Оглавление

1. Введение

Требуется рассчитать численное значение числа π с заданным заданной точностью (в виде числа цифр после запятой). Для решения задачи применим составную квадратурную формулу левых прямоугольников для расчета определенного интеграла, аналитическое значение которого равно π, и многопоточное программирование для ускорения вычислений.

2. Подробное описание задачи

Для расчета числа π возьмем определенный интеграл, значение которого будет равно π.

$$\int_0^1 \frac{4}{1+x^2}\,dx = 4 * \int_0^1 \frac{1}{1+x^2}\,dx = 4\arctan(x)\,|_0^1 = 4\arctan(1) - 4\arctan(0) = \pi.$$

Для численных расчетов воспользуемся формулой левых прямоугольников, для чего разобьем исходный промежуток $[a, b]$ на N равных промежутков $[x_k, x_{k+1}]$

$$h = \frac{b-a}{N}, x_k = x_0 + kh, x_0 = a, x_{N=b}.$$

На каждом промежутке применим формулу левых прямоугольников и сложим результаты:

$$I_k = \int_{x_k}^{x_{k+1}} f(x)dx \approx (x_{k+1} - x_k)f(x_k) = \frac{b-a}{N}\, f(x_k);$$

$$I_{\text{лев.пр.}} = \sum_{k=0}^{N-1} I_k \approx \frac{b-a}{N} \sum_{k=0}^{N-1} f(x_k).$$

3. Описание базового алгоритма

Для расчета значений с заданной погрешностью будем сравнивать результаты, полученные по составной формуле для *N* и *2N*. В случае неудовлетворительного совпадения значение *N* удваивается до тех пор, пока требуемая точность не будет достигнута.

Блок-схема описанного выше алгоритма приведена на рис. 1.

