Министерство образования и науки Российской Федерации

Федеральное государственное автономное   
образовательное учреждение высшего образования   
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Отчёт по учебной практике

«Вычисление справедливой цены опциона европейского типа»

**Выполнил:** студент ИИТММ гр. 381503-3,

Романов А. А.

**Проверил:** к.т.н., доцент кафедры МОСТ ИИТММ,

Мееров И. Б.

Нижний Новгород

2017 г.

**Оглавление**

[Введение 3](#_Toc503283350)

[1. Основные понятия и обозначения 4](#_Toc503283351)

[2. Постановка задачи 7](#_Toc503283352)

[3. Методы и алгоритмы 8](#_Toc503283353)

[4. Программная реализация 11](#_Toc503283354)

[5. Результаты экспериментов 12](#_Toc503283355)

[5.1. Программно-аппаратное окружение 12](#_Toc503283356)

[5.2. Анализ результатов 12](#_Toc503283357)

[Заключение 14](#_Toc503283358)

[Литература 15](#_Toc503283359)

[Приложение А. Фрагменты программного кода 16](#_Toc503283360)

# **Введение**

Одним из широко распространённых способов получения прибыли в современном финансовом мире является торговля на биржах, в том числе акциями и облигациями. Исходя из возникновения случайных, трудно прогнозируемых событий, влияющих на финансовый рынок, и непрерывной его эволюции, имеет место задача расчёта стоимости купли-продажи активов в кратчайшие сроки (чем быстрее будет получен результат, тем меньше за это время изменится ситуация на рынке).

Так как финансовые расчёты – важная, трудоёмкая и востребованная часть индустрии, область высокопроизводительных вычислений находит своё применение в экономике и занимает там свою нишу. В данной работе будет рассматриваться применение высокопроизводительного программного обеспечения для аналитического вычисления справедливой цены европейского опциона. Также будет показано, что техника программирования и оптимизация алгоритмов прямым образом влияет на время расчётов, столь важное для финансового рынка.

После разработки базового алгоритма вычисления цены будут представлены несколько его оптимизированных версий; ускорение – отношение времени работы базового алгоритма к времени работы ускоренного– будет являться доказательством важности техники программирования и рефакторинга готового кода.

# **Основные понятия и обозначения**

**Финансовый рынок**

**Финансовый рынок** — система отношений, возникающая в процессе купли-продажи товаров, услуг, финансовых благ и т.д., а также перераспределения денежных средств между субъектами рынка.

Участниками рынка могут являться:

* Продавцы и покупатели активов
* Финансовые посредники (инвестиционные компании, брокеры и т.д.)

**Финансовая модель**

(этот раздел написан с использованием материалов из источников [‎2],[‎5].)

Исходя из особенностей организации финансовых рынков различных стран мира выделяют три основные модели: североамериканскую (рыночную, характерные пример – США), европейскую (банковскую, пример – Германия) и смешанную.

К основным характеристикам европейской модели относятся:

низкая доля акционерного капитала, высокая доля финансирования за счет выпуска облигационных займов (соотношение облигаций и акций – 10:1);

доминирующая роль коммерческих банков на финансовом рынке;

традиция прямого кредитования на покрытие дефицита бюджета наряду с выпуском государством ценных бумаг;

высокая доля прямого банковского кредита (50-60%) в финансировании экономики страны;

Для моделирования рынка будем использовать модельБлэка-Шоулза ‎[1], представляющую собой систему дифференциальных уравнений:

(1)

(2)

Уравнение (1) – ОДУ, отражающее изменение цены облигации *.*На влияет *r* – фиксированный процент. Уравнение (2) – стохастическое ДУ, описывающее эволюцию цены акции . Оно содержит:

* *δ*– ставка дивиденда
* волатильность, изменчивость — статистический финансовый показатель, характеризующий изменчивость цены.
* – Винеровский случайный процесс
* – начальная цена акции
* – начальная цена облигации

При некоторых ограничениях система имеет аналитическое решение. В этом случае цену акции можно найти с помощью формулы (3):

(3)

**Европейский опцион**

**Опцион** — договор, по которому покупатель опциона получает право, но не обязательство, купить или продать актив (акцию – рисковый актив, облигацию – безрисковый актив) по заранее оговоренной цене в течение определенного промежутка времени. В свою очередь, продавец опциона обязан продать актив или купить его у покупателя опциона в соответствии с оговоренными заранее условиями.

