

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ М.В. ЛОМОНОСОВА

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Отчет о проекте, выполненном в рамках курса:

«ПРОГРАММНАЯ ИНЖЕНЕРИЯ и C++ для количественного
анализа и алгоритмической торговли»

Parallel Monte Carlo Option Pricing

Выполнила:

Студентка 5 курса
Долгих Елизавета

Преподаватель:

Сергей Владимирович Шелягин

Москва - 2025

Содержание

1 Введение	2
2 Требования	2
2.1 Функциональные требования	2
2.2 Нефункциональные требования	3
3 Математическая модель	4
3.1 Стохастическая динамика актива	4
3.2 Оценка стоимости методом Монте-Карло	4
3.3 Азиатские опционы	4
3.4 Метод антитетических переменных	5
3.5 Вычисление чувствительностей (Greeks)	5
4 Архитектура и проектирование программного обеспечения	6
4.1 Общая структура компонентов	6
4.2 Паттерн Стратегия	7
4.3 Движок Монте-Карло	7
4.4 Модель многопоточности	7
4.5 Визуализация архитектуры (UML)	7
5 Технические особенности реализации	10
5.1 Генерация псевдослучайных чисел (RNG)	10
5.2 Потокобезопасность и Детерминизм	10
5.3 Оптимизация компилятора и стандарты	10
6 Инфраструктура и контроль качества	11
6.1 Система сборки CMake	11
6.2 Модульное тестирование (Unit Testing)	11
6.3 Непрерывная интеграция (CI/CD)	11
6.4 Демонстрация работы Quality Gate	12
7 Анализ производительности	14
7.1 Методика тестирования	14
7.2 Результаты измерений	14
7.3 Анализ масштабируемости	15
7.4 Архитектурные оптимизации и работа с памятью	15
8 Заключение	16
8.1 Основные результаты	16
8.2 Направления дальнейшего развития	16

1 Введение

В данной работе рассматривается разработка высокопроизводительного программного комплекса для оценки стоимости финансовых деривативов методом Монте-Карло. Проект направлен на создание инструмента, соответствующего стандартам промышленной разработки.

Оценка стоимости опционов является одной из фундаментальных задач в количественных финансах. В то время как для базовых европейских опционов существуют аналитические решения (формула Блэка-Шоулза), оценка более сложных инструментов, таких как азиатские опционы или опционы с зависящей от пути выплатой, требует применения численных методов.

Метод Монте-Карло является стандартом для таких задач благодаря своей универсальности и простоте реализации. Однако его главным недостатком является высокая вычислительная стоимость: для достижения приемлемой точности необходимо смоделировать миллионы траекторий цены актива.

В связи с этим актуальной задачей является разработка программного обеспечения, способного эффективно утилизировать ресурсы современных многоядерных процессоров для ускорения расчетов.

2 Требования

Для обеспечения качества и надежности системы были сформулированы функциональные и нефункциональные требования.

2.1 Функциональные требования

Система обеспечивает выполнение следующих функций:

- **Оценка стоимости Европейских опционов.** Реализация алгоритма Монте-Карло для вычисления справедливой стоимости опционов Call и Put с возможностью настройки параметров с рынка (S_0, K, T, r, σ).
- **Поддержка экзотических опционов.** Реализация механизма оценки Азиатских опционов (среднее арифметическое цены актива) для демонстрации гибкости архитектуры и поддержки опционов, зависящих от траектории цены.
- **Расчет коэффициентов чувствительности (Греков).** Автоматический расчет параметров Delta (Δ) и Gamma (Γ) с использованием метода конечных разностей для оценки рисков портфеля.
- **Валидация и верификация.** Наличие модуля сравнения численных результатов с аналитическим решением (формула Блэка-Шоулза) для контроля точности вычислений в реальном времени.
- **Понижение дисперсии.** Применение метода антитетических переменных для уменьшения стандартной ошибки оценки и ускорения сходимости алгоритма.
- **Экспорт результатов.** Возможность сохранения рассчитанных цен, греков и метрик производительности в структурированный файл формата CSV для последующей аналитики.

2.2 Нефункциональные требования

К системе предъявляются следующие архитектурные и системные требования:

- **Производительность и масштабируемость.** Архитектура приложения должна обеспечивать эффективное распараллеливание задачи на многоядерных процессорах. Целевой показатель ускорения на 6 физических ядрах — не менее $4x$ по сравнению с однопоточным исполнением.
- **Точность вычислений.** Относительная погрешность численного метода по сравнению с аналитическим решением не должна превышать 5% при количестве симуляций $N \geq 10^6$.
- **Детерминированность.** Система должна гарантировать полную воспроизводимость результатов при повторном запуске с фиксированным начальным значением генератора псевдослучайных чисел.
- **Надежность и качество кода.** Исходный код должен соответствовать стандарту C++17, проходить проверки статического анализатора Clang-Tidy с флагом `WarningsAsErrors` и быть покрыт модульными тестами (GoogleTest).
- **Переносимость (Portability).** Система сборки должна быть реализована на базе CMake и обеспечивать корректную компиляцию и запуск на операционных системах Linux, Windows и macOS без модификации исходного кода.
- **Документация.** Должна быть автоматическая генерация документации (Doxygen).

