Taylor Approximation with Threads

Esra Kantarcı - 20160808023

May 2020

1 Introduction to Task

The π number is very important and widely used constant in math and physics which is actually the ratio of the circumference of a circle to its diameter. The exact number of the all digits are said to be going to infinity (but not in a repetitive way.) People tried to calculate the more precise value of the π, since approximation is good but not as good as being 100% percent sure!

Around 1650 BC, an Ancient Egyptian wrote in the Rhind Papyrus:"Cut off 1/9 of a diameter and construct a square upon the remainder; this has the same area as the circle" which means π equals 3,16049 approximately using the calculation on $(8/9)^2 * 4$.

In 250 BC, Archimedes used polygons (up to 96-sided-polygons) to approximate the value of π, which concludes pi is between 3.1408 and 3.1428. The calculations and methods regarding this issue started 3000 years ago and now we can calculate up to 70.000 digits thanks to supercomputers.But as you see there were many people who tried to get accurate and precise digits of π and in this task, we need to use one of these methods: Taylor Series Approximation.

What is Taylor Series? First, we need to understand Taylor's theorem and Taylor polynomial. The name of these methods comes from the English mathematician Brook Taylor who lived in 1700s. The idea of the polynomials is imitating the behaviors of the irregular or complicated functions and therefore making a ground for approximations.

So, the first step for the approximation is finding an accurate polynomial. The one of the most important issue we should consider is that the polynomial should be continuous and derivable, and it should also be convergent when it goes to infinity. For example this series was described by Isaac Newton:

$$|x| < 1$$

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4...$$

So, the absolute value of x will still be less than 1, even if we change variable x to ($-x^2$). But now, what we have is:

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8...$$

Which is an alternating series example. If we integrate this with respect to x, since it has continuous features, we will get arctanx at the left hand side and something like that in the right hand side:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + ...$$

As you can see, the new series are alternating sum series with same denominator with the power of the respective x in the ratio. The power is always an odd number. The odd numbers can be shown as 2i+1 or 2i-1. We will use this approach on our code implementation.

Now, let's get to the point. What is the point with all of these series and π? Easy. What is tangent of $\pi/4$? It equals 1. Arc-tangent is the reverse operation of the tangent equation, so arctan1 equals $\pi/4$.

So, if we write 1 instead of x in the equation, we will get:

$$\arctan 1 \cong 1 - \frac{1^3}{3} + \frac{1^5}{5} - \frac{1^7}{7} + ... = \pi/4$$

Therefore, if we add more and more add/sub operations into this series, we will be getting closer and closer to infinity, which brings us the more precise equivalent values of the π.

This method of the π approximation is what we are going to code in this particular assignment.

2 Requirements

The assignment says that we should use Java or C language to code and provide a report for it. There are 8 requirements to fulfil within the assignment:

1- Multi-threaded code

2- Approximation of the π using Taylor series

3- N operations (N/2 subtractions, N/2 additions) which N will be taken from the user

4- Approximation error should be displayed as the report

5- Computation time should be displayed as the report

6- Input for the number of computation-threads should be taken from user

7- The maximum number of threads should be 64

8- Number of the threads should be increased by 1 in the every iteration

In order to do that, I made the necessary literature and code researches. I found a good and reusable solution at code5.cn website.The base thread code which was used there is very nice and clean. It provides all the necessary features for the multi-thread support(1), Taylor series implementation and iterations(2). However, it calculates only $\pi/4$, so I just used a multiplication inside the thread.

So, after that, I focused on the other tasks. First of all, display of the report of the approximation error and the result (4) was done. The approximation error is actually the difference between real π value and the calculated π. So, using the Math.PI is a good way to represent the real π.