Европейский опцион (call) исключает возможность исполнения прав до оговоренного срока. Исполнение договора может состояться только в самый последний день его действия либо в ту дату, которая была согласована двумя сторонами. Любая попытка исполнить договор раньше срока приводит к наложению штрафов на соответствующую сторону

После заключении договора между сторонами *P*1 и *P*2 происходит следующее: Вторая сторона выплачивает денежные средства *C* и в некоторый фиксированный момент времени *T* решает, стоит ли покупать акции по цене *K* у первой стороны. Решение принимается исходя из соотношения цены акции в момент иначальной цены . Если , покупать акции бессмысленно, сторона *P*1получает прибыль *C*, сторона*P*2 теряет *C*. В противном случае*P2* покупает у *P1* акции по цене *K*, в ряде случаев получая прибыль (зависит от соотношения C и ).

**Проблема заключается в нахождении справедливой цены европейского опциона** - минимальной цены контракта, при которой соблюдается баланс между выигрышем и проигрышем каждой из сторон. Часто справедливую цену называют рациональной стоимостью опциона.

Определим эту цену как средний выигрыш второй стороны:

, где (4)

*E* – математическое ожидание (зависит в том числе от стохастических факторов). Произведение экспоненты и скобкиотражает дисконтирование–инфляцию при фиксированной процентной ставке *r*.

Идея образования опционов была поднята Фишером Блэком, Майроном Шоулзом и Робертом Мертоном в 1973 году[‎4]. Выведенная ими формула теперь известна как формула Блэка-Шоулза, за которую в 1997 году Шоулз и Мертон получили Нобелевскую премию.

С помощью этой формулы найдём величину :

(5)

– функция стандартного нормального распределения,

Именно эта формула ляжет в основу алгоритма, вычисляющего справедливую цену.

# **Постановка задачи**

Пусть известны волатильность, процентная ставка, время исполнения опциона, начальная его цена и цена исполнения опциона . Необходимо разработать базовый алгоритм, вычисляющий справедливую цену для набора опционов с помощью формулы (5). После этого оптимизировать его с целью уменьшения времени расчёта. После получения результатов работы алгоритмов на высокопроизводительной системе сравнить время работы базовой и модернизированной версий и сделать вывод о влиянии приёмов программирования на результат.

# **Методы и алгоритмы**

**Базовая версия алгоритмаV0**

Представляет собой формулу (5) Блэка-Шоулза:

void \_V0(float \*pT, float \*pK, float \*pS0, float \*pC)

{

int i;

float d1, d2, p1, p2;

for (i = 0; i < N; i++)

{

d1 = (log(pS0[i] / pK[i]) + (r + sig \* sig \* 0.5) \*

pT[i]) / (sig \* sqrt(pT[i]));

d2 = (log(pS0[i] / pK[i]) + (r - sig \* sig \* 0.5) \*

pT[i]) / (sig \* sqrt(pT[i]));

vsCdfNorm(1, &d1, &p1);

vsCdfNorm(1, &d2, &p2);

pC[i] = pS0[i] \* p1 - pK[i] \*

exp((-1.0) \* r \* pT[i]) \* p2;

}

}

Для увеличения времени работы программы будем считать в цикле N европейских опционов.

Предварительно объявим массивы pT, pK, pS0, pC, в которых будем хранить время исполнения опциона, фиксированную, начальную и справедливую цену соответственно.

Сравним получившуюся цену pC[0] с верным результатом,

TEST(EU\_OP, test\_V0)

{

float \*pT, \*pK, \*pS0;

floatpC;

\*pT = T;

\*pS0 = S0;

\*pK = K;

option\_array[0](pT, pK, pS0, &pC);

ASSERT\_EQ(pC, float(20.9244));

}

после чего говорим о корректности работы алгоритма.

**Версия V1. Исключение ненужных преобразований типов**

Для вычисления справедливой цены достаточно использовать single precision, поэтому заменим функции log, sqrt, exp на соответствующие logf, sqrtf, expf и добавим к числам суффикс **f**.

**Версия V2. Эквивалентные преобразования**

Функцию vsCdfNorm() можно заменить на erff(), так как vsCdfNorm() == 0.5f + erff().

Изменим код:

erf1 = 0.5f + 0.5f \* erff(d1 / sqrtf(2.0f));

erf2 = 0.5f + 0.5f \* erff(d2 / sqrtf(2.0f));

pC[i] = pS0[i] \* erf1 - pK[i] \* expf((-1.0f) \* r \*pT[i]) \* erf2;

**Версия V3. Ключевое слово restrict**

Сначала необходимо сменить режим компиляции с SSEна AVX, используя ключ –xAVX. После этого добавим ключевое слово restrictв декларацию функции:

void \_V3(float\* restrict pT, float\* restrict pK, float\* restrict pS0, float\* restrict pC);

Restrictозначает, что на данную область памяти не будут ссылаться другие указатели, то есть массивы не пересекаются, и их можно векторизовать

**Версия V4. Использование директив**

Ещё один способ векторизации заключается в использовании перед циклом директив #pragma simd и #pragma vector always, дающих знать компилятору о том, что с точки зрения программиста массивы не пересекаются и о том, что если векторизация возможна, то она эффективна.

