УДК 004.432+519.673

ОПТИМИЗАЦИЯ АЛГОРИТМОВ ЧИСЛЕННОГО МОДЕЛИРОВАНИЯ С++ С ИСПОЛЬЗОВАНИЕМ МЕТОДОВ МНОГОПОТОЧНОСТИ

Ю. С. Ефимов[0009-0003-6333-2109]

H&H company, г. Лиссабон, 1675-107, Португалия

yura2x2@gmail.com

Аннотация

Представлены основные методы численного моделирования (конечных разностей, конечных элементов, Монте-Карло, Рунге–Кутты). Рассмотрены основные параметры, используемые для оптимизации алгоритмов численного моделирования с точки зрения длительности выполнения кода и эффективного использования ресурсов процессора. Проанализированы основные недостатки многопоточности, связанные с синхронизацией данных, дедлоками и состояниями гонки и методы их устранения на основе применения мьютексов и атомарных операций на примере метода Монте-Карло.

Ключевые слова*: язык программирования C++, методы многопоточности, численное моделирование, синхронизация данных.*

ВВЕДЕНИЕ

Численное моделирование играет ключевую роль в современных научных исследованиях и технических разработках [1]. Оно позволяет анализировать сложные физические, биологические и экономические процессы с высокой точностью [2]. Однако вычислительная сложность многих численных методов требует эффективного использования аппаратных ресурсов. Одним из наиболее продуктивных подходов к ускорению вычислений является многопоточность в языке C++ [3], которая позволяет распределять задачи между несколькими потоками на многоядерных процессорах.

Многопоточность в C++ предоставляет мощные инструменты для параллелизации вычислительных алгоритмов. Используя библиотеки, такие как std::thread, и механизмы синхронизации, такие как мьютексы и условные пере-

менные, разработчики могут разделить задачу на независимые подзадачи и выполнять их параллельно. Это особенно эффективно для численных методов, где операции над большими массивами данных или сложные итерационные процессы могут быть разделены между несколькими ядрами процессора.

Применение многопоточности в численном моделировании требует тщательного анализа алгоритма и выбора оптимальной стратегии параллелизации. Важно учитывать накладные расходы на создание и синхронизацию потоков, а также потенциальные проблемы, связанные с гонками данных и взаимными блокировками. Эффективная реализация многопоточности позволяет существенно сократить время вычислений и повысить производительность численных моделей [4].

Примерами успешного применения многопоточности при численном моделировании являются расчеты в гидродинамике, моделирование климата, анализ данных в физике высоких энергий и оптимизация логистических процессов. В этих областях многопоточные алгоритмы позволяют проводить более масштабные и точные симуляции, что приводит к новым научным открытиям и технологическим инновациям.

Таким образом, многопоточность в C++ является важным инструментом для повышения эффективности численного моделирования. Правильное применение этой технологии позволяет значительно ускорить вычисления и расширить возможности анализа сложных процессов, открывая новые горизонты в научных исследованиях и инженерных разработках.

Целью настоящей работы является исследование возможностей многопоточного программирования в C++ для оптимизации алгоритмов численного моделирования. Рассмотрены основные принципы многопоточного программирования, методы балансировки нагрузки и синхронизации потоков, а также практические примеры реализации многопоточных алгоритмов.

ОСНОВЫ ЧИСЛЕННОГО МОДЕЛИРОВАНИЯ

Численное моделирование включает в себя использование математических методов для приближенного решения уравнений, описывающих сложные процессы. Оно применяется в таких областях, как аэродинамика, климатология, финансовый анализ, медицинская диагностика и др.

Наиболее популярные численные методы включают (см., например, [4]):

1. **Метод конечных разностей** – это численный способ решения дифференциальных уравнений, где производные аппроксимируются разностными отношениями. Он относится к классу сеточных методов.