3 Математическая модель

В основе разработанного программного обеспечения лежит модель Блэка-Шоулза-Мертона, предполагающая, что цена базового актива эволюционирует во времени как геометрическое броуновское движение.

3.1 Стохастическая динамика актива

В риск-нейтральной мере \mathbb{Q} динамика цены актива S_t описывается стохастическим дифференциальным уравнением (СДУ):

$$dS_t = rS_t dt + \sigma S_t dW_t, \quad (1)$$

где:

- r — безрисковая процентная ставка,
- σ — волатильность актива,
- W_t — стандартный винеровский процесс.

Применяя лемму Ито к функции $f(S_t) = \ln S_t$, мы получаем точное решение данного уравнения для момента времени T :

$$S_T = S_0 \exp \left(\left(r - \frac{\sigma^2}{2} \right) T + \sigma \sqrt{T} Z \right), \quad (2)$$

где $Z \sim \mathcal{N}(0, 1)$ — стандартная нормальная случайная величина.

3.2 Оценка стоимости методом Монте-Карло

Справедливая стоимость опциона V_0 в момент времени $t = 0$ равна математическому ожиданию дисконтированной функции выплаты в риск-нейтральной мере:

$$V_0 = e^{-rT} \mathbb{E}^{\mathbb{Q}}[\text{Payoff}(S_T)]. \quad (3)$$

Численная оценка данного интеграла методом Монте-Карло производится как среднее значение по N симуляциям:

$$\hat{V}_{MC} = e^{-rT} \frac{1}{N} \sum_{i=1}^N \text{Payoff}(S_T^{(i)}). \quad (4)$$

Для Европейского Call-опциона функция выплаты имеет вид $\max(S_T - K, 0)$, для Put-опциона — $\max(K - S_T, 0)$.

3.3 Азиатские опционы

Для оценки азиатских опционов выплата зависит от средней цены актива за период жизни опциона. Траектория цены моделируется пошагово на дискретной временной сетке t_0, t_1, \dots, t_M , где $\Delta t = T/M$:

$$S_{t_{j+1}} = S_{t_j} \exp \left(\left(r - \frac{\sigma^2}{2} \right) \Delta t + \sigma \sqrt{\Delta t} Z_j \right). \quad (5)$$

Выплата для Азиатского Call-опциона определяется как $\max(A_T - K, 0)$, где $A_T = \frac{1}{M} \sum_{j=1}^M S_{t_j}$ — среднее арифметическое цен.

3.4 Метод антитетических переменных

Для повышения точности оценки и ускорения сходимости алгоритма применен метод антитетических переменных.

Суть метода заключается в том, что для каждой сгенерированной случайной величины $Z \sim \mathcal{N}(0, 1)$ мы используем пару значений $(Z, -Z)$ для генерации двух траекторий цены:

$$S_T^+ = S_0 \exp \left(\mu T + \sigma \sqrt{T} Z \right), \quad (6)$$

$$S_T^- = S_0 \exp \left(\mu T + \sigma \sqrt{T} (-Z) \right). \quad (7)$$

При $\text{Cov}(X, \tilde{X}) < 0$ дисперсия среднего значения уменьшается:

$$\text{Var} \left(\frac{X + \tilde{X}}{2} \right) = \frac{1}{4} \left(\text{Var}(X) + \text{Var}(\tilde{X}) + 2\text{Cov}(X, \tilde{X}) \right) \quad (8)$$

и итоговая ошибка уменьшается быстрее, чем при простом увеличении числа путей.

Оценка стоимости опциона рассчитывается как среднее значение по парам:

$$\hat{V}_{AV} = e^{-rT} \frac{1}{N} \sum_{i=1}^N \frac{\text{Payoff}(S_{T,i}^+) + \text{Payoff}(S_{T,i}^-)}{2} \quad (9)$$

Данный подход позволяет существенно снизить стандартную ошибку оценки без увеличения количества генераций псевдослучайных чисел.

