2}\sigma^2 \frac{\partial^2 C_k}{\partial {z_t}^2} + \left(r_f - D - \frac{1}{2}\sigma^2\right)\frac{\partial C_k}{\partial z_t} - r_f C_k = 0 \tag{9}$$

Дальнейшие выкладки и преобразования будут выполняться с уравнением (9).$D,\ r_f,\ \sigma$ – параметры дифференциального уравнения, которые являются известными константами. Задача (9) с граничными условиями (2), (3), (4) и (5) решается в обратном времени для каждой стадии $[t_k, t_{k+1}]$ между двумя последовательными моментами принятия решения о конвертации. Затем на текущей стадии определяется оптимальное значение КО.

В соответствии с введенными обозначениями переменная $z_t$ принимает значения в бесконечных пределах. При построении разностной схемы необходимо ограничить интервал изменения значений $z_t$ достаточно малой величиной снизу $\underline{z} = \ln \underline{V}$ и достаточно большой величиной сверху $\overline{z} = \ln \overline{V}$ ($\underline{V} = 0.01, \overline{V} = 10000.0$ – некоторые постоянные, значения которых обусловлены рассматриваемой прикладной задачей, т.к. курс акций не может упасть ниже величины $\underline{V}$ или подняться выше $\overline{V}$).

Введем равномерную сетку по времени для каждого *k*-ого интервала и по переменной $z_t$ следующим образом (см. рис. 2):

$$t_{k+1} - t_k = M\tau,\ \overline{z} - \underline{z} = J\zeta.$$

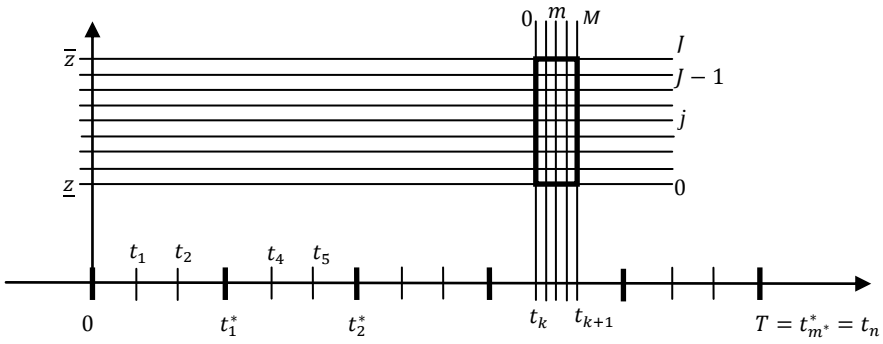


**Рис. 2.** Сетка для построения вычислительной схемы Кранка-Николсона

Для аппроксимации уравнения в частных производных (9) будем использовать следующие разностные операторы:

$$C_k \approx \frac{1}{2}\left(C_{k,j}^{m+1} + C_{k,j}^{m}\right); \tag{10}$$

$$\frac{\partial C_k}{\partial t} \approx \frac{C_{k,j}^{m+1} - C_{k,j}^{m}}{\tau}; \tag{11}$$

$$\frac{\partial C_k}{\partial z_t} \approx \frac{1}{2}\left(\frac{C_{k,j+1}^{m+1} - C_{k,j-1}^{m+1}}{2\zeta} + \frac{C_{k,j+1}^{m} - C_{k,j-1}^{m}}{2\zeta}\right); \tag{12}$$

$$\frac{\partial^2 C_k}{\partial z_t^2} \approx \frac{1}{2}\left(\frac{C_{k,j+1}^{m+1} - 2C_{k,j}^{m+1} + C_{k,j-1}^{m+1}}{\zeta^2} + \frac{C_{k,j+1}^{m} - 2C_{k,j}^{m} + C_{k,j-1}^{m}}{\zeta^2}\right); \tag{13}$$

где $C_{k,j}^{m} = C_k\left(\underline{z} + j\zeta, t_k + m\tau\right)$.

Подставляя разностные операторы, получаем аппроксимацию дифференциального уравнения (9):

$$\frac{C_{k,j}^{m+1} - C_{k,j}^{m}}{\tau} + \frac{1}{4}\sigma^2\left(\frac{C_{k,j+1}^{m+1} - 2C_{k,j}^{m+1} + C_{k,j-1}^{m+1}}{\zeta^2} + \frac{C_{k,j+1}^{m} - 2C_{k,j}^{m} + C_{k,j-1}^{m}}{\zeta^2}\right) + \frac{1}{2}\Big(r_f - D - \frac{1}{2}\sigma^2\Big)\left(\frac{C_{k,j+1}^{m+1} - C_{k,j-1}^{m+1}}{2\zeta} + \frac{C_{k,j+1}^{m} - C_{k,j-1}^{m}}{2\zeta}\right) - \frac{1}{2}r_f\left(C_{k,j}^{m+1} + C_{k,j}^{m}\right) = 0, \tag{14}$$

где $j = \overline{1, J-1}, m = \overline{0, M-1}$.

Коэффициенты в узловых значениях сетки вычисляются по формулам:

$$C_{k,j-1}^{m}: \quad a_{-1} = \frac{\sigma^2\tau}{4\zeta^2} - \frac{\left(r_f - D - \frac{1}{2}\sigma^2\right)\tau}{4\zeta}; \tag{15}$$

$$C_{k,j}^{m}: \quad a_0 = -\frac{\sigma^2\tau}{2\zeta^2} - 1 - \frac{r_f\tau}{2}; \tag{16}$$

$$C_{k,j+1}^{m}: \quad a_1 = \frac{\sigma^2\tau}{4\zeta^2} + \frac{\left(r_f - D - \frac{1}{2}\sigma^2\right)\tau}{4\zeta} \tag{17}$$

$$C_{k,j-1}^{m+1}: \quad b_{-1} = -\frac{\sigma^2\tau}{4\zeta^2} + \frac{\left(r_f - D - \frac{1}{2}\sigma^2\right)\tau}{4\zeta}; \tag{18}$$

$$C_{k,j}^{m+1}: \quad b_0 = \frac{\sigma^2\tau}{2\zeta^2} - 1 + \frac{r_f\tau}{2}; \tag{19}$$

$$C_{k,j+1}^{m+1}: \quad b_1 = -\frac{\sigma^2\tau}{4\zeta^2} - \frac{\left(r_f - D - \frac{1}{2}\sigma^2\right)\tau}{4\zeta}. \tag{20}$$

Тогда систему разностных уравнений (14) для $k$-ой стадии можно записать в виде:

$$a_{-1}C_{k,j-1}^{m} + a_0C_{k,j}^{m} + a_1C_{k,j+1}^{m} = b_{-1}C_{k,j-1}^{m+1} + b_0C_{k,j}^{m+1} + b_1C_{k,j+1}^{m+1}, \tag{21}$$

где $j \in \{1, \dots, J-1\}, m \in \{0, \dots, M-1\}$.

Запишем матричное представление приведенной системы:

$$\begin{bmatrix} a_0 & a_1 & & & & \\ a_{-1} & a_0 & a_1 & & & \\ & a_{-1} & a_0 & a_1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{-1} & a_0 & a_1 \\ & & & & a_{-1} & a_0 \end{bmatrix}_{(J-1)\times(J-1)} \begin{pmatrix} C_{k,1}^{m} \\ C_{k,2}^{m} \\ \vdots \\ C_{k,J-2}^{m} \\ C_{k,J-1}^{m} \end{pmatrix}_{(J-1)\times 1}$$

$$= \begin{pmatrix} b_{-1}C_{k,0}^{m+1} + b_0C_{k,1}^{m+1} + b_1C_{k,2}^{m+1} - a_{-1}C_{k,0}^{m} \\ b_{-1}C_{k,1}^{m+1} + b_0C_{k,2}^{m+1} + b_1C_{k,3}^{m+1} \\ b_{-1}C_{k,2}^{m+1} + b_0C_{k,3}^{m+1} + b_1C_{k,4}^{m+1} \\ \vdots \\ b_{-1}C_{k,J-3}^{m+1} + b_0C_{k,J-2}^{m+1} + b_1C_{k,J-1}^{m+1} \\ b_{-1}C_{k,J-2}^{m+1} + b_0C_{k,J-1}^{m+1} + b_1C_{k,J}^{m+1} - a_1C_{k,J}^{m} \end{pmatrix}_{(J-1)\times 1}$$

$$= \begin{bmatrix} b_{-1} & b_0 & b_1 & & & \\ & b_{-1} & b_0 & b_1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{-1} & b_0 & b_1 \\ & & & & b_{-1} & b_0 & b_1 \end{bmatrix}_{(J-1)\times(J+1)} \begin{pmatrix} C_{k,0}^{m+1} \\ C_{k,1}^{m+1} \\ \vdots \\ C_{k,J-1}^{m+1} \\ C_{k,J}^{m+1} \end{pmatrix}_{(J+1)\times 1} -$$

$$\begin{pmatrix} a_{-1}C_{k,0}^{m} \\ 0 \\ \vdots \\ 0 \\ a_1C_{k,J}^{m} \end{pmatrix}_{(J-1)\times 1}, \tag{22}$$

где $k \in \{0, \dots, n-1\}, m \in \{0, \dots, M-1\}$.