2. **Численный метод конечных элементов** (МКЭ) представляет собой вычислительную стратегию для нахождения решений дифференциальных уравнений в частных производных и интегральных уравнений, которые часто возникают в различных областях прикладных математики и физики. Он широко используется для исследования деформаций в твердых телах, процессов теплопередачи, гидродинамических явлений, электромагнитных полей и задач оптимизации формы.

3. **Методы Монте-Карло** (ММК) – это совокупность вычислительных приемов, предназначенных для анализа случайных событий. Ключевой принцип заключается в имитации процесса с использованием генератора случайных чисел, многократном выполнении моделирования и последующем расчете вероятностных характеристик исследуемого процесса на основе полученных результатов. В качестве примера, для определения среднего расстояния между произвольными точками в круге создают случайные пары координат, вычисляют расстояние для каждой пары, а затем определяют среднее арифметическое значение.

4. **Методы Рунге–Кутты** – это обширная группа численных алгоритмов, предназначенных для решения задачи Коши для обыкновенных дифференциальных уравнений и их систем.

ОСНОВЫ МНОГОПОТОЧНОСТИ В C++

Многопоточность позволяет программе одновременно выполнять несколько задач, используя ресурсы многоядерных процессоров. В C++ многопоточное программирование реализуется с помощью библиотеки <thread>.

Пример простого многопоточного кода в C++:

```
#include <iostream>
#include <thread>
void task() {
   std::cout << "Поток выполняется" << std::endl;
}
```

```
int main() {
   std::thread t(task);  // Создание потока
   t.join();  // Ожидание завершения потока
   return 0;
}
```

Метод .join () гарантирует, что основной поток дождётся завершения порожденного потока.

При использовании многопоточности возникают сложности, связанные с:

1) **синхронизацией данных** – необходимость защиты общих ресурсов от одновременного изменения;

2) **состояниями гонки** (race conditions) – ситуациями, когда несколько потоков изменяют переменные одновременно;

3) **дедлоками** (deadlocks) – взаимоблокировками, приводящими к зависанию программы.

Для решения этих проблем используют механизмы синхронизации, такие как мьютексы (std::mutex) и атомарные операции (std::atomic).

Пример использования мьютекса:

```
#include <iostream>
#include <thread>
#include <mutex>
 std::mutex mtx;

void printData(int id) {
   std::lock_guard<std::mutex> lock(mtx);
   std::cout << "Поток " << id << " выполняется\n";
}

int main() {
   std::thread t1(printData, 1);
   std::thread t2(printData, 2);

   t1.join();
   t2.join();

   return 0;
}
```

Использование std::lock_guard предотвращает дедлоки, автоматически освобождая мьютекс при выходе из области видимости.

ОПТИМИЗАЦИЯ МНОГОПОТОЧНЫХ ВЫЧИСЛЕНИЙ

Правильное распределение вычислений между потоками позволяет избежать простаивания процессорных ядер и повысить эффективность выполнения программы. Использование динамического распределения задач и алгоритмов балансировки нагрузки играет важную роль в оптимизации.

Переключение между потоками требует значительных затрат ресурсов. Использование эффективных структур данных, уменьшение количества синхронизаций и минимизация блокировок позволяют снизить накладные расходы.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ МНОГОПОТОЧНОСТИ В ЧИСЛЕННОМ МОДЕЛИРОВАНИИ

В качестве типичного примера вычислительного моделирования можно привести метод Монте-Карло. Он широко применяется для решения задач интегрирования, проведения симуляций в физических исследованиях и построения моделей для анализа финансовых рынков.