**В данной версии метод Antithetic Variates реализован только для Европейских опционов, так как это основной сценарий использования. Для Азиатских опционов используется стандартный Монте-Карло, но архитектура позволяет легко добавить туда AV в будущем.*

3.5 Вычисление чувствительностей (Greeks)

Для расчета параметров чувствительности (Греков) применяется метод конечных разностей с центральной разностью (для обеспечения второго порядка точности $O(h^2)$). Пусть $V(S)$ - функция цены опциона от цены базового актива. Тогда Delta Δ (чувствительность цены опциона к изменению цены базового актива) и Gamma Γ (скорость изменения Дельты (вторая производная по цене)) аппроксимируются следующим образом:

$$\Delta \approx \frac{V(S + h) - V(S - h)}{2h} \quad (10)$$

$$\Gamma \approx \frac{V(S + h) - 2V(S) + V(S - h)}{h^2} \quad (11)$$

где h — малый шаг приращения цены, выбранный как $h = \max(S_0 \cdot 10^{-4}, 10^{-4})$. Данный подход позволяет вычислять греки одновременно с ценой, переиспользуя логику симуляции.

4 Архитектура и проектирование программного обеспечения

Архитектура программного комплекса была спроектирована с целью удовлетворения функциональных и нефункциональных требований, сформулированных в Главе 2. Выбор ключевых технологий и паттернов проектирования обусловлен необходимостью обеспечения гибкости, производительности и надежности системы.

Ниже приведено обоснование перехода от требований к конкретным архитектурным решениям:

- **Реализация гибкости расчетов → Паттерн Strategy.**

Использован паттерн Стратегия для отделения алгоритма выплаты от вычислительного ядра, что позволяет добавлять новые инструменты без изменения кода движка.

- **Обеспечение производительности → Task-based Parallelism.**

Требование линейной масштабируемости на многоядерных системах продиктовало отказ от низкоуровневого управления потоками (`std::thread`) в пользу высокоуровневых абстракций (`std::async`).

- **Гарантия детерминизма и потокобезопасности → Thread-Local RNG.**

Требование воспроизводимости результатов в многопоточной среде делает невозможным использование глобального генератора случайных чисел (из-за необходимости блокировок и недетерминированного порядка вызовов). Архитектурным решением стало внедрение локальных генераторов (`thread_local`) для каждого вычислительного потока.

- **Надежность и управление ресурсами → Smart Pointers.**

Управление временем жизни динамических объектов (стратегий выплат) делегировано умным указателям `std::shared_ptr`, что обеспечивает автоматическое освобождение ресурсов и безопасное совместное владение объектами в многопоточном окружении.

Далее рассматривается детальная реализация компонентов системы, спроектированных на основе данных принципов.

Архитектура библиотеки спроектирована в соответствии с принципами объектно-ориентированного программирования (ООП) и SOLID, обеспечивая модульность, расширяемость и безопасность управления памятью.

4.1 Общая структура компонентов

Система разделена на три логических слоя:

1. **Ядро симуляции:** Отвечает за генерацию стохастических процессов и управление потоками.
2. **Логика финансовых инструментов:** Инкапсулирует правила расчета выплат (Payoffs).
3. **Аналитический модуль:** Предоставляет точные решения для верификации результатов.

4.2 Паттерн Стратегия

Базовый абстрактный класс `Payoff` определяет интерфейс функтора:

```
1 class Payoff {
2 public:
3     virtual ~Payoff() = default;
4     [[nodiscard]] virtual double operator()(double spot) const noexcept = 0;
5     [[nodiscard]] virtual std::string name() const = 0;
6 };
```

Листинг 1: Интерфейс `Payoff` (src/Payoff.hpp)

Конкретные реализации (Call, Put, Asian) наследуются от этого интерфейса. Такой подход соответствует принципу открытости/закрытости (Open/Closed Principle): добавление нового типа опциона (например, Digital Option) не требует изменения кода `MonteCarloEngine`.

4.3 Движок Монте-Карло

Класс `MonteCarloEngine` является центральным компонентом системы. Он агрегирует параметры с рынка (S_0, r, σ, T) и владеет указателем на стратегию выплаты.

Владение объектом `Payoff` осуществляется через умный указатель `std::shared_ptr`.

4.4 Модель многопоточности

Для реализации параллелизма используется высокоуровневая абстракция `std::async` и `std::future`.

Архитектура распараллеливания выглядит следующим образом:

1. Общее число симуляций N делится на K потоков (где K определяется аппаратно через `std::thread::hardware_concurrency`).
2. Метод `calculatePrice` запускает K асинхронных задач (`runSimulationChunk`).
3. Каждая задача имеет свой локальный генератор случайных чисел, инициализированный уникальным зерном (`seed`), и накапливает частичные суммы в локальных переменных, что исключает состояние гонки (`data race`) и необходимость в мьютексах.
4. Главный поток ожидает завершения всех `std::future` и агрегирует результаты.

Данная архитектура минимизирует накладные расходы на синхронизацию (lock-free design) и обеспечивает линейную масштабируемость.

4.5 Визуализация архитектуры (UML)

Для наглядного представления структуры классов и взаимодействия компонентов приведены диаграммы на языке UML.