Далее получим разностные уравнения для граничных условий, используя ранее приведенные разностные операторы:

$$C_{n-1,j}^{M} = max\left\{K(1+r_n), \frac{K}{I_c}e^{\underline{z}+j\zeta}\right\}, j = \overline{0,J} \tag{23}$$

$$C_{k-1,j}^{M} = max\left\{Kr_k + C_{k,j}^{0}, \frac{K}{I_c}e^{\underline{z}+j\zeta}\right\}, k = \overline{1,n-1};\ j = \overline{0,J} \tag{24}$$

$$C_{k,0}^{m} = Ke^{-r_f(t_{k+1}-t_k-m\tau)}\left(\sum_{l=0}^{n-k-1} r_{k+l+1}e^{-r_f(t_{k+l+1}-t_{k+1})} + e^{-r_f(t_n-t_{k+1})}\right),$$

$$m = \overline{0,M}; k = \overline{0,n-1} \tag{25}$$

$$C_{k,J}^{m} = \frac{K}{I_c}e^{\overline{z}}e^{-D(t_{k+1}-t_k-m\tau)}, m = \overline{0,M};\ k = \overline{0,n-1}. \tag{26}$$

Таким образом, решение задачи вычисления оптимальной цены КО включает последовательность нескольких действий.

1. Вычислить коэффициенты $a_{-1}, a_0, a_1, b_{-1}, b_0, b_1$ (на каждой стадии – временном интервале $[t_k, t_{k+1}]$ – коэффициенты принимают одинаковые значения и зависят от известных параметров уравнения и величины разбиения).
2. На каждой стадии, начиная с последней:
   a. Вычислить граничные значения $C_{n-1,j}^{M}$ и $C_{k-1,j}^{M}$ согласно формулам (23) и (24).
   b. Для каждого момента времени в разбиении интервала $[t_k, t_{k+1}]$ от $M-1$ до 0:
      i. Умножить матрицу $B$ на вектор $CB = \left(C_{k,0}^{m+1}, \dots, C_{k,J}^{m+1}\right)$. Обозначим это произведение $CB_{next} = B * CB$.
      ii. Вычислить значения $C_{k,0}^{m}$ и $C_{k,J}^{m}$ согласно (25) и (26).
      iii. Вычесть из первой и последней компоненты вектора $CB_{next}$ произведения $a_{-1}C_{k,0}^{m}$ и $a_1C_{k,J}^{m}$ соответственно.
      iv. Решить систему разностных уравнений $Ax = CB_{next}$, где $A$ – трехдиагональная матрица, на каждой диагонали которой стоят одинаковые значения.
   c. Выбрать из полученного набора значений – вектора, содержащего значения цены КО в зависимости от стоимости акции, – оптимальное значение цены КО.

Чтобы определить оптимальное значение цены КО в каждый момент принятия решения о продаже, введем функцию $G_{k,j}$ [6]:

$$G_{k,j} = \begin{cases} Kr_k + C_{k,j}^0 - \frac{K}{I_c} e^{\underline{z}+j\zeta}, \ \ k = 1,2,..,n-1, \\ \\ \qquad K(1+r_n) - \frac{K}{I_c} e^{\underline{z}+j\zeta}, \ \ k = n. \end{cases} \tag{27}$$

Тогда из равенства $z_t = \ln V_t$ оптимальное значение цены КО определяется из формулы:

$$V_{t_k}^* = e^{\underline{z}+j^*(k)\zeta},$$

где $j^*(k) = \{j \big| G_{k,j} = \min_{j\in[0,J]} \big| G_{k,j} \big| \}, k = 1, \dots, n.$

## 4. Метод прогонки

Решение задачи (22) с граничными условиями (23), (24), (25) и (26), в конечном счете, сводится к многократному решению трехдиагональной системы уравнений вида (28).

$$\begin{bmatrix} b & c & & & & \\ a & b & c & & & \\ & a & b & c & & \\ & & \ddots & \ddots & \ddots & \\ & & & a & b & c \\ & & & & a & b \end{bmatrix}_{(N-1)\times(N-1)} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-2} \\ x_{N-1} \end{pmatrix}_{(N-1)\times 1} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{pmatrix}_{(N-1)\times 1} \tag{28}$$

Рассмотрим метод прогонки, применяемый для решения СЛАУ с трехдиагональной матрицей, для случая, когда на каждой диагонали в отдельности стоят одинаковые элементы[2].

Предположим, что имеет место соотношение (29).

$$x_i = \alpha_{i+1} x_{i+1} + \beta_{i+1} \tag{29}$$

с неопределенными коэффициентами $\alpha_{i+1}$ и $\beta_{i+1}$, и подставим выражение (29) в$i$-е уравнение системы:

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

Сравнивая полученное выражение с (29), находим

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

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

$$\beta_{i+1} = \frac{f_i - a\beta_i}{a\alpha_i + b}, i = 2, \dots, N-2 \quad (30)$$

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

Вычислив $\alpha_2, \beta_2$ и переходя от $i$ к $i+1$ в формулах (29), определим $\alpha_i, \beta_i, i = 3, \dots N-1$.

Определим $x_{N-1}$ из последнего уравнения системы и условия (28) при $i = N-2$.

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

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

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 (29).

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

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:

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

## 5. 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 (28) и рассмотрим метод циклической редукции[3] применительно к указанной системе.

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

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 (28) можно записать в виде (31).

$$\begin{bmatrix} a & b & c & & & \\ & a & b & c & & \\ & & \ddots & \ddots & \ddots & \\ & & & a & b & c \\ & & & & a & b & c \end{bmatrix}_{(N-1)\times(N+1)} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}_{(N+1)\times 1} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}_{(N+1)\times 1} \tag{31}$$

Будем считать, что размерность системы (31) 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 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.

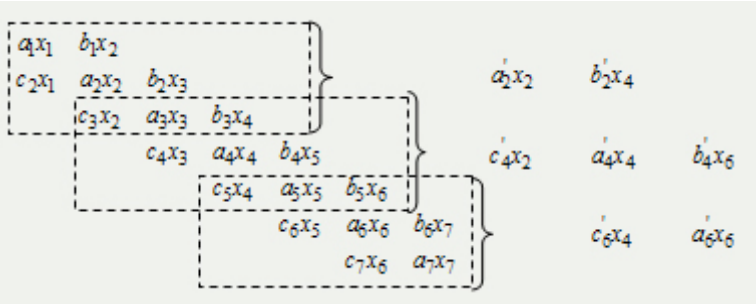


**Рис. 3.** Схема циклической редукции

На первой итерации алгоритма циклической редукции тройка рассматриваемых уравнений для системы (31) выглядит следующим образом:

$$\begin{cases} ax_{i-2} + bx_{i-1} + cx_i = f_{i-1} \\ ax_{i-1} + bx_i + cx_{i+1} = f_i, \text{ где } i = 2,4,6, \dots N-2 \\ ax_i + bx_{i+1} + cx_{i+2} = f_{i+1} \end{cases} \quad (32)$$

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 (32):

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

Подставим полученное выражение во второе уравнение системы (32):

$$\frac{a}{b}(f_{i-1} - ax_{i-2} - cx_i) + bx_i + cx_{i+1} = f_i$$

$$-\frac{a^2}{b}x_{i-2} + x_i\left(b - \frac{ac}{b}\right) + cx_{i+1} = f_i - \frac{a}{b}f_{i-1}$$

$$-a^2x_{i-2} + x_i(b^2 - ac) + bcx_{i+1} = bf_i - af_{i-1} \quad (33)$$

Умножим уравнение (32.3) на $c$:

$$acx_i + bcx_{i+1} + c^2x_{i+2} = cf_{i+1} \quad (34)$$

Вычтем из (33) уравнение (34). В результате приведения подобных слагаемых получим уравнение вида:

$$-a^2x_{i-2} + (b^2 - 2ac)x_i - c^2x_{i+2} = bf_i - cf_{i+1} - af_{i-1} \quad (35)$$

Разделим обе части уравнения на $b$, введем обозначения $\alpha = -\frac{a}{b}, \beta = -\frac{c}{b}$. Получим систему уравнений (36).