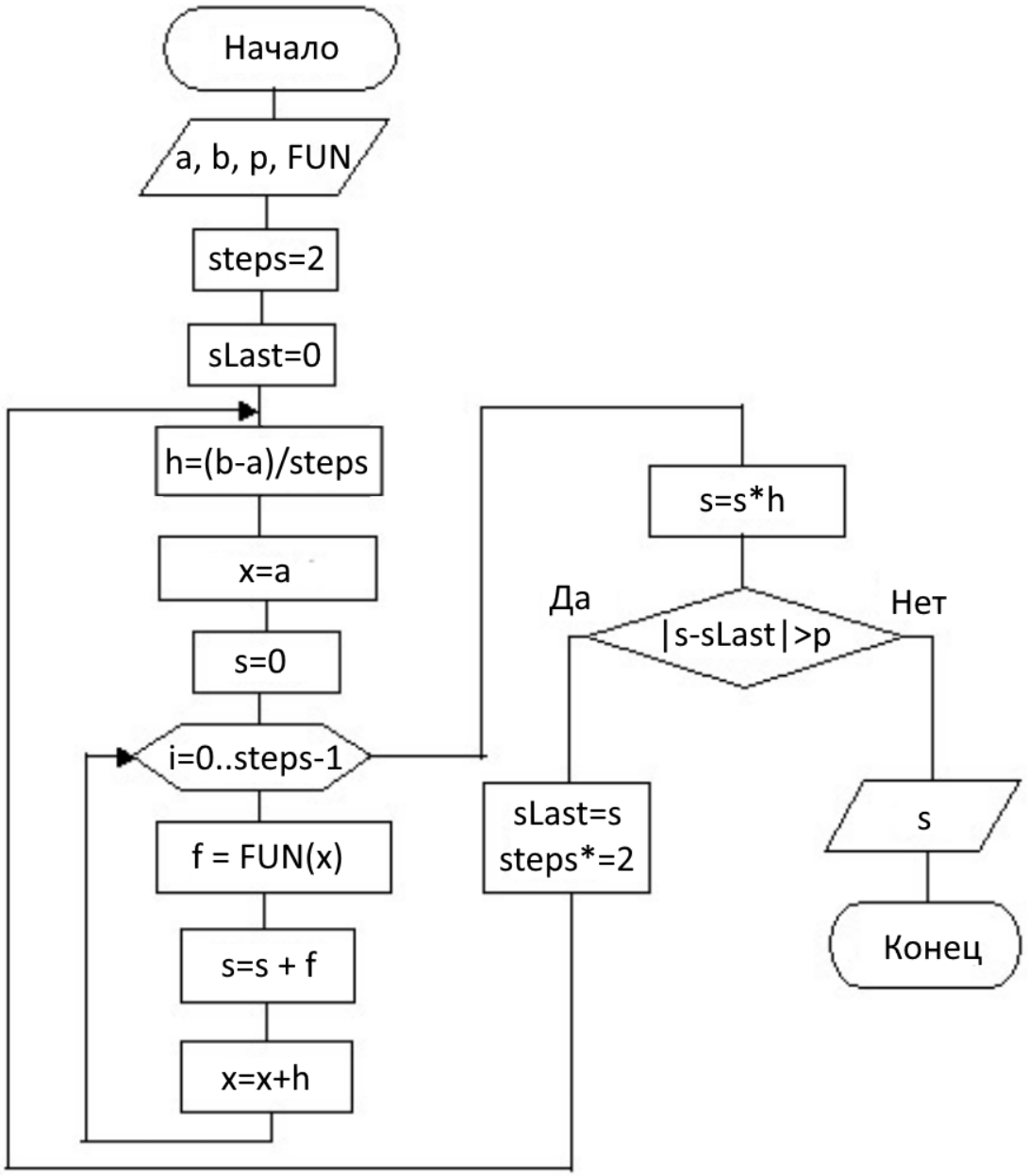

Рис. 1. Блок-схема алгоритма

4. Описание основных процедур

Реализуем однопоточную версию алгоритма на языке Java в виде метода *countIntegral(a, b, precision, function, round).* Метод приведен в листинге 1 класса *NumericalCount*.

На вход метода мы принимаем начало *a* и конец *b* промежутка интегрирования, точность расчетов (количество разрядов после запятой) и переменную типа функционального интерфейса *Function*, у которого есть единственный метод *apply*, с помощью этого интерфейса мы будем передавать подынтегральную функцию, значение которой в точке *x* возвращает метод *apply*. Параметр *round* – переменная булева типа указывает, в каком виде мы хотим получить результат: округленном или нет. Он понадобится со значением *true* для проведения тестирования (сравнения с эталонным

результатом, округленным до того же разряда) и со значением *false*, когда мы будем вызывать метод *countIntegral* из множества потоков, чтобы не накапливать ошибку округления.

В случае с многопоточной реализацией мы будем делить исходный промежуток на более маленькие, создавать пул потоков и передавать множество созданных ранее более маленьких промежутков потокам из пула, внутри самих потоков будем считать заданный интеграл на маленьком промежутке с точностью на один порядок выше. Для этого дополнительно будем принимать параметры, такие как *segments* – количество отрезков, на которые делится исходный промежуток и *threadAmount* – количество потоков, создаваемое многопоточной реализацией. Для управления потоками воспользуемся специальным API *ExecutorService*. Потоки создаются вызовом *Executors.newFixedThreadPool(threadAmount),* после чего в цикле потокам задаются задачи на исполнение с помощью вызова метода *executor.submit(),* в который передается лямбда функция для исполнения – вызов метода *countIntegral*, который рассчитывает результат на *i*-ом промежутке из *segments*. Когда всем потокам будут даны задания на своих промежутках, главный поток перейдет в режим ожидания после вызова метода *future.get()*. Этот метод вызывается циклически *segments* раз, в итоге в переменной *results* будет сформирован ответ, когда будут просуммированы результаты со всех *segments* промежутков, возвращенные потоками после вызова future.get(). Метод *countIntegralParallel*, реализующий описанную функциональность приведен в листинге 1 класса *NumericalCount*.

Методы *countIntegral* и *countIntegralParallel* объединим в класс *NumericalCount*.

5. Результаты работы приложения. Сравнение работы многопоточной и однопоточной реализаций

Для более объективной оценки результатов работы приложения будем сравнивать их со значением Math.PI из библиотеки Java, округленным до той же точности, что и результат наших методов. Для реализации этой задачи воспользуемся фреймворком JUnit и напишем параметризируемые тесты для наших методов. Класс *NumericalCountTest* приведен в листинге 2. Результаты прохождения тестов приведены на рис. 2.