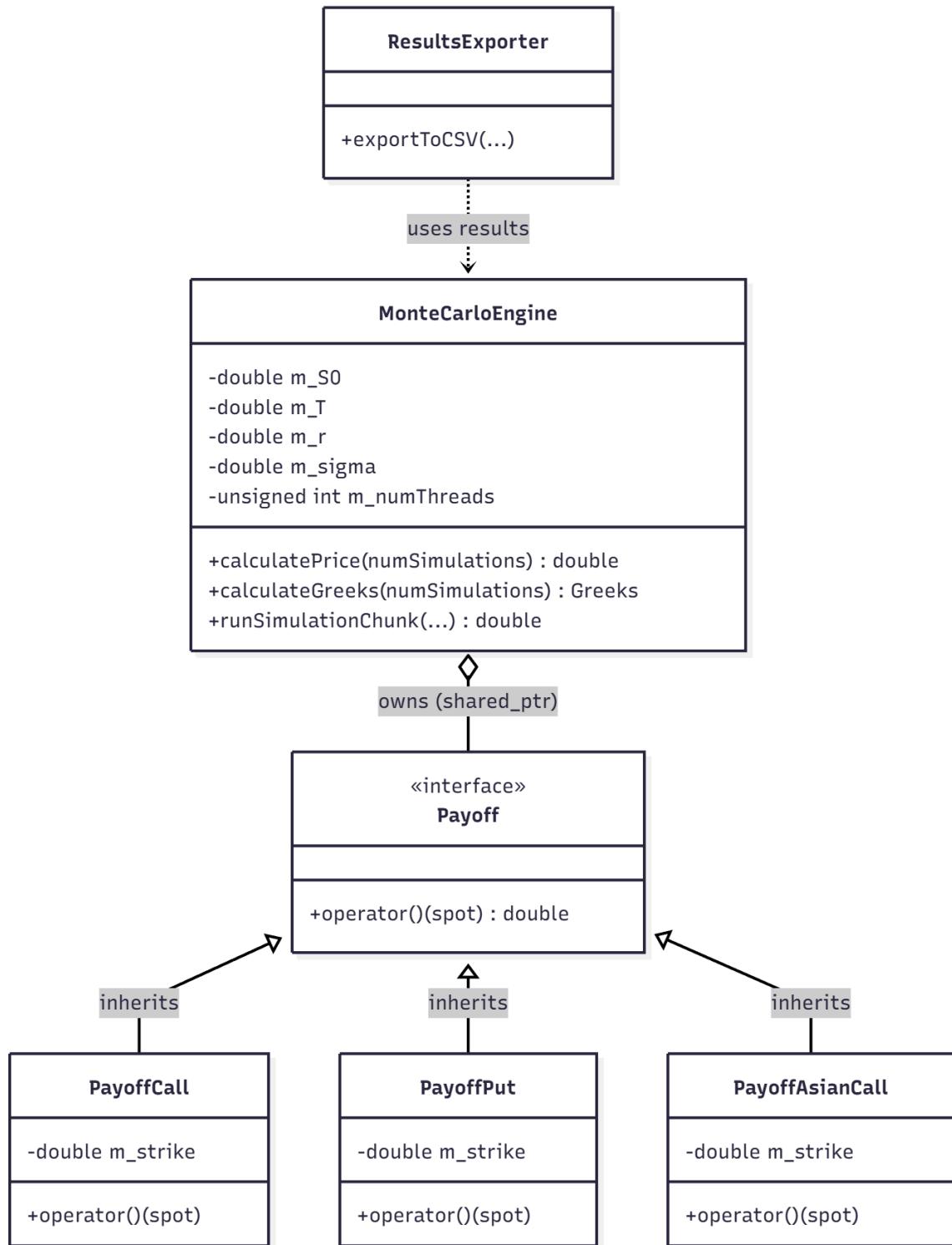


Рис. 1: Диаграмма классов (Class Diagram). Демонстрация паттерна Strategy.