But, it should be noted that the smaller number of the iterations ends up with bad approximations and the number of threads does not change too much since there are no precise-loss operations:

```
How many threads should work concurrently? Enter an integer(Between 1 and 64):
10
The thread count is 10 and number of iterations is 10
This means there will be 5 add and sub operations each.

The Real PI is            3.141592653589793
Calculated PI is          3.232315809405594

Time spent: 1 ms
The approximation error: -0.09072315581580082
```

Figure 1: 10 operations with 10 threads

However if we change the operations to a relatively bigger number, the approximation error gets smaller and more threads can bring difference.

```
The thread count is 2 and number of iterations is 1000000
This means there will be 500000 add and sub operations each.

The Real PI is            3.141592653589793
Calculated PI is          3.141593653588707

Time spent: 39 ms
The approximation error: -9.999989138975707E-7
```

Figure 2: 1.000.000 operations with 2 threads

```
The thread count is 64 and number of iterations is 1000000
This means there will be 500000 add and sub operations each.

The Real PI is            3.141592653589793
Calculated PI is          3.1415936535888185

Time spent: 42 ms
The approximation error: -9.999990253639623E-7
```

Figure 3: 1.000.000 operations with 64 threads

So whenever the number of iterations get higher, the difference goes lower. And also, this way the threads' work can be divided and done concurrently, and the calculations will be more precise since the loss will be smaller. But the

joining the threads and summation of their results take more time, and therefore time spent will be increasing by a little.

After this step, I added computation-time for the each execution cycle(5). I used nanoTime() function to get the start and end time of the thread-executions for the more precise results, then converted them to milliseconds for the readability purposes.

The input and dynamic changes were the my next step(3)(6). The code was suitable for the variables, therefore I just added scanning input features and some restrictions on the inputs. Regarding this part, I had encountered a bug of the Scanner library. You cannot actually clear the buffer of the scanner, but you can use nextLine() method to check the next inputs. However, it did not work very well with my current architecture (after I added do-while loop). So, I made a workaround, by creating 3 different scanners for 3 different variables. This should have been done more elegantly, by using maybe different variables or assignments, but I liked this approach because it worked quite efficiently.

The maximum thread number is ranged between 1 and 64. So, whenever user enters an integer which is off-limits, I set the default thread number to maximum(7). Also, if the user enters non-integer input, the thread number will be randomly chosen by using Math.random().

Since the last task was increasing the number of threads by one in every iteration(8), I added a do-while loop. This loop breaks whenever the thread number is higher than 64 or the user is not willing to continue. I needed to add hasNext and operationSet booleans and separate Scanners in order to prevent the problems with user's meaningless inputs(like using "n" for the numerical inputs, which affects the answer afterwards).

```
The thread count is 64 and number of iterations is 5000000
This means there will be 2500000 add and sub operations each.

The Real PI is            3.141592653589793
Calculated PI is          3.1415928535898257

Time spent: 202 ms
The approximation error: -2.0000003253528575E-7
```

Figure 4: 5.000.000 operations with 64 threads

And after some fine-tunings with formatting the outputs, the code fulfills all the requirements.

3 Conclusion

The threads and concurrent calculations are very handy if the problem can be divided into sub-parts which are not dependent to each other's result. So, atomic operations with the known-limits show great results with multi-threading methods.

We see that whenever the number of the operations is chosen higher, the results are getting better and better with the threads. However if the number of operations is low, the number of threads does not really change the game much.

I believe the results may have been better and faster if C language was used. With new technology we can calculate more and more precise π digits. But we will not be able to calculate the infinity, since the infinity cannot be reached with materialistic methods. The best we can is the approximation or the assumptions.