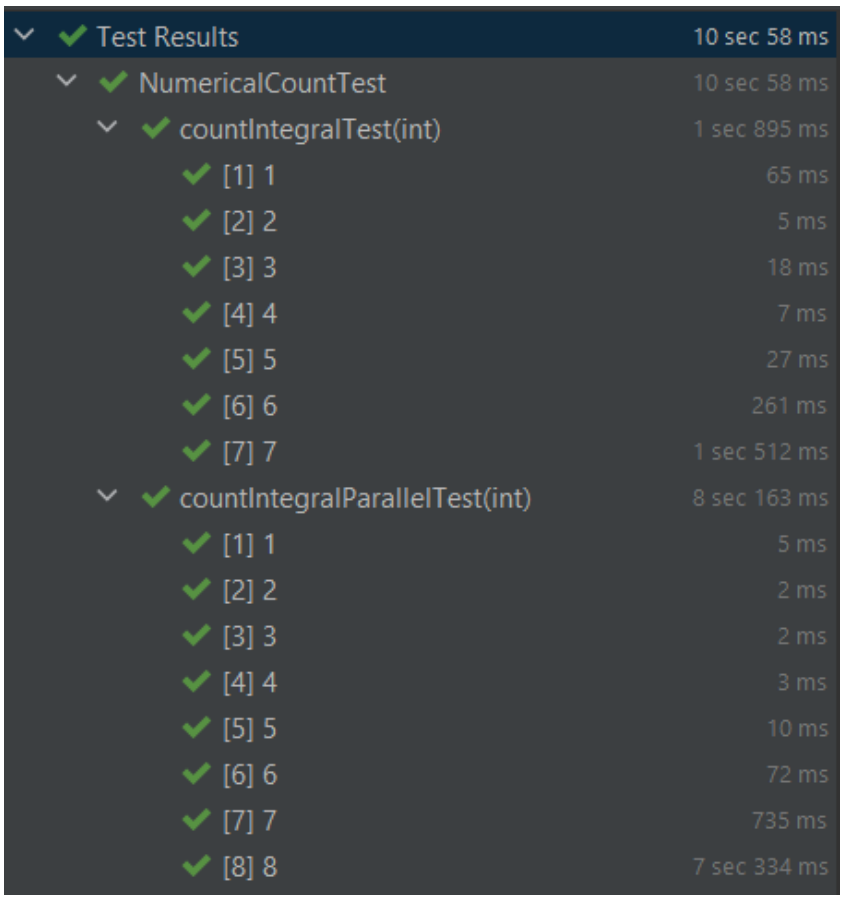

Рис. 2. Результаты тестирования.

Тестирование было проведено вплоть до 7 разряда (для многопоточного метода до 8) после запятой, дальнейшее тестирование вызывает затруднения, так как требует гораздо больше времени (количество операций необходимых для более точных расчетов растет пропорционально 2^N в лучшем случае (метод *countIntegral* увеличивает *steps* в 2 раза на каждой итерации)).

6. Сравнение временных характеристик

Для сравнения временных характеристик воспользуемся фреймворком для бенчмарка Java Microbenchmark Harness (JMH). Для определения количества потоков обратимся к среде выполнения с помощью метода *Runtime.getRuntime().availableProcessors()*. Класс *RealisationsBenchmark* приведен в листинге 3.

```
Thread amount to create: 8
```

Рис. 3. Значение, возвращаемое *Runtime.getRuntime().availableProcessors()*

Проведем временную оценку различных реализаций методов, результаты приведены в табл. 1.

Табл. 1. Результаты бенчмарка. Пометка (B) после метода означает, что исходный промежуток делился на число вдвое большее, чем количество потоков, в остальных случаях

отрезок делится на количество создаваемых потоков. Измерения приведены в микросекундах на операцию (вызов метода).