#pragma simd

#pragma vector always

for (i = 0; i < N; i++)

{…}

**Версия V5. Вынос инвариантов.**

Чтобы сэкономить время на многократном подсчёте значения, используем изначально объявленную константу.

const float invsqrt2 = 0.707106781f;

Заменим соответствующие строки.

erf1 = 0.5f + 0.5f \* erff(d1 \* invsqrt2);

erf2 = 0.5f + 0.5f \* erff(d2 \* invsqrt2);

**Версия V6. Эквивалентные преобразования. Вычисление квадратного корня.**

Заменим деление умножением, это ещё больше уменьшит время работы программы. Заменим1**f** / sqrtf(pT[i]) на функцию vsInvSqrt()из библиотеки MKL и проверим, сделал ли компилятор замену деления умножением самостоятельно.

**Версия V7. Параллельная версия.**

Используем библиотеку omp.h для распараллеливания цикла. Перед началом цикла добавим

#pragma omp parallel for private(invf, d1, d2, erf1, erf2)

**Версия V8. Оптимизация кэша.**

Из четырёх массивов, с которыми мы работаем, только три используются для чтения. В четвёртый (pС) происходит запись результатов, которые в цикле никак не используются, поэтому имеет смысл записывать массив pCнапрямую в память, минуя кэш. При таком подходе будут уменьшены накладные расходы на перенос данных, будет сэкономлено по одной операции чтения из кэша за итерацию цикла.

Для того, чтобы дать понять компилятору об отсутствии необходимости кэшировать четвёртый массив, используем #pragma vector nontemporal.

# **Программная реализация**

* 1. **Структура проекта**

Программный комплекс состоит из трёх проектов:

1. main\_proj. Содержит декларации переменных и функций. Также включает реализацию всех алгоритмов.
2. gtest. Статическая библиотека с google-тестами.
3. tests. Содержит реализацию тестового покрытия алгоритмов.
   1. **Основные структуры данных**

Объявленные переменные:

int num\_Threads;

int N; //amount of options

int version;

double \_time;

double start, finish;

const float invsqrt2 = 0.707106781f;

const float sig = 0.2f; // volatility; percent per year 0.2 -> 20%

const float r = 0.05f; // the interest rate; percent per year 0.05 -> 5%

const float T = 3.0f; // option execute time (years)

const float S0 = 100.0f; // option price at t == 0;

const float K = 100.0f; // strike price -- price fixed in option

Новый тип данных:

Typedef void(\*GetPrices)(float \*pT, float \*pK, float \*pS0, float \*pC);

Массив указателей на функции, содержащие модифицированные алгоритмы:

GetPrices option\_array[9] =

{

\_V0, //preference 1

\_V1, //preference 2

\_V2, //erf

\_V3, //restrict

\_V4, //#pragma simd #pragma vector always

\_V5, //#pragma simd invsqrt2\_1

\_V6, //#pragma simd invsqrt2\_2

\_V7, //#pragma simd #pragma omp parallel for private

\_V8, // \_V7 + #pragma vector nontemporal

};

# **Результаты экспериментов**

# **Программно-аппаратное окружение**

**Программное обеспечение:**

1. Операционная система Linux CentOS 6.2
2. IDLE: Microsoft Visual Studio 2015
3. Компилятор Intel C++ 17.0
4. Библиотеки Intel OpenMP, Intel MKL

**Аппаратное обеспечение:**

* 2 процессора Intel Xeon Phi 5110P (60 ядер, 240 потоков, память 8 Gb).
* Сетевое оборудование на основе GigabitEthernet.

Пиковая производительность СК – 573 Tflops

# **Анализ результатов**

Был написан скрипт, запускающий 15 раз на исполнение определённую версию алгоритма и записывающий в лог минимальное время работы программы и сопутствующую информацию.

* сравнение времени работы последовательных версий кода
* сравнение времени работы параллельных версий кода (16 потоков)
* эффективность масштабируемости параллельной реализации для версии 8

Результаты экспериментов приведены в таблицах 1 – 3.

Количество опционов N = . Каждый запуск выполнен 15 раз, приведено минимальное время работы в миллисекундах.

Таблица 1. сравнение времени работы последовательных версий кода

|  |  |
| --- | --- |
| **Версия** | **Минимальное время** |
| preference 1 | 18530 |
| preference 1 | 16440 |
| erf | 5832 |
| restrict | 1069 |
| #pragma simd #pragma vector always | 1070 |
| #pragma simd invsqrt2\_1 | 1068 |
| #pragma simd invsqrt2\_2 | 5678 |

Существенное замедление алгоритма V6 объясняется дополнительными затратами времени на две операции чтения, операцию записи и операцию умножения.