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

где $i = 2, 4, 6, \dots, N-2$. (36)

Система (36) имеет структуру, схожую со структурой исходной системы уравнений (31), но полученная система меньшей размерности. 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.

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

$$a^{(1)} = \alpha a^{(0)}, b^{(1)} = b^{(0)} + 2\alpha c^{(0)}, c^{(1)} = -\beta c^{(0)}, \quad (37)$$

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

$$a^{(j+1)} = \alpha a^{(j)},\ b^{(j+1)} = b^{(j)} + 2\alpha c^{(j)},\ c^{(j+1)} = -\beta c^{(j)}, \quad (38)$$

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:

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

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

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

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

$$f_i^{(j+1)} = \alpha f_{i-2^j}^{(j)} + f_i^{(j)} + \beta f_{i+2^j}^{(j)},\ i = 2^j, 2^{j+1} \ldots N-2^j, \qquad (39)$$

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.

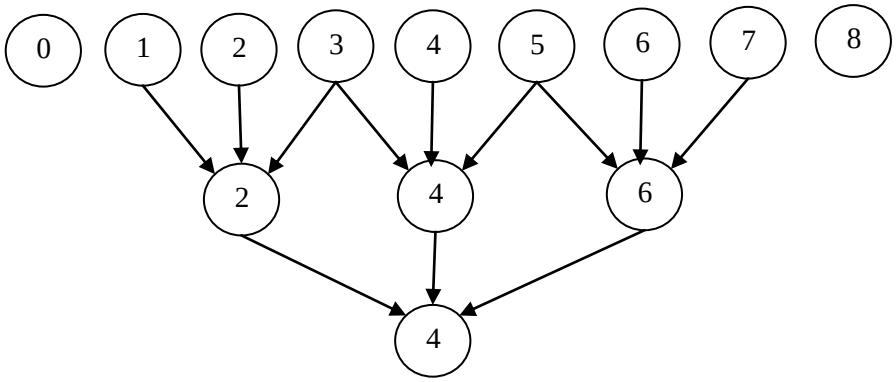


**Рис. 4.** Схема исключения переменных с нечетными номерами при $N = 8$

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 (40) через известные переменные, полученные на предшествующем шаге обратного хода.

$$a^{(l)}x_{i-2^l} + b^{(l)}x_i + c^{(l)}x_{i+2^l} = f_i^{(l)} \qquad (40)$$

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

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 (40)). Такая схема восстановления позволяет при пересчете правых частей уравнений в прямом ходе редукции затирать значения, полученные на предыдущем шаге.

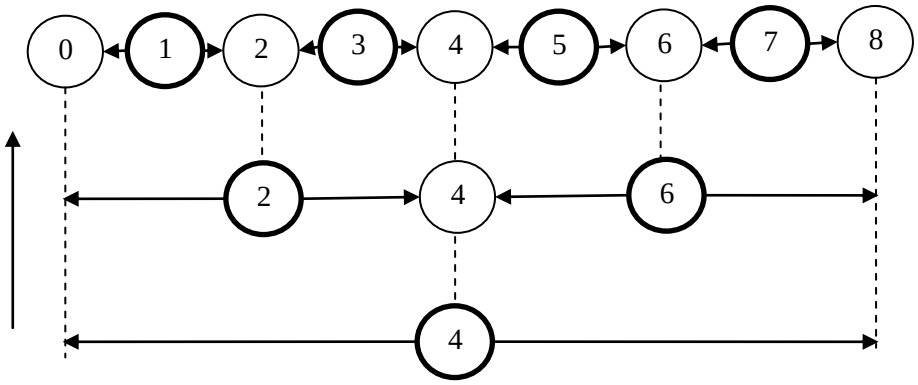


**Рис. 5.** Схема восстановления переменных на каждой итерации обратного хода редукции при $N = 8$

# 6. Программная реализация

## 6.1. Вычислительная схема Кранка-Николсона с использованием метода прогонки

Перейдем к программной реализации рассмотренных выше методов. Прежде всего, создадим новое **Решение** (**Solution**), в которое включим первый **Проект** (**Project**) данной лабораторной работы. Для этого последовательно выполните следующие шаги:

- Запустите приложение **Microsoft Visual Studio 2008**.
- В меню **File** выполните команду **New→Project…**.
- Как показано на рис. 6, в диалоговом окне **New Project** в типах проекта выберите **Win32**, в шаблонах **Win32 Console Application**, в поле **Solution** введите **07_PDE**, в поле **Name** – **01_Sweep**, в поле **Location** укажите путь к папке с лабораторными работами курса – **c:\ParallelCalculus**. Нажмите **OK**.

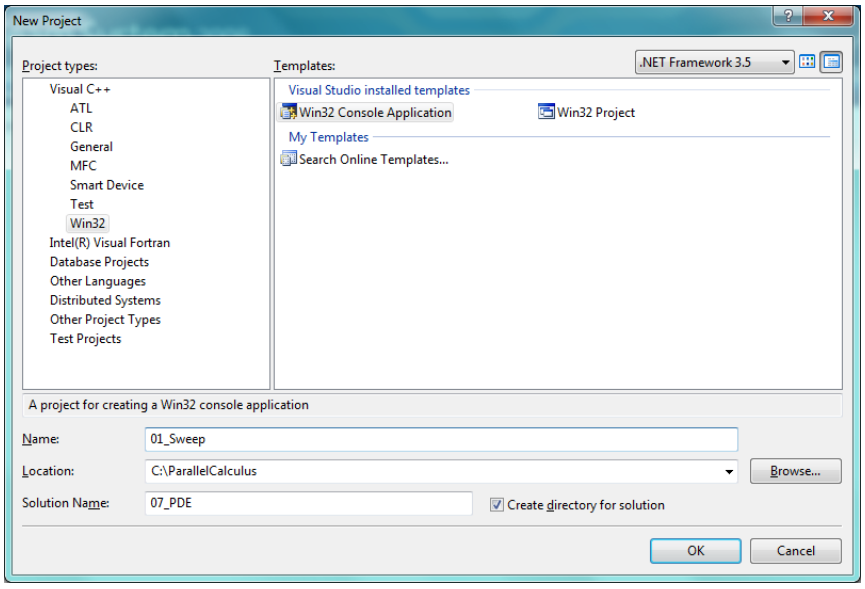


**Рис. 6.** Создание решения для лабораторной работы

- 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→NewItem…**. В дереве категорий слева выберите **Code**, в шаблонах справа – **C++ File (.cpp)**, в поле **Name** введите имя файла **main**. Нажмите **Add**.
- 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 **DiffEquation**. Нажмите **Add**. Созданный файл будет в дальнейшем содержать реализации функций, необходимых для решения поставленной прикладной задачи.

- В окне **Solution Explorer** в папке **Header Files** выполните команду контекстного меню **Add→New Item…**. В дереве категорий слева выберите **Code**, в шаблонах справа – **Header File (.h)**, в поле **Name** введите имя файла **DiffEquation**. Нажмите **Add**. В данном файле будут находиться прототипы необходимых функций.