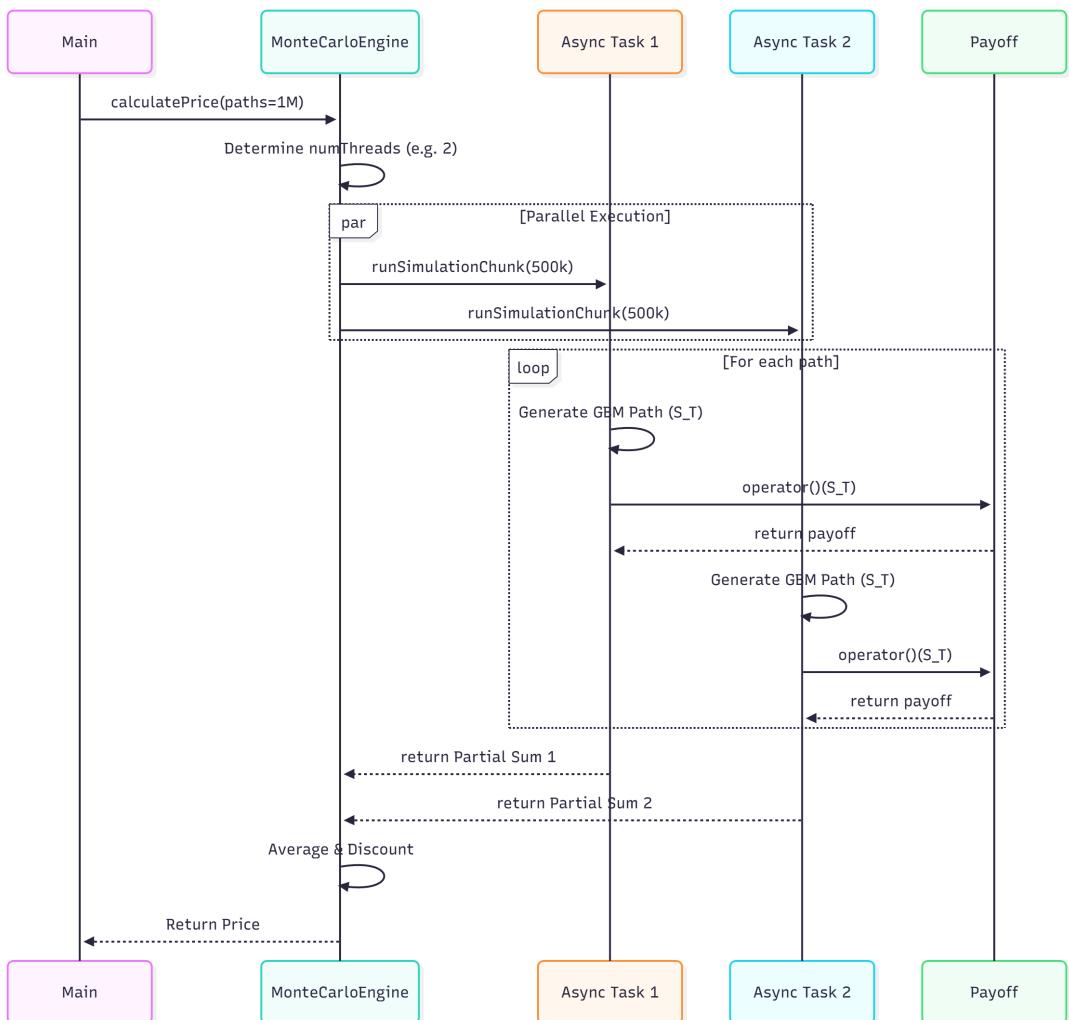


Рис. 2: Диаграмма последовательности (Sequence Diagram). Жизненный цикл параллельного расчета.

5 Технические особенности реализации

В данном разделе рассматриваются ключевые программные решения, обеспечивающие производительность и корректность работы библиотеки.

5.1 Генерация псевдослучайных чисел (RNG)

Качество метода Монте-Карло напрямую зависит от качества генератора псевдослучайных чисел. Стандартный `std::rand()` не подходит для финансовых симуляций из-за короткого периода повторения и низкой статистической надежности.

В проекте используется 64-битный вихрь Мерсенна (`std::mt19937_64`). Это обеспечивает:

- Огромный период повторения ($2^{19937} - 1$), достаточный для миллиардов симуляций.
- Высокую скорость генерации.
- 64-битную точность, необходимую для работы с типом `double`.

5.2 Потокобезопасность и Детерминизм

Одной из главных проблем параллельного Монте-Карло является корректная инициализация генераторов в потоках. Если использовать один глобальный генератор с мьютексом, производительность упадет из-за блокировок. Если использовать `time(0)` для инициализации локальных генераторов, потоки, запущенные одновременно, могут получить одинаковые последовательности чисел.

В проекте реализована схема со смещением зерна (seed offset):

```
1 double MonteCarloEngine::runSimulationChunk(...,
2                                     unsigned long long chunkIndex) const {
3     // Уникальный seed для каждого потока
4     // m_seed - глобальное зерно, chunkIndex - номер потока
5     std::mt19937_64 rng(m_seed + chunkIndex);
6
7     std::normal_distribution<double> dist(0.0, 1.0);
8     // ...
9 }
```

Листинг 2: Инициализация RNG в потоке (MC Engine.cpp)

Это гарантирует, что каждый поток работает с уникальной, независимой последовательностью чисел, и при этом результат полностью воспроизводим при повторном запуске с тем же `m_seed`.

5.3 Оптимизация компилятора и стандарты

В проекте используются современные возможности стандарта C++17/20:

- `[[nodiscard]]`: Атрибут, предупреждающий, если возвращаемое значение функции (например, цена опциона) игнорируется.
- `constexpr`: Использование констант времени компиляции (в файле `Constants.hpp`) для математических констант (π , $\sqrt{2}$), что позволяет избежать вычислений в runtime.
- `noexcept`: Методы класса `Payoff` помечены как `noexcept`, что позволяет компилятору генерировать более эффективный код, так как не требуется создавать таблицы размотки стека для исключений.