Float tmp = sig2 \* pT[i];

vsInvSqrt(1, &tmp, &invf);

Таблица 2. Сравнение времени работы параллельных версий кода (16 потоков).

|  |  |
| --- | --- |
| **Версия** | **Минимальное время** |
| #pragma simd #pragma omp parallel for private | 99 |
| \_V7 + #pragma vector nontemporal | 106 |

Время работы алгоритмов V7 и V8 совпадает с точностью до , это означает, что мощности кластера достаточно для того, чтобы хранить в кэше все четыре массива без потери производительности.

Таблица 3. Эффективность масштабируемости параллельной реализации для версии V8

|  |  |  |
| --- | --- | --- |
| **Число потоков** | **Минимальное время** | **Ускорение** |
| **1** | 1123 | - |
| **2** | 577 | 1,964 |
| **4** | 305 | 3,682 |
| **8** | 146 | 7,692 |
| **16** | 106 | 10,594 |

Можно заметить, что параллельная версия имеет ускорение, примерно совпадающее с числом потоков (В случае 16 потоков имеем ускорение 10,5 вместо 16 из-за значительных накладных расходов на создание и уничтожение потоков)

После всех оптимизаций алгоритма лучший результат показал V8, сократив время работы по сравнению с базовой версией в раза.

# **Заключение**

Была разработана начальная версия алгоритма, вычисляющего справедливую цену опциона по формуле 5. Было разработано 8 различных модификаций базовой версии, каждая из которых давала значительное ускорение. Наибольшее ускорение было получено в последней параллельной версии алгоритма – время работы программы уменьшилось в 175 раз.

Из-за большого объёма однообразных действий возникла потребность в автоматизации работы программного комплекса. Для решения проблемы использовался высокоуровневый скриптовый язык python.

Полученное ускорение в работе алгоритмов показывает важность техники программирования и необходимости добиться максимальной оптимизации рабочего кода в тех задачах, где важна скорость получения точного результата.

# **Литература**

1. Black F., Scholes M. The pricing of options and corporate liabilities // Journal of political economy – 1973 – Т. 81 – №. 3 – С. 637–654. https://www.cs.princeton.edu/courses/archive/fall09/cos323/papers/black\_scholes73.pdf
2. Оптимизация расчетов на примере задачи вычисления справедливой цены опциона Европейского типа. <https://www.intuit.ru/studies/courses/14004/1095/lecture/22917>
3. Справедливая стоимость опциона. <https://utmagazine.ru/posts/17075-spravedlivaya-stoimost-opciona>
4. Модель Блэка-Шоулза <https://en.wikipedia.org/wiki/Black–Scholes_model>
5. Модели финансового рынка и их особенности <https://studopedia.su/10_149780_modeli-finansovogo-rinka-i-ih-osobennosti.html>
6. Документация MKL<http://old.parallel.ru/ftp/libs/mkl/mklman52.pdf>

# **Приложение А. Фрагменты программного кода**

void \_V8(float \*pT, float \*pK, float \*pS0, float \*pC)

{#pragma simd

#pragma vector nontemporal

#pragma omp parallel for private(d1, d2, erf1, erf2)

for (i = 0; i < N; i++)

{

d1 = (logf(pS0[i] / pK[i]) + (r + sig \* sig \* 0.5f) \*

pT[i]) / (sig \* sqrtf(pT[i]));

d2 = (logf(pS0[i] / pK[i]) + (r - sig \* sig \* 0.5f) \*

pT[i]) / (sig \* sqrtf(pT[i]));

erf1 = 0.5f + 0.5f \* erff(d1 / sqrtf(2.0f));

erf2 = 0.5f + 0.5f \* erff(d2 / sqrtf(2.0f));

pC[i] = pS0[i] \* erf1 - pK[i] \* expf((-1.0f) \* r \* pT[i]) \* erf2;

}

}

int main(int argc, char \*argv[])

{

version = atoi(argv[1]);

N= atoi(argv[2]);

num\_Threads = atoi(argv[3]);

float\* pT = new float[4 \* N];

float\* pK = pT + N;

float\* pS0= pT + 2 \* N;

float\* pC = pT + 3 \* N;

for (int i = 0; i < N; i++)

{

pT[i] = T;

pS0[i] = S0;

pK[i] = K;

}

start = omp\_get\_wtime();

option\_array[version](pT, pK, pS0, pC);

finish = omp\_get\_wtime();

\_time = finish - start;

std::cout << \_time << std::endl;

std::cout << pC[1];

delete[] pT;

return 0;

}