В качестве иллюстрации рассмотрим многопоточную реализацию метода Монте-Карло для приближенного определения значения числа π:

```
#include <iostream>
#include <thread>
#include <vector>
#include <random>
#include <atomic>

const int num_threads = 4;
const int num_points = 1000000;
std::atomic<int> inside_circle(0);
void monteCarlo(int num_samples) {
   std::random_device rd;
   std::mt19937 gen(rd());
   std::uniform_real_distribution<double> dist(-1.0, 1.0);

   int count = 0;
   for (int i = 0; i < num_samples; i++) {
```

```
        double x = dist(gen);
        double y = dist(gen);
        if (x * x + y * y <= 1.0) count++;
    }
    inside_circle += count;
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < num_threads; i++) {
        threads.emplace_back(monteCarlo, num_points / num_threads);
    }
    for (auto& t : threads) {
        t.join();
    }
    double pi = 4.0 * inside_circle / num_points;
    std::cout << "Приближенное значение числа π: " << pi << std::endl;
    return 0;
}
```

В данном коде каждый поток независимо выполняет расчет, а переменная inside_circle обновляется атомарно для предотвращения состояния гонки.

Библиотека OpenMP позволяет упростить распараллеливание циклов без явного управления потоками.

Пример параллельного вычисления суммы массива:

```
#include <iostream>
#include <omp.h>
 int main() {
    const int N = 100000;
    int arr[N], sum = 0;
    for (int i = 0; i < N; i++) arr[i] = i;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++) {
        sum += arr[i];
    }
    std::cout << "Сумма элементов массива: " << sum << std::endl;
    return 0;
```

```
}
```

Использование #pragma omp parallel for автоматически распределяет итерации цикла между потоками, а reduction(+:sum) гарантирует корректное суммирование.

ПРОИЗВОДИТЕЛЬНОСТЬ И ТЕСТИРОВАНИЕ

Коэффициент ускорения многопоточной программы вычисляется по формуле

$$S = T_1/T_n,$$

где T_1 – время выполнения на одном потоке, а T_n – время выполнения на нескольких потоках.

Для измерения времени выполнения можно использовать std::chrono:

```
#include <iostream>
#include <chrono>

int main() {
   auto start = std::chrono::high_resolution_clock::now();

   // Ваш код

   auto end = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double> elapsed = end - start;
   std::cout << "Время выполнения: " << elapsed.count() << " секунд" << std::endl;
   return 0;
}
```

Необходимо равномерно распределять вычисления между потоками, чтобы избежать ситуаций, когда один поток загружен больше других. Для этого можно использовать динамическое распределение в OpenMP:

```
#pragma omp parallel for schedule(dynamic)
```

ГЛУБОКАЯ ОПТИМИЗАЦИЯ МНОГОПОТОЧНЫХ АЛГОРИТМОВ

При разработке многопоточных алгоритмов важно выбирать структуры данных, которые минимизируют блокировки потоков. Некоторые из наиболее эффективных подходов таковы:

1) **атомарные структуры данных** (std::atomic, std::atomic_flag) – используются для защиты переменных от одновременной записи. Они эффективнее мьютексов, так как работают без блокировки;

2) **параллельные контейнеры** (например, concurrent_queue в TBB) – обеспечивают потокобезопасное взаимодействие без необходимости вручную управлять мьютексами;

3) **чтение – копирование – обновление** (Read–Copy–Update, RCU) – техника, позволяющая избежать блокировок при работе с разделяемыми структурами.

Пример использования std::atomic в многопоточном счетчике:

```
#include <iostream>
#include <thread>
#include <atomic>
std::atomic<int> counter(0);
void increment(int iterations) {
   for (int i = 0; i < iterations; i++) {
      counter.fetch_add(1, std::memory_order_relaxed);
   }
}
int main() {
   const int num_threads = 4;
   const int iterations_per_thread = 1000000;
   std::thread threads[num_threads];
   for (int i = 0; i < num_threads; i++) {
      threads[i] = std::thread(increment, iterations_per_thread);
   }
   for (auto& t : threads) {
      t.join();
   }
   std::cout << "Итоговое значение счетчика: " << counter.load() << std::endl;
```

```
    return 0;
}
```

Этот метод позволяет избежать проблем состояния гонки, улучшая стабильность программы.