Теперь создадим в файле **main.cpp** заготовку функции **`main()`**, в которую через аргумент командной строки будем передавать число разбиений $J$. Остальные параметры дифференциального уравнения примем равными некоторым константам, которые чаще всего используются при решении задачи вычисления цены конвертируемой облигации. Также для контроля правильности работы приложения все результаты решения прикладной задачи (цена КО на начальной стадии для всех возможных значений цены акции и оптимальная цена КО 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
  // Вызов функции ВЫЧИСЛЕНИЯ ЦЕНЫ КО
  // ...
  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]);
  // закрытиефайлов
  fclose(fstockprice);
  fclose(fcbprice);
  // освобождение памяти, выделенной под результаты
  free(stockprice);
  free(cbprice);
  return 0;
}
```

Относительно представленного кода необходимо сделать два комментария: 1) в нем отсутствуют подключения необходимых библиотек – предоставляем читателю внести их самостоятельно; 2) вместо комментария «Вызов функции ВЫЧИСЛЕНИЯ ЦЕНЫ КО» необходимо вставить вызов расчетной функции, к обсуждению реализации которой мы и переходим.

Начнем с того, что исходную прикладную задачу можно разделить на несколько более простых, каждой из которой соответствует программная функция:

1. Функции вычисления диагональных элементов $a_{-1}, a_0, a_1, b_{-1}, b_0, b_1$ матриц – **computecoeff_a()**, **computecoeff_b()**. Эта пара функций содержит реализацию формул (15) – (20).

```
// step = T / m*
// delta_t = τ
// delta_z = ζ
// rk_s = r*k
// sigma - параметр σ дифференциального уравнения
// D, rf - параметры дифференциального уравнения
int computecoeff_a(double sigma, double delta_t,
                   double delta_z, double D, double rf,
                   double *a_0, double *a_1, double *a_2)
{
  double sqsigma = sigma * sigma;
  double sqdeltaz = delta_z * delta_z;
  (*a_0) = (sqsigma / (4.0 * sqdeltaz) -
    (rf - D - sqsigma * 0.5) / (4.0 * delta_z)) * delta_t;
  (*a_1) = -1.0 - (sqsigma / (2.0 * sqdeltaz) + rf * 0.5)
           * delta_t;
  (*a_2) = (sqsigma / (4.0 * sqdeltaz) +
           (rf - D - sqsigma * 0.5) / (4.0 * delta_z))
           * delta_t;
  return 0;
}
int computecoeff_b(double sigma, double delta_t,
                   double delta_z, double D, double rf,
                   double *b_0, double *b_1, double *b_2)
{
  double sqsigma = sigma * sigma;
  double sqdeltaz = delta_z * delta_z;
  (*b_0) = (-sqsigma / (4.0 * sqdeltaz) +
           (rf - D - sqsigma * 0.5) / (4.0 * delta z))
           * delta_t;
  (*b_1) = -1.0 + (sqsigma / (2.0 * sqdeltaz) + rf * 0.5)
           * delta_t;
  (*b_2) = (sqsigma / (4.0 * sqdeltaz) +
           (rf - D - sqsigma * 0.5) / (4.0 * delta_z)) *
           (-delta_t);
  return 0;
}
```

2. Функции вычисления правого граничного условия на последней стадии **rightboundcondlast()** и на промежуточных стадиях разбиения **rightboundcond()** в соответствии с формулами (23) и (24).

```
// zmin = z
// delta_z = ζ
// K - номинальная стоимость облигации
// Ic - параметр дифференциального уравнения
```

```
int rightboundcond(double K, double Ic,
                   double zmin, double delta_z, double rk,
                   double *ck0j, int J, double *res)
{
  int i;
  double coeff1, coeff2, fe, se, step;
  coeff1 = K * rk;
  coeff2 = K / Ic;
  for (i = 0; i < J + 1; i++)
  {
    step = zmin + i * delta_z;
    fe = coeff1 + ck0j[i];
    se = coeff2 * exp(step);
    res[i] = (fe > se) ? fe : se;
  }
  return 0;
}
int rightboundcondlast(double K, double Ic,
                       double zmin, double delta_z,
                       double rn, int J, double *res)
{
  int i;
  double fe, se, coeff;
  fe = K * (1.0 + rn);
  coeff = K / Ic;
  for (i = 0; i < J + 1; i++)
  {
    se = coeff * exp(zmin + i * delta_z);
    res[i] = (fe > se) ? fe : se;
  }
  return 0;
}
```

3. Функция вычисления нижнего граничного условия **`bottomboundcond()`** по формуле (25).

```
// step = T / m*
// delta_t = τ
// rfunc – массив значений r_k
// K – номинальная стоимость облигации
// rf – параметр дифференциального уравнения
int bottomboundcond(int kind, int m, int n, double step,
                    double K, double rf, double *rfunc,
                    double delta_t, double *cond)
{
  int l;
  double coeff, term;
  coeff = K * exp((-1.0) * rf * (step - m * delta_t));
  term = exp((-1.0) * rf * (n - kind - 1) * step);
```

```
  *cond = 0.0;
  for (l = 0; l < n - kind; l++)
    *cond += (rfunc[kind + l + 1]*exp(-rf * l * step));
  (*cond) += term;
  (*cond) *= coeff;
  return 0;
}
```

4. Функция вычисления верхнего граничного условия **`topboundcond()`** в соответствии с формулой (26).

```
// step = T / m*
// delta_t = τ
// K – номинальная стоимость облигации
// D, Ic – параметры дифференциального уравнения
// zmax = z̄
int topboundcond(int m, double K, double D, double Ic,
                 double delta t, double step, double zmax,
                 double *cond)
{
  (*cond) = K * exp(zmax - D * (step - m * delta_t))/Ic;
  return 0;
}
```

5. Функция определения оптимальной цены конвертируемой облигации **`getoptimalstockprice()`** в некоторый момент принятия решения о продаже (см. формулу (27)).

```
// step = T / m*
// delta_t = τ
// delta_z = ζ
// zmin = z
// K – номинальная стоимость облигации
// Ic – параметр дифференциального уравнения
int getoptimalstockprice(double K, double Ic, double zmin,
    double step, double *rfunc,double delta_z,
    double delta_t, int J, int kind, double *c, double *V,
    int *optindex)
{
  int i, index;
  double gkj, rk, coeff, tmp, min;
  rk = rfunc[kind];
  coeff = K * rk;
  tmp = K / Ic;
  min = fabs(coeff + c[0] - tmp * exp(zmin));
  index = 0;
  for (i = 1; i < J; i++)
  {
    gkj = fabs(coeff+c[i]-tmp*exp(zmin+i*delta_z));
    if (gkj < min)
    {
```

```
      min = gkj;
      index = i;
    }
  }
  *V = exp(zmin + index * delta_z);
  *optindex = index;
  return 0;
}
```

6. Функция **matrvecmulti()** умножения трехдиагональной матрицы на вектор для вычисления произведения $CB_{next} = B * CB$ (см. описание схемы Кранка-Николсона). Каждая диагональ в отдельности содержит одинаковые элементы, поэтому представленная реализация не распространяется на общий случай трехдиагональных матриц.

```
// b_0 - элемент на нижней побочной диагонали
// b_1 - элемент на главной диагонали
// b_2 - элемент на верхней побочной диагонали
int matrvecmulti(double b_0, double b_1, double b_2,
                 double *vec, int J, double *res)
{
  int i;
  for (i = 0; i < J - 1; i++)
    res[i] = b_0*vec[i] + b_1*vec[i+1] + b_2*vec[i+2];
  return 0;
}
```

7. Реализация метода прогонки для решения СЛАУ с трехдиагональной матрицей, элементы каждой диагонали которой одинаковы, **sweepmethod()**. Поскольку прогонка выполняется неоднократно, а размерность системы не изменяется, то выделение памяти для хранения коэффициентов перенесем на уровень вызывающей функции. Описание метода было рассмотрено в § 4.