4 The Code without Commentaries

```
import java.util.Scanner;
import java.util.concurrent.TimeUnit;
public class TaylorApprox {

    public static void main(String[] args) throws InterruptedException {

    boolean hasNext=true;
    boolean operationSet=false;
    int N = 1_000_000;
    String answer="N";
    String div= "\n--------------------------";
    Scanner scan= new Scanner(System.in);

    if(!operationSet){
        System.out.print("Hello World!\n\nHow many operations should "
                + "we have in the calculation? \nTry a high number, because "
                + "more operations come with the better approximation!"
                + "\nEnter an integer(Between 1 and 5.000.000): \n");

        if(scan.hasNextInt()){
            N = scan.nextInt();

             if(N>5_000_000 || N<1){
                System.out.println("This is offlimits. "
                        + "I will calculate from 1.000.000, which is "
                        + "a great number.");
                N=1_000_000;
                operationSet=true;
            }
        }

        else{
            System.out.println("Ok, it is not an integer. I set the number of "
```

```
                + "operations to " + N);
        operationSet=true;
    }
}

Scanner sc= new Scanner(System.in);
int input = (int) (Math.random()*65)-1;
System.out.print("\nHow many threads should work concurrently? "
        + "Enter an integer(Between 1 and 64): \n");
    if(sc.hasNextInt()){
        input = sc.nextInt();


        if(input>64 || input<1){
            System.out.println("This is offlimits. "
                    + "I will calculate from 64.");
            input=64;
            hasNext=false;
        }

    }else{
        System.out.println("You should have entered an integer only... "
                + "\nOk I will randomize the number of the threads, "
                + "which is " + input + " now." + div);
    }

do{

long startTime,endTime,totalTime;
int threadCount = input;

System.out.println("The thread count is " + threadCount +
        " and number of iterations is " + N + "\nThis means there will be "
        + N/2 + " add and sub operations each.");


PiThread[] threads = new PiThread[threadCount];
startTime = System.nanoTime();

for (int i = 0; i < threadCount; i++) {
    threads[i] = new PiThread(threadCount, i, N);
    threads[i].start();
}

for (int i = 0; i < threadCount; i++) {
    threads[i].join();
```

```
}

double pi = 0;
for (int i = 0; i < threadCount; i++) {
    pi += threads[i].getSum();
    // a+=threads[i].getCount()-1;
}

endTime=System.nanoTime();
totalTime=TimeUnit.NANOSECONDS.toMillis(endTime-startTime);

String heading1 = "The Real PI is";
String heading2 = "Calculated PI is";

System.out.printf( "\\n\%-25s \%s \%n\%-25s \%s \%n",
                    heading1, Math.PI, heading2, pi);


System.out.println("\nTime spent: " + totalTime + " ms"
                  + "\nThe approximation error: " + (Math.PI-pi) + div);

Scanner scanner=new Scanner(System.in);
answer="N";
if(answer.equalsIgnoreCase("N")){
    if (input<64){
        System.out.println("Want to add one more thread? Y/N?");
        answer=scanner.next();
        if (answer.equalsIgnoreCase("Y")){
            System.out.println("OK, Calculating again");
            System.out.println("The new thread count is " + ++input + div);
        }
         else{
            System.out.println("Understood, bye.");
            answer="N";
        }
    }
}
}
while(hasNext && input<65 && answer.equalsIgnoreCase("Y"));
}

static class PiThread extends Thread{

private final int threadCount;
private final int threadRemainder;
private final int N;
```

```
    private double sum  = 0;

    public PiThread(int threadCount, int threadRemainder, int n) {
        this.threadCount = threadCount;
        this.threadRemainder = threadRemainder;
        N = n;
    }

    @Override
    public void run() {

        for (int i = 0; i <= N; i++) {
            if (i \% threadCount == threadRemainder) {.
                sum += 4*( Math.pow(-1, i) / (2 * i + 1) );
            }
        }
    }

    public double getSum() {
        return sum;
    }
    }
}
```

References

[1] Code5. *How to write PI calculation program in Java using multi-thread*, ? (accessed 5 May, 2020).

[2] MrYouMath. *Taylor Series - 4 - Arctan and Pi*, 2014 (accessed 5 May, 2020).

[3] MIT Opencourseware. *Taylor Serie Example of Isaac Newton*, 2010 (accessed 5 May, 2020).

[4] Ranjan Roy. *The Discovery of the Series Formula for by Leibniz, Gregory and Nilakantha*, 2017 (accessed 5 May, 2020).

[5] SteamKing. *Taylor Series Approximation for Pi*, 2015 (accessed 5 May, 2020).

[6] Arts University of Pennsylvania and Sciences Dept. *Taylor Polynomials*, 2017 (accessed 5 May, 2020).

[7] David Wilson. *The History of Pi*, 2000 (accessed 5 May, 2020).

[8] Leonardo Henrique Zapparoli. *Java Concurrency Patterns*, 2019 (accessed 5 May, 2020).