Method	(precision)	Score		Error	Units
countIntegralParallel	1	676.555	±	54.562	мкс/операцию
	2	686.329	±	108.222	мкс/операцию
	3	733.448	±	331.687	мкс/операцию
	5	6726.287	±	2228.835	мкс/операцию
	7	724886.391	±	141165.963	мкс/операцию
countIntegralParallel (B)	1	661.929	±	216.624	мкс/операцию
	2	714.439	±	85.015	мкс/операцию
	3	764.515	±	45.262	мкс/операцию
	5	3620.565	±	866.249	мкс/операцию
	7	322928.207	±	53136.573	мкс/операцию
countIntegral (sequential)	1	1.740	±	0.315	мкс/операцию
	2	20.741	±	0.950	мкс/операцию
	3	197.122	±	2.629	мкс/операцию
	5	9233.452	±	1432.857	мкс/операцию
	7	1223239.231	±	43803.060	мкс/операцию

Как мы видим из таблицы разделение промежутка на число маленьких промежутков значительно улучшило временные показатели работы программы. Это происходит из-за того, что в случае деления промежутка поровну между всеми потоками, если один из потоков посчитает свою часть раньше — то соответствующее ядро будет простаивать, т. е. мы потеряем производительность. Поэтому вариант разделения интервала на множество более маленьких участков и их раздача потокам по мере того, как они справляются со своей работой, является более быстродейственным.

Заметна значительная разница при небольшом объеме вычислений (при *precision* = 1, 2, 3), последовательная реализация работает быстрее, параллельная же дает значительный выигрыш лишь при больших объемах вычислений (когда создание и управление потоками становится оправданным).

7. Определение оптимального числа создаваемых потоков

Проверим, что значение, возвращаемое *Runtime.getRuntime().availableProcessors()* является оптимальным. Для этого измерим время исполнения метода *countIntegralParallel* с различным числом создаваемых потоков, значение segments, как мы уже выяснили, следует брать значительно большее, чем число потоков, точность вычислений зададим до 5 разряда. Класс *ThreadsBenchmark* приведен в листинге 4.

Табл. 2. Определение оптимального числа потоков

Method	(threads)	Score		Error	Units
countIntegralParallel	2	27113.33	±	8114.409	мкс/операцию
	3	15991.25	±	453.984	мкс/операцию
	4	9540.643	±	1483.266	мкс/операцию

	5	5794.597	±	1504.246	мкс/операцию
	6	9924.868	±	2775.222	мкс/операцию
	7	4452.742	±	2407.820	мкс/операцию
	8	5619.391	±	581.747	мкс/операцию
	9	4464.218	±	1001.070	мкс/операцию
	10	5392.384	±	1008.006	мкс/операцию
	11	3709.333	±	1267.950	мкс/операцию
	12	3741.088	±	628.991	мкс/операцию

Как можно заметить вплоть до 8 потоков растет быстродействие, после чего время выполнения метода значительно не меняется и в среднем составляет порядка 4000 микросекунд.

8. Вывод

В ходе данной работы было разработано и исследовано простое многопоточное приложение, реализующее численное интегрирование на примере расчета числа π.

Было проведено тестирование, а также оценено время исполнения различных реализаций.

Применение многопоточной реализации не всегда дает выигрыш в быстродействии, а только при большом объеме вычислений.

Было экспериментально определено, что оптимальное количество создаваемых потоков для многопоточного приложения равно числу доступных ядер, а также оптимальный метод распараллеливания.

9. Список используемой литературы

1. Устинов С. М., Зимницкий В.А. Вычислительная математика – СПб.: БВХ-Петербург, 2009. – 336 с. – (Учебное пособие.)
2. Документация по JMH
3. Документация по JUnit

10. Приложение. Листинг программной реализации

Исходный код проекта доступен в репозитории по ссылке:
https://github.com/mishkowsky/NumericalCountJMH