```
// a_0 - число на нижней побочной диагонали
// a_1 - число на главной диагонали
// a_2 - число на верхней побочной диагонали
// alpha - область памяти для хранения коэффициентов α
// beta - область памяти для хранения коэффициентов β
int sweepmethod(double a_0, double a_1, double a_2,
                double *cb_next, int J, double *x,
                double *alpha, double *beta)
{
    int size, i;
    double denominator;
    size = J - 1;

    // 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];
    }
    return OPERATION_OK;
}
```

8. Функция вычисления процента **r()**, выплачиваемого держателю в каждый момент времени, в соответствии с формулой (1).

```
// n - количество моментов принятия решения о конвертации
// m - количество моментов выплаты процента по облигации
// m = m*, step = T / m*
// delta_t = τ
// rk_s = r_k*
//EPS = 0.000001
double* r(double step, double delta_t, double rk_s,
          int n, int m)
{
  int i, j;
  double *r = (double *)malloc(sizeof(double) * (n + 1));
  for (i = 0; i <= n; i++)
    r[i] = 0.0;
  for (i = 0; i <= n; i++)
    for (j = 1; j <= m; j++)
      if (fabs(i * step - j) < EPS)
        r[i] = rk_s;
  return r;
}
```

9. И, наконец, функция **crancknikolson()**, содержащая реализацию вычислительной схемы Кранка-Николсона применительно к исходной задаче. В данном случае для решения СЛАУ с трехдиагональной матрицей используем метод прогонки, реализованный в функции **sweepmethod()**.

```
// rk_s = r_k*
// zmin = z
// zmax = z̄
```

```
// T - время жизни КО
// sigma - параметр σ дифференциального уравнения
// K - номинальная стоимость облигации
// D, Ic, rf - параметры дифференциального уравнения
// m = m*, M, J, N - параметры сетки (см. Рис. 2)
int cranckniколson(double T, double m, double sigma,
    double D, double K, double rk_s, double Ic, double rf,
    int M, int J, int n, double zmin, double zmax,
    double **cb_price, double *V)
{
    int kind, i, optindex, j;
    double a_0, a_1, a_2, b_0, b_1, b_2;
    double delta_t, delta_z, step, rn, rk, *cb_next;
    double topborder, bottomborder, *x_next, *rfunc,
           *alpha, *beta;
    cb_next = (double *)malloc(sizeof(double) * (J + 1));
    x_next  = (double *)malloc(sizeof(double) * (J + 1));

    delta_z = (zmax - zmin) / J;// шаг по J
    step = T / ((double) n); // шаг по времени на всем T
    delta_t = step / ((double) M); // шаг по времени в
                                   //малом интервале
    // вычисление коэффициентов
    computecoeff_a(sigma, delta_t, delta_z, D, rf,
                   &a_0, &a_1, &a_2);
    computecoeff_b(sigma, delta_t, delta_z, D, rf,
                   &b_0, &b_1, &b_2);

    // вычисление значений r
    rfunc = r(step, delta_t, rk_s, n, m);

    // вычисление на последней стадии
    rn = rfunc[n]; // получение rn
    rightboundcondlast(K, Ic, zmin, delta_z, rn,
       J, x_next);  // правое ГУ (23) размерность (J + 1)
    alpha = (double *)malloc(sizeof(double) * (J - 1));
    beta  = (double *)malloc(sizeof(double) * (J - 1));
    // проход по всем стадиям
    for (kind = n - 1; kind >= 0; kind--)
    {
        rk = rfunc[kind];       // получение значения rk
        // проход по разбиению малого интервала
        for (i = M; i > 0; i--)
        {
            // вычисление произведения
            // B * x_next = cb_next (J - 1)
            matrvecmulti(b_0, b_1, b_2, x_next, J,
                         cb_next);
```

```
            // вычисление верхнего ГУ
            topboundcond(i, K, D, Ic, delta_t, step,
                         zmax, &topborder);
            // вычисление нижнего ГУ
            bottomboundcond(kind, i, n, step, K, rf,
                            rfunc, delta_t, &bottomborder);
            // вычитание нижнего ГУ из первой компоненты
            // cb_next
            cb_next[0] -= (a_0 * bottomborder);
            // вычитание верхнего ГУ из последней
            // компоненты cb_next
            cb_next[J - 2] -= (a_2 * topborder);
            // запуск метода прогонки для
            // A * x_next = cb_next
            sweepmethod(a_0, a_1, a_2, cb_next, J,
                        x_next + 1, alpha, beta);
        }
        // вычисление оптимальной цены
        getoptimalstockprice(K, Ic, zmin, step, rfunc,
                             delta_z, delta_t, J, kind,
                             x_next,
                             &(V[kind]), &optindex);
        // вычисление правого ГУ на малом интервале
        rightboundcond(K, Ic, zmin, delta_z, rfunc[kind],
                       x_next, J, x_next);
    }
    *cb_price = x_next;
    free(cb_next);
    free(rfunc);
    free(alpha);
    free(beta);
    return OPERATION_OK;
}
```

Реализовав все подготовительные функции, мы можем теперь написать функцию **`getprices()`**, которая, фактически, будет содержать вызов выбранного метода решения – функцию **`crancknikolson()`** плюс некоторые предварительно необходимые действия.

```
// rk_s= r_k^*
// T - время жизни КО
// sigma - параметр σ дифференциального уравнения
// K - номинальная стоимость облигации
// D, Ic, rf - параметры дифференциального уравнения
// m = m*, M, J, N - параметры сетки (см. рис. 2)
int getprices(double T, int m, double sigma, double D,
  double K, double rk_s, double Ic, double rf, int M,
  int J, int n, double **stockprice, double **cbprice)
{
```

```
  double zmin, zmax;
  zmin = log(MINPRICE);
  zmax = log(MAXPRICE);
  *stockprice = (double *)malloc(sizeof(double) * n);
  crancknikolson(T, m, sigma, D, K, rk_s, Ic, rf, M, J,
                 n, zmin, zmax, cbprice, *stockprice);
  return 0;
}
```

И, наконец, осталось вставить в функцию **`main()`** вызов функции **`getprices()`**, и 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 при описании основной функции. При таких параметрах результирующие значения цены КО (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 КО при всех значениях цены акций на начальной стадии при $J = 8$

| **Курс акций** | **Цена конвертируемой облигации** |
|---|---|
| 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 КО на каждой стадии задайте размерности сетки, равными $\boldsymbol{J = 1024}$, $\boldsymbol{n = 5}$, $\boldsymbol{M = 100}$. В результате выполнения программы с указанным набором параметров значения оптимальной цены КО в пределах машинной погрешности должны быть равными 14.788423, 15.820447, 14.590242, 13.097473, 10.990454.

## 6.2. Последовательная версия алгоритма циклической редукции

Перед непосредственной реализацией алгоритма циклической редукции создадим в рамках решения **07_PDE** новый проект с названием **02_Cycle**. Повторите все действия, описанные в § 6.1. Также необходимо создать файл с именем **main.cpp,** заголовочный файл и файл исходных кодов **DiffEquation**.

После получения пустых файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**скопируем в них код из файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp** проекта **01_Sweep**. Из файлов **DiffEquation.h** и **DiffEquation.cpp** удалите объявление и реализацию метода прогонки.

Создайте файлы **CycleReduction.h** и **CycleReduction.cpp**, повторив действия, которые выполнялись при создании файлов **DiffEquation.h** и **DiffEquation.cpp**. В этих файлах будут размещаться объявление и реализация метода циклической редукции.

Реализацию метода циклической редукции, описанного в § 5, 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 циклического редукции для решения трехдиагональной системы вида $Ax = f$. 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,
//                 верхня 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 реализации алгоритма циклической редукции, остается модифицировать имеющуюся реализацию схемы Кранка-Николсона. Для этого достаточно добавить выделение памяти для хранения обновленных значений диагональных элементов и освобождение занятой памяти (аналогично реализации метода прогонки), т.к. на каждом шаге алгоритма поиска решения

ДУ размерность системы не изменяется, и заменить вызов функции обычной прогонки на вызов метода циклической редукции. Все необходимые изменения в функции, содержащей реализацию вычислительной схемы, выделены полужирным начертанием.