6 Инфраструктура и контроль качества

Разработка велась с соблюдением практик CI/CD, что обеспечивает надежность и поддерживаемость кода.

6.1 Система сборки CMake

Проект использует кроссплатформенную систему сборки **CMake** (версия 3.14+). Конфигурация `CMakeLists.txt` обеспечивает:

- Автоматическую загрузку зависимостей (GoogleTest) через `FetchContent`.
- Настройку флагов компиляции для разных стандартов C++ (C++17/20).
- Интеграцию статического анализатора `clang-tidy` для автоматической проверки стиля и поиска потенциальных ошибок при каждой сборке.

6.2 Модульное тестирование (Unit Testing)

Для верификации корректности вычислений используется фреймворк **GoogleTest**. Набор тестов (в директории `tests/`) покрывает следующие сценарии:

1. **Валидация аргументов:** Проверка того, что движок выбрасывает исключения при некорректных данных (например, отрицательная волатильность или $T = 0$).
2. **Сходимость к аналитике:** Сравнение результатов Монте-Карло с формулой Блэка-Шоулза (допустимая погрешность $\epsilon < 0.5\%$).
3. **Финансовая состоятельность:** Тест `PutCallParityMC` проверяет выполнение фундаментального паритета опционов:

$$C - P = S_0 - Ke^{-rT} \quad (12)$$

4. **Воспроизводимость:** Тест `Reproducibility` гарантирует, что два запуска с одинаковым зерном (seed) дают идентичный финансовый результат.

6.3 Непрерывная интеграция (CI/CD)

В репозитории настроен пайплайн **GitHub Actions**, который автоматически запускается при каждом коммите и pull request.

Этапы пайплайна (Workflow):

- **Build Matrix:** Проект собирается параллельно на трех операционных системах: Ubuntu (GCC/Clang), Windows (MSVC) и macOS (Clang). Это гарантирует переносимость кода.
- **Run Tests:** После сборки автоматически запускается `ctest`. Если хотя бы один тест падает, сборка помечается как неуспешная.
- **Documentation Deploy:** Отдельный workflow автоматически генерирует документацию из комментариев в коде с помощью Doxygen и публикует её на GitHub Pages.

6.4 Демонстрация работы Quality Gate

Для обеспечения строгого соблюдения стандартов кодирования (Google C++ Style Guide) и предотвращения попадания в репозиторий потенциально опасных конструкций, в системе реализован механизм **Quality Gate**.

В конфигурации CMake для статического анализатора Clang-Tidy активирован режим `WarningsAsErrors` («предупреждения как ошибки»). Это означает, что любое нарушение стиля автоматически прерывает процесс сборки (Build Failure), не позволяя скомпилировать проект.

Конфигурация проверок включает:

- `bugprone-*` — поиск потенциальных ошибок;
- `modernize-*` — рекомендации по использованию стандартов C++17/20;
- `performance-*` — оптимизация производительности;
- `cppcoreguidelines-*` — соблюдение рекомендаций C++ Core Guidelines.

Эксперимент. Для проверки надежности системы защиты в исходный код была намеренно внесена «диверсия» — использование приведения типов в стиле Си (C-style cast), которое запрещено современными стандартами C++:

```
1 double pi = 3.14159;
2 // Нарушение правила: cppcoreguidelines-pro-type-cstyle-cast
3 int pi_int = (int)pi;
```

Листинг 3: Пример небезопасного кода (диверсия)

Данная конструкция является небезопасной, так как игнорирует проверки типов во время компиляции и может привести к потере данных, в отличие от рекомендуемого оператора `static_cast<int>(...)`.

Результат. Система непрерывной интеграции (CI) на базе Ubuntu автоматически обнаружила нарушение и заблокировала сборку проекта. В логах пайплайна была зафиксирована следующая ошибка:

```
error: C-style casts are discouraged; use static_cast
[google-readability-casting,-warnings-as-errors]
```

Это подтверждает работоспособность внедренного контроля качества. Настроенная инфраструктура гарантирует, что любой код, попадающий в ветку `main`, не только проходит функциональные тесты, но и соответствует строгим критериям статического анализа, что минимизирует технический долг проекта.

The screenshot shows a CI dashboard with a summary and a detailed build log.