Листинг 1. Класс *NumericalCount.*

```
public class NumericalCount {

    public static double countIntegral(
            double a, double b, int precision, Function<Double, Double>
function, boolean round) {
        long steps = 2;
        double s = 0;
        double sLast;
        do {
            steps *= 2;
            sLast = s;
            double h = (b - a) / steps; // h = (b - a) / pow(2,n);
```

```
            double x = a;
            for (int i = 0; i < steps; i++) {
                s += function.apply(x); // s = s + f(x);
                x += h; // x = x + h;
            }
            s *= h; // s = s*h;
        }
        // чтобы корректно округлить до precision цифры после запятой, нужно
знать precision+1 цифру после запятой
        while (abs(s - sLast) >= pow(10, -precision - 1));
        if (round) return round(s, precision);
        else return s;
    }

    public static double countIntegralParallel(
            double a, double b, int precision, int segments,
            Function<Double, Double> piIntegral, int threadAmount) {
        double result = 0;
        double h = (b - a) / segments;
        double ai = a;
        double bi = a + h;
        List<Future<Double>> futures = new ArrayList<>();
        ExecutorService executor =
Executors.newFixedThreadPool(threadAmount);
        for (int i = 0; i < segments; i++) {
            double aConst = ai;
            double bConst = bi;
            futures.add(executor.submit(() ->
                    countIntegral(aConst, bConst, precision + 1, piIntegral,
false)));
            ai += h;
            bi += h;
        }
        for (Future<Double> future : futures) {
            try {
                result += future.get();
            } catch (InterruptedException | ExecutionException e) {
                System.out.println("ERROR!");
                e.printStackTrace();
            }
        }
        executor.shutdown();
        return round(result, precision);
    }
}
```

Листинг 2. Класс *NumericalCountTest*

```
public class NumericalCountTest {

    private static final int threads =
Runtime.getRuntime().availableProcessors();

    private final double a = 0;
    private final double b = 1;
    private final Function<Double, Double> piIntegral = (x) -> 4 / (1 + x *
x);

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5, 6, 7})
    public void countIntegralTest(int precision) {
        assertEquals(round(PI, precision),
                countIntegral(a, b, precision, piIntegral, true));
    }
```

```
    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 4, 5, 6, 7, 8})
    public void countIntegralParallelTest(int precision) {
        assertEquals(round(PI, precision),
                countIntegralParallel(a, b, precision, threads, piIntegral,
threads));
    }
}
```

Листинг 3. Класс RealisationsBenchmark

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Benchmark)
@Fork(value = 1, jvmArgs = {"-Xms2G", "-Xmx2G"})
@Warmup(iterations = 3)
@Measurement(iterations = 5)
public class RealisationsBenchmark {

    private static final int threadAmount =
Runtime.getRuntime().availableProcessors();

    @Param({"1", "2", "3", "5", "7"})
    private int precision = 1;

    private final double a = 0d;
    private final double b = 1d;

    private final Function<Double, Double> piIntegral = (x) -> 4 / (1 + x *
x);

    public static void main(String[] args) throws RunnerException {
        System.out.println("Thread amount to create: " + threadAmount);
        Options opt = new OptionsBuilder()
                .include(RealisationsBenchmark.class.getSimpleName())
                .build();
        new Runner(opt).run();
    }

    @Benchmark
    public void sequentialDouble(Blackhole bh) {
        double result = countIntegral(a, b, precision, piIntegral, true);
        bh.consume(result);
    }

    @Benchmark
    public void parallelDouble(Blackhole bh) {
        double result = countIntegralParallel(
                a, b, precision, threadAmount, piIntegral, threadAmount);
        bh.consume(result);
    }

    @Benchmark
    public void parallelDoubleB(Blackhole bh) {
        double result = countIntegralParallel(
                a, b, precision, 2 * threadAmount, piIntegral, threadAmount);
        bh.consume(result);
    }
}
```

Листинг 4. Класс ThreadsBenchmark

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@State(Scope.Benchmark)
@Fork(value = 1, jvmArgs = {"-Xms2G", "-Xmx2G"})
```

```java
@Warmup(iterations = 3)
@Measurement(iterations = 5)
public class ThreadsBenchmark {

    @Param({"2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12"})
    private int threads = 2;

    private final Function<Double, Double> piIntegral = (x) -> 4 / (1 + x *
x);

    @Benchmark
    public void threads(Blackhole bh) {
        int precision = 5;
        double a = 0d;
        double b = 1d;
        double result = countIntegralParallel(
                a, b, precision, threads, piIntegral, threads);
        bh.consume(result);
    }
}
```