```
int crancknikolson(double T, double m, double sigma,
  double D, double K, double rk_s, double Ic, double rf,
  int M, int J, int n, double zmin, double zmax,
  double **cb_price, double *V)
{
  int kind, i, optindex, size;
  int dim;
  double a_0, a_1, a_2, b_0, b_1, b_2;
  double delta_t, delta_z, step, rn, rk, *cb_next;
  double topborder, bottomborder, *x_next, *rfunc;
  int q = (int)(log((double)J) / log(2.0));
  double *a, *b, *c;
  a = (double *)malloc(sizeof(double) * q);
  b = (double *)malloc(sizeof(double) * q);
  c = (double *)malloc(sizeof(double) * q);
  dim = J - 1;
  size = J - 2;
  cb_next = (double *)malloc(sizeof(double) * (J + 1));
  x_next  = (double *)malloc(sizeof(double) * (J + 1));

  delta_z = (zmax - zmin) / J;
  step = T / ((double) n);
  delta_t = step / ((double) M);
  computecoeff_a(sigma, delta_t, delta_z, D, rf,
                 &a_0, &a_1, &a_2);
  computecoeff_b(sigma, delta_t, delta_z, D, rf,
                 &b_0, &b_1, &b_2);
  rfunc = r(step, delta_t, rk_s, n, m);
  rn = rfunc[n];
  rightboundcondlast(K, Ic, zmin, delta_z,
                     rn, J, x_next);
  for (kind = n - 1; kind >= 0; kind--)
  {
    rk = rfunc[kind];
    for (i = M; i > 0; i--)
    {
      matrvecmulti(b_0, b_1, b_2, x_next, J,
                   cb_next + 1);
      topboundcond(i, K, D, Ic, delta_t, step,
                   zmax, &topborder);
      bottomboundcond(kind, i, n, step, K, rf,
                      rfunc, delta_t, &bottomborder);
      cb_next[1] -= (a_0 * bottomborder);
      cb_next[size + 1] -= (a_2 * topborder);
      CycleReductionMethod(x_next, a_0, a_1, a_2,
```

```
                              a, b, c, cb_next, J, q);
    }
    // вычисление оптимальной цены
    getoptimalstockprice(K, Ic, zmin, step, rfunc,
                  delta_z, delta_t, J, kind, x_next,
                  &(V[kind]), &optindex);
    rightboundcond(K, Ic, zmin, delta_z, rfunc[kind],
                  x_next, J, x_next);
  }
  *cb_price = x_next;
  free(cb_next);
  free(rfunc);
  free(a);
  free(b);
  free(c);
  return 0;
}
```

После того, как разработана программная реализация алгоритма циклической редукции и внесены изменения, необходимые для ее интеграции в схему Кранка-Николсона, скомпилируйте проект, выполнив команду **Build→Rebuild 02_Cycle**. Убедитесь в корректности результатов работы программы. Для этого воспользуйтесь данными, приведенными в § 6.1 (в частности, в табл. 2).

## 6.3. Параллельная версия алгоритма 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 **07_PDE** новый проект с названием **03_CycleParallel**. Повторите все действия, описанные в § 6.2. Также необходимо создать заголовочные файл **CycleReduction.h** и файл исходных кодов **CycleReduction.cpp**.

После получения пустых файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**,**CycleReduction.h**, **CycleReduction.cpp**скопируйте в них код из файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**, **CycleReduction.h**, **CycleReduction.cpp** проекта **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 код 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(THREADS_NUMBER)
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§ 6.1 (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).

## 6.4. Параллельная версия алгоритма 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 средств библиотеки Intel TBB [8, 9] создадим в рамках решения **07_PDE** новый проект с названием **04_CycleTBB**. Повторите все действия, описанные в § 6.1-6.3.

После получения пустых файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**,**CycleReduction.h**,**CycleReduction.cpp**скопируйте в них код из файлов **main.cpp**, **DiffEquation.h**, **DiffEquation.cpp**, **CycleReduction.h**,**CycleReduction.cpp** 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 TBB 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 TBB 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, **FFCompFunctor**, 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 FFCompFunctor
{
private:
  double *f;
  double alpha;
  double beta;
  int step;
  int start;
public:
  FFCompFunctor(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),
                 tbb::affinity_partitioner());
}
```