Summary:

- ubuntu-latest - Release** (Failed)
- windows-latest - Release** (Success)
- macos-latest - Release** (Success)

Build Log:

```

ubuntu-latest - Release
failed 1 hour ago in 39s
14s
Search logs Explain error ⚙️ ⚙️

Build
▼ ✘ Run cmake --build build --config Release
  1 ► Run cmake --build build --config Release
  2 [ 5%] Building CXX object CMakeFiles/CoreEngine.dir/src/Payoff.cpp.o
  3 [ 10%] Building CXX object CMakeFiles/CoreEngine.dir/src/Analytical.cpp.o
  4 [ 15%] Building CXX object CMakeFiles/CoreEngine.dir/src/NCEngine.cpp.o
  5 [ 20%] Linking CXX static library libCoreEngine.a
  6 [ 20%] Built target CoreEngine
  7 [ 25%] Building CXX object CMakeFiles/MonteCarloApp.dir/main.cpp.o
  8 [ 25%] Built target MonteCarloApp
  9 [ 25%] Building CXX object CMakeFiles/ParallelIMCOptionPricing/ParallelIMCOptionPricing/main.cpp.o
10 /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/main.cpp:39:18: error: C-style casts are discouraged; use static_cast [google-readability-casting,
  warnings-as-errors]
11   39 |     int pi_int = (int)pi;
12   |             ^~~~~
13   |             static_cast<int>()
14 /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy:22:1: error: unknown key '...'
15 ...
16 ^~~
17 Error parsing /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy: Invalid argument
18 /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy:22:1: error: unknown key '...'
19 ...
20 ^~~
21 Error parsing /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy: Invalid argument
22 /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy:22:1: error: unknown key '...'
23 ...
24 ^~~
25 Error parsing /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy: Invalid argument
26 /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy:22:1: error: unknown key '...'
27 ...
28 ^~~
29 Error parsing /home/runner/work/ParallelIMCOptionPricing/ParallelIMCOptionPricing/.clang-tidy: Invalid argument
30 199 warnings generated.
31 Suppressed 198 warnings (198 in non-user code).
32 Use -header-filter=.* to display errors from all non-system headers. Use -system-headers to display errors from system headers as well.
33 1 warning treated as error
34 make[2]: *** [CMakeFiles/MonteCarloApp.dir/build.make:79: CMakeFiles/MonteCarloApp.dir/main.cpp.o] Error 1
35 make[1]: *** [CMakeFiles/Makefile2:197: CMakeFiles/MonteCarloApp.dir/all] Error 2
36 make: *** [Makefile:146: all] Error 2
37 Error: Process completed with exit code 2.

```

Run Tests 0s

Рис. 3: Блокировка сборки системой CI при обнаружении нарушения стиля кода.

7 Анализ производительности

Одной из ключевых целей работы являлась демонстрация эффективности параллельных вычислений. В данном разделе приводится методика тестирования, полученные результаты и их теоретическое обоснование.

7.1 Методика тестирования

Тестирование производилось на аппаратной платформе с процессором, имеющим 6 физических ядер и поддерживающим технологию Hyper-Threading (12 логических потоков).

Параметры бенчмарка:

- **Инструмент:** Европейский Call-опцион.
- **Число симуляций (N):** 10^7 (10 миллионов путей).
- **Параметры рынка:** $S_0 = 100$, $K = 100$, $T = 1.0$, $r = 5\%$, $\sigma = 20\%$.
- **Метрика:** Ускорение (Speedup) $S_p = \frac{T_1}{T_p}$, где T_1 — время выполнения на 1 потоке, T_p — на p потоках.

7.2 Результаты измерений

В ходе эксперимента фиксировалось время выполнения расчета при изменении числа потоков от 1 до 12. Результаты представлены в таблице ниже и на графике (Рис. 4).

Таблица 1: Зависимость времени выполнения и ускорения от числа потоков

Потоки	Время (сек)	Ускорение (x)	Эффективность (%)
1	1.3654	1.00	100%
2	0.7582	1.80	90%
4	0.3828	3.57	89%
6	0.2673	5.11	85%
8	0.2151	6.35	79%
10	0.1754	7.79	78%
12	0.1557	8.77	73%