Производительность многопоточных программ может снижаться из-за ложного разделения – ситуации, когда разные потоки работают с разными переменными, но эти переменные находятся в одной кэш-линии процессора.

Методы устранения ложного разделения:

1) выравнивание данных по размеру кэш-линии (обычно 64 байта);

2) использование alignas(64) или std::hardware_destructive_interference_size;

3) разделение потоков по разным частям памяти.

Пример устранения ложного разделения с alignas(64):

```
#include <iostream>
#include <thread>
#include <vector>

const int num_threads = 4;
alignas(64) int data[num_threads] = {0};
void increment(int index) {
    for (int i = 0; i < 1000000; i++) {
        data[index]++;
    }
}
int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < num_threads; i++) {
        threads.emplace_back(increment, i);
    }
    for (auto& t : threads) {
        t.join();
    }
    for (int i = 0; i < num_threads; i++) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }
```

```
    return 0;
}
```

Вместо явного управления потоками с std::thread можно использовать std::async, который позволяет запускать задачи асинхронно с автоматическим управлением ресурсами.

Пример использования std::async:

```
#include <iostream>
#include <future>
 int slowCalculation() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 42;
}

int main() {
    std::future<int> result = std::async(std::launch::async, slowCalculation);
    std::cout << "Ждем результата..." << std::endl;
    std::cout << "Результат: " << result.get() << std::endl;
    return 0;
}
```

Применение `std::async` упрощает структуру программы по сравнению с использованием `std::thread` в сочетании с `std::mutex`, поскольку механизм синхронизации в этом случае становится прозрачным и автоматизированным.

ПРОИЗВОДИТЕЛЬНОСТЬ И ТЕСТИРОВАНИЕ МНОГОПОТОЧНЫХ АЛГОРИТМОВ

Для анализа производительности параллельных вычислений:

1) время выполнения кода оценивается с помощью библиотек `std::chrono` для точного измерения временных интервалов;

2) для оценки загрузки процессора применяются такие инструменты, как системные команды `top`, `htop` и специализированное приложение `perf` в операционных системах на базе Linux;

3) ключевыми инструментами профилирования являются `gprof` для Unix-подобных систем, `Intel VTune` и профайлер Visual Studio для оценки эффективности кода.

Пример измерения времени выполнения программы:

```
#include <iostream>
#include <chrono>
#include <thread>
void task() {
   std::this_thread::sleep_for(std::chrono::seconds(1));
}
int main() {
   auto start = std::chrono::high_resolution_clock::now();
    std::thread t(task);
   t.join();
    auto end = std::chrono::high_resolution_clock::now();
   std::chrono::duration<double> elapsed = end - start;
   std::cout << "Время выполнения: " << elapsed.count() << " секунд" << std::endl;
   return 0;
}
```

Коэффициент ускорения (speedup) рассчитывается по формуле

$$S = T_1/T_p,$$

где T_1 – время выполнения на одном потоке, а T_p – время выполнения на p потоках.

Коэффициент эффективности (efficiency) определяется как

$$E = S/p.$$

При идеальном масштабировании S приближается к p, а E – к 1.

ПЕРСПЕКТИВЫ РАЗВИТИЯ МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ В ЧИСЛЕННОМ МОДЕЛИРОВАНИИ

Рассмотрим использование CUDA (NVIDIA) и OpenCL для переноса вычислений на видеокарту.

Параллелизация численных методов на GPU дает прирост производительности в 10–100 раз.

Пример использования CUDA:

```
__global__ void add(int *a, int *b, int *c, int N) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (i < N) c[i] = a[i] + b[i];
}
```

Многие процессоры поддерживают инструкции SIMD, позволяющие выполнять несколько операций одновременно.