Теперь приведем реализации класса-функтора для распаллеливания 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),
                      tbb::affinity_partitioner());
}
```

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 реализация алгоритма циклической редукции. Задайте количество потоков, равным двум. Далее скомпилируйте проект, выполнив команду **Build→Rebuild 04_CycleTBB**, и воспользуйтесь данными, приведенными в § 6.1 (в частности, в табл. 2).

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

Задача вычисления цены КО 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 (29). Суммарное

количество операций прямого хода составляет $\big(1+3(J-2)\big)+\big(1+5(J-1)\big)=8J-14$. В процессе выполнения обратного хода вычисляется решение СЛАУ согласно (30). 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 уменьшится. 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 (37), (38) и пересчет правых частей системы в соответствии с выражениями (39). 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 $6(J-1)\left(\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{2^q}\right)=\frac{6(J-1)\frac{1}{2}\left(1-\frac{1}{2^q}\right)}{1-\frac{1}{2}}=\frac{6(J-1)^2}{J}$, 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 (40). Восстановление одной переменной требует 5 операций. В системе таким 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. 7 показана зависимость времени поиска оптимальной цены КО от размерности разбиения по вертикали, которое и определяет размерность трехдиагональной матрицы в СЛАУ.

Таблица 3. Время вычисления цены КО и оптимальной цены КО 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 | 0.078 | 0.031 |
| 512 | 0.125 | 0.078 |
| 1024 | 0.265 | 0.14 |
| 2048 | 0.499 | 0.265 |
| 4096 | 0.998 | 0.514 |
| 8192 | 2.012 | 1.092 |
| 16384 | 4.055 | 2.169 |
| 32768 | 8.126 | 4.399 |
| 65536 | 16.268 | 9.111 |

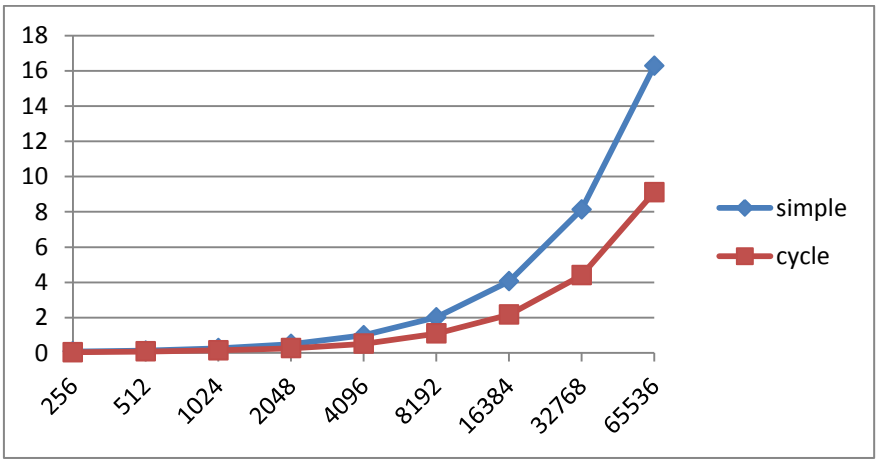


**Рис. 7.** Время поиска цены КО с использованием последовательных реализаций метода прогонки (simple) и метода циклической редукции (cycle)

Как видим, численные результаты противоречат теоретическим. Время решения задачи с помощью метода прогонки на больших размерностях матрицы почти 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 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 $\alpha_{i+1}$ и $\beta_{i+1}$ 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 noinline`**. Запустим Amplifier в режиме **LightWeight Hostspots** (слева в поддереве режимов анализа **Algorithm Analisys**). Из полученных результатов (Рис. 9 и Рис. 10) 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 КО при использовании редукции меньше, чем при использовании метода прогонки.

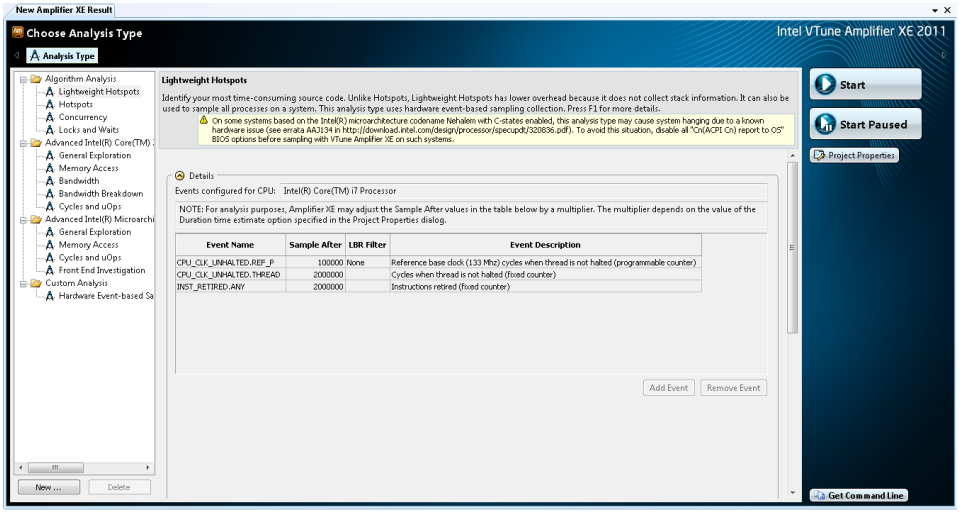


**Рис. 8.** Главное окно инструмента Intel Parallel Amplifier XE

| /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. 9.** 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. 10.** 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.

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

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

На рис. 11 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.

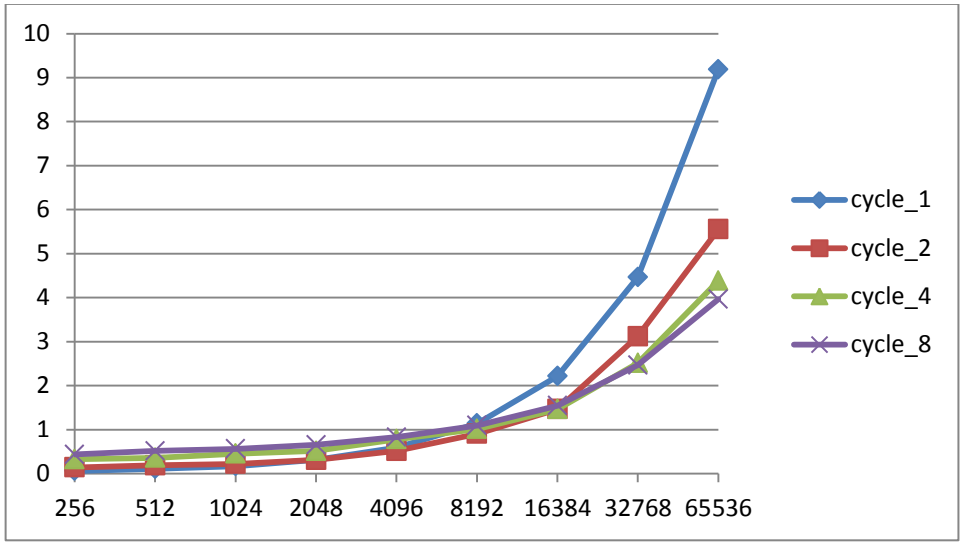


**Рис. 11.** Время решения задачи поиска оптимальной цены КО с использованием параллельной версии циклической редукции (cycle_1 – в 1 поток, cycle_2 – в 2 потока, cycle_4 – в 4 потока, cycle_8 – в 8 потоков)

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. 12) задайте параметр командной строки 256.

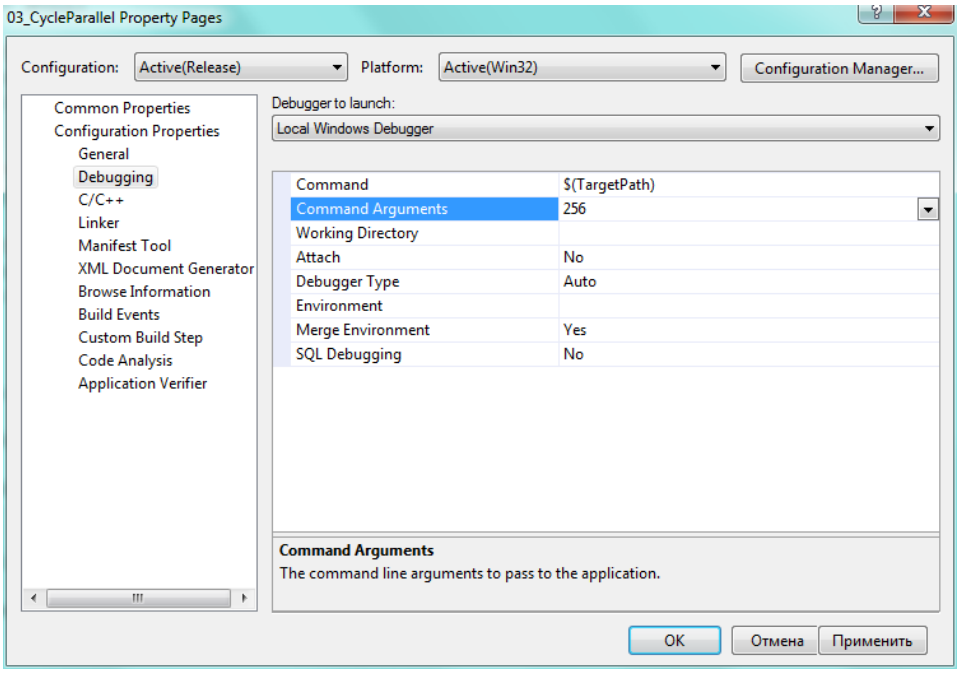