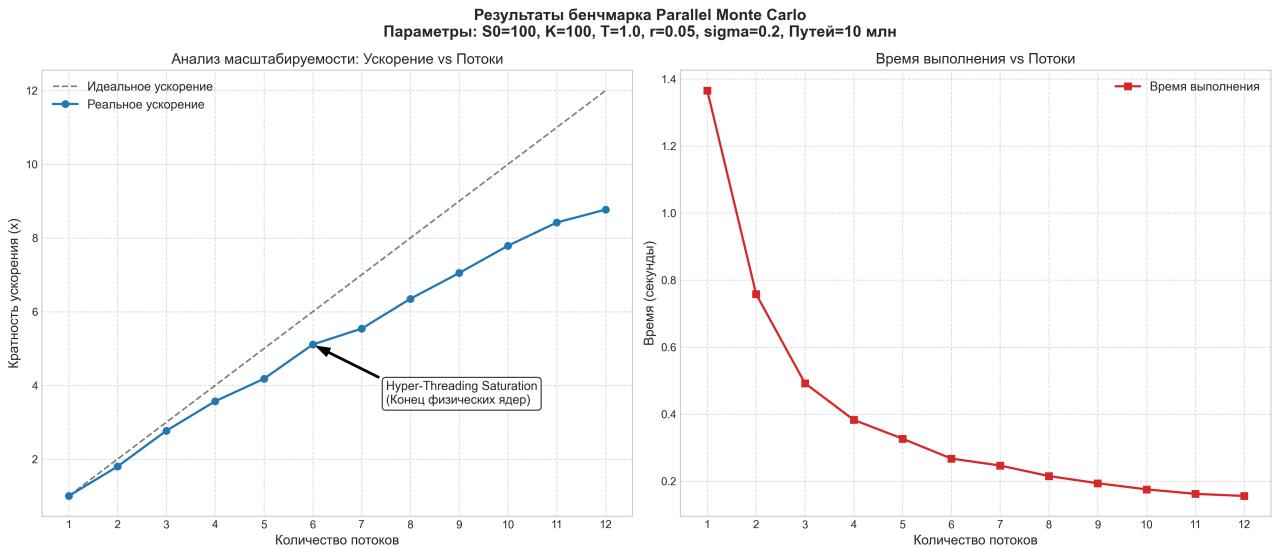


Рис. 4: График зависимости ускорения от количества потоков.

7.3 Анализ масштабируемости

На графике можно выделить две характерные фазы роста производительности, которые объясняются архитектурой процессора.

Фаза 1: Линейный рост (1–6 потоков) В диапазоне от 1 до 6 потоков наблюдается почти линейное ускорение.

- Задача Монте-Карло относится к классу *Embarrassingly Parallel*: вычисления траекторий полностью независимы друг от друга.
- Отсутствие разделяемых данных (Shared State) между потоками сводит к минимуму накладные расходы на синхронизацию.
- Каждому программному потоку соответствует отдельное физическое ядро, обладающее собственным блоком вычислений с плавающей точкой (FPU) и L1/L2 кэшем.

Фаза 2: Насыщение Hyper-Threading (7–12 потоков) При использовании более 6 потоков график отклоняется от идеальной прямой (пунктирная линия).

- Технология Hyper-Threading позволяет выполнять два потока на одном физическом ядре, однако они делят общие исполнительные ресурсы (ALU, FPU).
- Поскольку генерация случайных чисел и вычисление экспонент (exp) и интенсивно нагружают FPU, два потока на одном ядре начинают конкурировать за ресурс, что снижает эффективность масштабирования.
- Тем не менее, итоговое ускорение **8.77x** значительно превышает возможности физических ядер (6x), что доказывает эффективность утилизации всех логических ядер.

7.4 Архитектурные оптимизации и работа с памятью

Высокая производительность движка достигается не только за счет использования всех ядер процессора, но и благодаря низкоуровневым оптимизациям работы с кэшем и планировщиком операционной системы.

Использование локальных переменных для аккумуляции сумм (см. п. 4.4) также позволяет избежать эффекта False Sharing, так как потоки не конкурируют за одну кэш-линию L1.

8 Заключение

В ходе выполнения проекта была спроектирована и реализована библиотека для оценки стоимости опционов методом Монте-Карло.

8.1 Основные результаты

1. **Архитектура:** Разработана гибкая объектно-ориентированная архитектура с использованием паттерна *Стратегия*, позволяющая легко добавлять новые типы контрактов без модификации ядра вычислителя.
2. **Производительность:** Реализован параллельный алгоритм на базе `std::async`, обеспечивающий ускорение в 8.77 раза на 12-поточном процессоре. Время расчета 10 миллионов сценариев составляет всего 0.15 секунды.
3. **Точность:** Внедрен метод антитетических переменных, повышающий точность оценки. Корректность расчетов подтверждена сравнением с аналитической формулой Блэка-Шоулза.
4. **Качество кода:** Настроена инфраструктура CI/CD, статический анализ и модульное тестирование, что соответствует стандартам индустриальной разработки.

8.2 Направления дальнейшего развития

В качестве дальнейших улучшений системы можно выделить:

- Реализацию квазислучайных последовательностей (Соболя) для дальнейшего ускорения сходимости.
- Портирование вычислительного ядра на GPU с использованием технологии CUDA для достижения массового параллелизма.
- Добавление поддержки американских опционов методом Лонгстаффа-Шварца.

Список литературы

- [1] John C. Hull. *Options, Futures, and Other Derivatives*. Pearson, 10th Edition, 2017.
- [2] Mark S. Joshi. *C++ Design Patterns and Derivatives Pricing*. Cambridge University Press, 2nd Edition, 2008.
- [3] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.
- [4] Anthony Williams. *C++ Concurrency in Action*. Manning Publications, 2nd Edition, 2019.
- [5] Fischer Black, Myron Scholes. *The Pricing of Options and Corporate Liabilities*. Journal of Political Economy, 1973.