Пример компиляции с автопараллелизацией в GCC:

```
g++ -O2 -march=native -ftree-vectorize program.cpp -o program
```

ПЕРСПЕКТИВЫ РАЗВИТИЯ

1. Использование векторизации с SIMD-инструкциями для повышения эффективности вычислений.
2. Применение GPU-ускорения с помощью CUDA и OpenCL для численного моделирования.
3. Развитие асинхронного программирования с использованием std::async и future.

ЗАКЛЮЧЕНИЕ

Использование многопоточности в C++ позволяет значительно повысить производительность численного моделирования, снижая время выполнения ресурсоемких расчетов. Однако для эффективного применения многопоточных технологий важно учитывать балансировку нагрузки, минимизацию состояния гонки и снижение накладных расходов на управление потоками.

СПИСОК ЛИТЕРАТУРЫ

1. *Умнов А.Е.* Методы математического моделирования: учебное пособие. М.: МФТИ, 2013. 295 с.
2. *Марчевский И.К., Щерица О.В.* Численные методы решения задач математической физики: учебно-методическое пособие. М.: МГТУ им. Н.Э. Баумана, 2016. 64 с.
3. *Уильям Э.* Практика многопоточного программирования C++. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. Пер. с англ. Слинкин А.А. М.: ДМК Пресс, 2012. 672с.
4. *Зенков А.В.* Численные методы: учеб. пособие. Екатеринбург: Изд-во Урал. ун-та, 2016. 124 с.

OPTIMIZATION OF NUMERCIAL SIMULATION ALGORTIHMS C++ WITH MULTITHREADING METHODS

Yu. S. Efimov[0009-0003-6333-2109]

H&H company, Lisbon, 1675-156, Portugal

yura2x2@gmail.com

Abstract

The main methods of numerical simulation (finite difference method, finite element method, Monte Carlo method, Runge–Kutta method) are presented. The main parameters used to optimize numerical modeling algorithms in terms of code execution time and efficient use of processor resources are considered. The main disadvantages of multithreading related to data synchronization, deadlocks and race conditions and methods for eliminating them based on the use of mutexes and atomic operations using the Monte Carlo method as an example were analyzed.

Keywords*: programming language C++, multithreading methods, numerical simulation, data synchronization.*

REFERENCES

1. *Umnov A.E.* Metody matematicheskogo modelirovaniya: uchebnoe posobie. M.: MFTI, 2013. 295 s. (in Russian).

2. *Marchevskij I.K., Shcherica O.V.* Chislennye metody resheniya zadach matematicheskoj fiziki: uchebno-metodicheskoe posobie. M.: MGTU im. N.E. Baumana, 2016. 64 s. (in Russian).

3. *Uil'yam E.* Praktika mnogopotochnogo programmirovaniya S++. Parallel'noe programmirovanie na C++ v dejstvii. Praktika razrabotki mnogopotochnyh programm. Per. s angl. Slinkin A.A. M.: DMK Press, 2012. 672s. (in Russian).

4. *Zenkov A.V.* Chislennye metody: ucheb. posobie. Ekaterinburg: Izd-vo Ural. un-ta, 2016. 124 s. (in Russian).

СВЕДЕНИЯ ОБ АВТОРЕ

ЕФИМОВ Юрий Сергеевич – 1979 года рождения, учился в Санкт-Петербургском государственном университете аэрокосмического приборостроения по специальности 2202 автоматизированные системы обработки информации и управления. В настоящее время работает программистом C++ в компании H&H. Область научных интересов: параллельные и распределённые вычисления, системы с низкой задержкой, оптимизация производительности программного обеспечения.

Yuri Sergeevich EFIMOV – born in 1979, studied at Saint Petersburg State University of Aerospace Instrumentation, specializing in Automated Information Processing and Control Systems (specialty code 2202). He currently works as a C++ programmer at H&H. Research interests: parallel and distributed computing, low-latency systems, software performance optimization.

email: yura2x2@gmail.com

ORCID: 0009-0003-6333-2109

Материал поступил в редакцию 2 апреля 2025 года