**Рис. 12.** Окно задания параметров командной строки в настройках проекта Visual Studio

Далее выберите режим анализа в дереве в левой части окна **Algorithm Analysis→Concurrency** и нажмите кнопку **Start** (Рис. 8). Процедура сбора статистики занимает некоторое время.

На рис. 13 показаны результаты запуска инструмента на тестовой инфраструктуре, описанной в § 1.3.

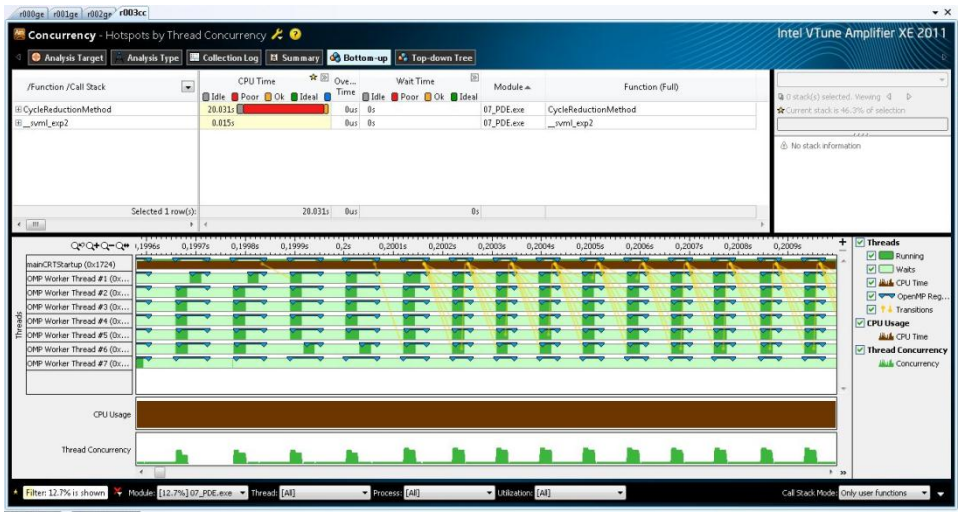


**Рис. 13.** Результаты запуска Intel Parallel Amplifier XE

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. 14).

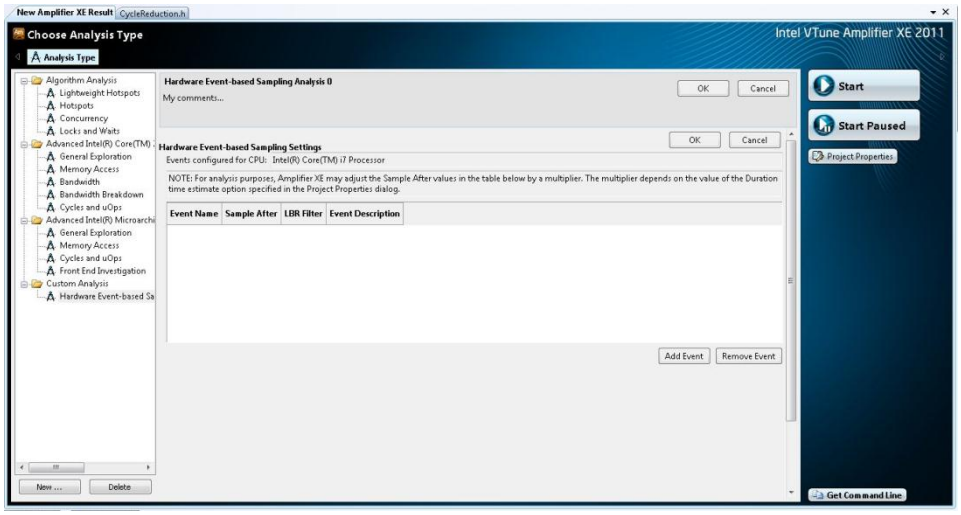


**Рис. 14.** Окно создания нового типа анализа программного приложения

Далее добавим интересующие события. После нажатия кнопки **Add Event** будет сформирован список допустимых событий, из которых необходимо выбрать L2_RQSTS.MISS и MEM_LOAD_RETIRED.LLC_MISS (рис. 15).

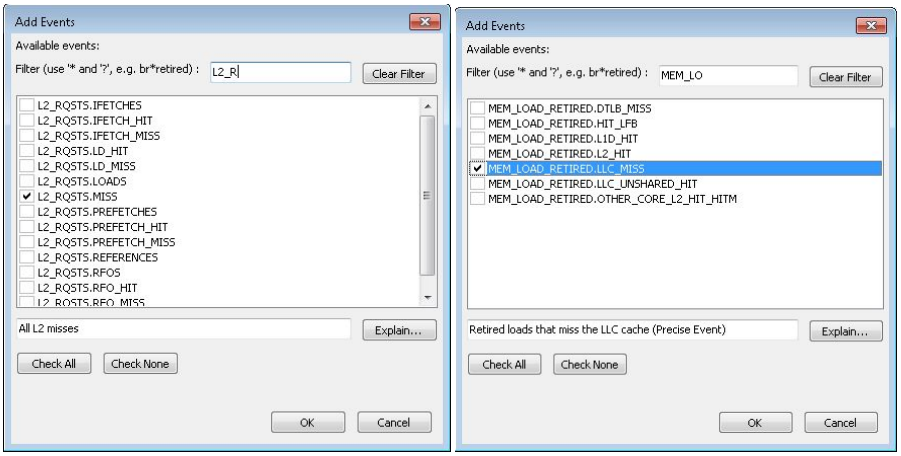


**Рис. 15.** Выбор интересующих событий

В результате, в таблице событий (см. рис. 14) 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. 16), 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. 16.** 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. 17 и рис. 18 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 | 07_pde.exe | CycleReductionMethod |
| __svml_exp2 | 10,000,000 | 0 | 0 | 07_pde.exe | __svml_exp2 |
| crancknikolson | 4,000,000 | 0 | 0 | 07_pde.exe | crancknikolson |

**Рис. 17.** 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. 18.** 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.

# 9. Анализ масштабируемости приложения при использовании TBB-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 TBB-реализация метода циклической редукции.

На рис. 19 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.

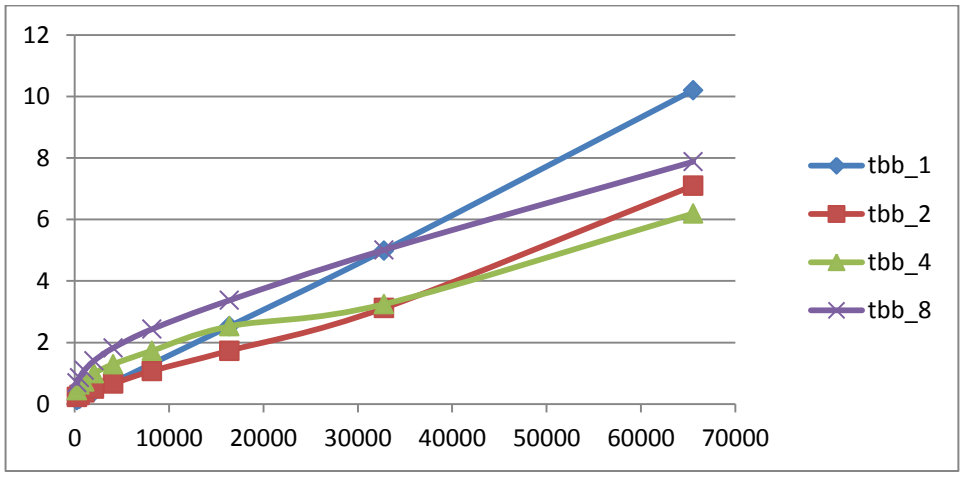


**Рис. 19.** Время решения задачи поиска оптимальной цены КО с использованием параллельной версии циклической редукции

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

| *J* | 1поток | 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 Concurrency-анализа (рис. 20), 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 TBB-реализации распределяется динамически.

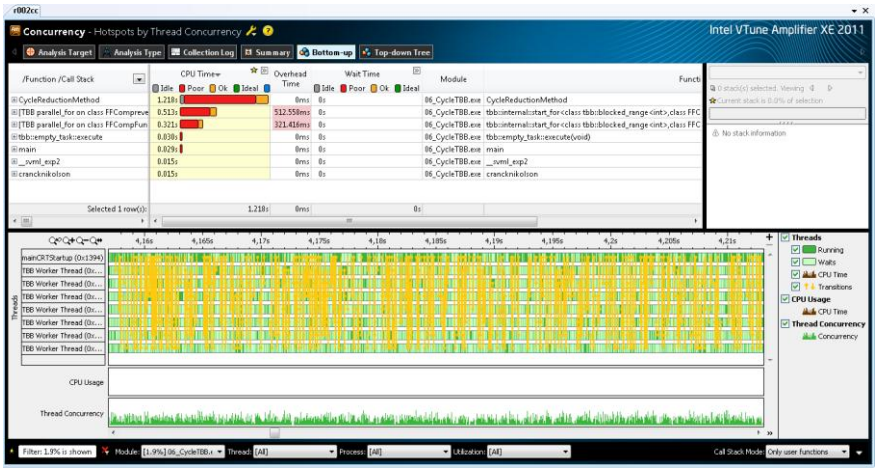


**Рис. 20.** Результаты запуска Intel Parallel Amplifier XE в режиме Concurrency

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.

## 10. Дополнительные задания

1. Обоснуйте применимость метода прогонки для решения СЛАУ с трехдиагональной матрицей, полученной при построении вычислительной схемы Кранка-Николсона в задаче вычисления цены конвертируемой облигации.
2. Оцените погрешность аппроксимации для вычислительной схемы Кранка-Николсона, построенной для задачи вычисления цены КО.
3. Для решения трехдиагональной системы встройте функцию библиотеки MKL. Оцените эффективность использования библиотечных функций по сравнению с прогонкой и последовательной реализацией метода циклической редукции.
4. Реализуйте метод встречной прогонки [1]. Оцените эффективность использования встречной прогонки по сравнению с обычной прогонкой и параллельным методом циклической редукции. Объясните полученные результаты экспериментов.
5. Реализуйте алгоритм блочной прогонки [1]. Оцените эффективность использования со всеми предшествующими реализациями методов решениятрехдиагональных систем. Объясните полученные результаты

экспериментов. Оцените масштабируемость выполненной реализации блочной прогонки.

# 11. Литература

## 11.1. 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.

## 11.2. 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/].

## 11.3. Дополнительная литература

13. 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

14. 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.