Assignment 1

Problem 1 - Approximating Pi

Description

To approximate Pi using the Bailey-Borwein-Plouffe formula, we define a number of digits we want the output to be formatted in, a number of terms we want the computation to be run for increased precision and finally we pass an executer service to run the method in parallel.

We split the work being the terms to the amount of cores the machine has available. This guarantees an even distribution of the calculations across all threads. The tasks get submitted to the executer service and are computed.

Finally, we iterate over the futures in sequence and add the partial sums to the total sum. Since the work was evenly distributed, this sequential processing is efficient, as we expect all tasks to be finished or nearly finished by the time the first one completes. Thus, the processing of the first future in the loop pauses further execution until it completes, but by that time, most partial sums computed in parallel should be available, minimizing any additional waiting. The executer service automatically sends a new task to a free core as soon as the compution has been completed.

Run Code

To run the code, you can just compile and execute the main method of problem_1 file/class.

```
javac problem_1.java && java problem_1
```

Or could just run the publicly available method `approximatePiParallel`.
The non-parallel function `approximatePi` is also available.

Results

As we would expect, running the `approximatePi` function which is not doing any work in parallel takes much longer than `approximatePiParallel`.

Both methods are executed with the same parameters being
- `digits`: 7
- `terms`: 1000000
The function `approximatePi` takes 134.848 seconds while `approximatePiParallel` accomplishes to make the calculation in just 4.883 seconds showing a significant difference in performance between both functions. The code was executed on a MacBook Pro with M1 Pro Chip.

Examining the scalibility, we can clearly see that the more cores of a CPU are utilized, the faster the calculation finishes. That applies to a small amount of terms, as well as to large amount of terms. However, we can see that the jump from 4 threads to 8 is not as significant as the jump from 1 to 2 or from 2 to 4.

```
Testing with 1 threads...
   Terms: 10000, Execution time: 0,10 seconds, Pi approximation: 3.1415927
   Terms: 100000, Execution time: 0,47 seconds, Pi approximation: 3.1415927
   Terms: 500000, Execution time: 5,02 seconds, Pi approximation: 3.1415927
   Terms: 1000000, Execution time: 16,67 seconds, Pi approximation: 3.1415927
   Terms: 2000000, Execution time: 59,57 seconds, Pi approximation: 3.1415927
Testing with 2 threads...
   Terms: 10000, Execution time: 0,01 seconds, Pi approximation: 3.1415927
   Terms: 100000, Execution time: 0,19 seconds, Pi approximation: 3.1415927
   Terms: 500000, Execution time: 2,58 seconds, Pi approximation: 3.1415927
   Terms: 1000000, Execution time: 8,42 seconds, Pi approximation: 3.1415927
   Terms: 2000000, Execution time: 30,24 seconds, Pi approximation: 3.1415927
Testing with 4 threads...
   Terms: 10000, Execution time: 0,01 seconds, Pi approximation: 3.1415927
   Terms: 100000, Execution time: 0,10 seconds, Pi approximation: 3.1415927
   Terms: 500000, Execution time: 1,38 seconds, Pi approximation: 3.1415927
   Terms: 1000000, Execution time: 4,51 seconds, Pi approximation: 3.1415927
   Terms: 2000000, Execution time: 16,39 seconds, Pi approximation: 3.1415927
Testing with 8 threads...
   Terms: 10000, Execution time: 0,01 seconds, Pi approximation: 3.1415927
   Terms: 100000, Execution time: 0,08 seconds, Pi approximation: 3.1415927
   Terms: 500000, Execution time: 0,92 seconds, Pi approximation: 3.1415927
   Terms: 1000000, Execution time: 3,12 seconds, Pi approximation: 3.1415927
   Terms: 2000000, Execution time: 11,41 seconds, Pi approximation: 3.1415927
```

Problem 2 - Applying Filters on an Image

Description

The task involves implementing two image processing techniques: Gaussian blur and Sobel filter, with a focus on optimizing the Gaussian blur using parallel processing in Java. The Gaussian blur algorithm applies a kernel to each pixel of the image to produce a blurred effect. This process is computationally intensive, especially for large images or when using a large radius for the blur effect.

The code is structured to allow comparison between a non-parallel (sequential) version and a parallelized version of the Gaussian blur. The parallelized version utilizes Java's ForkJoinPool framework, which enables efficient execution of parallel tasks, particularly beneficial for CPU-bound tasks like image processing. By dividing the image into chunks and processing each chunk in parallel, we aim to significantly reduce the overall execution time. The chunks are disgned to be of similar size, so that the calculation should take roughly the same time for every chunk.

The execution begins through looping over all tasks and calling `pool.execute(task)`. After completing the loop. We enter into another one to get or await the tasks value through `task.get()`. In this case it's actually just the completion of the `void` returning method. `task.get()` is a blocking call. So as we wait for the calculation of one task in one thread we actually also wait implicitly on all other threads to complete their current task. The thread pool autmatically starts the next task in an available thread, so that waiting time is actually minimal. It is important though to access the tasks value in the same order they were started.
Since the image is passed to every single task, we have to ensure that the tasks do not interfere with each other through simultaneous modification. To avoid this, we use Java's `synchronized` keyword on the `outputImage` object. This ensures that only one thread can modify the image at a time, preventing race conditions and ensuring the integrity of the image processing operation.

Run Code

To run the code again you'll need Java 8 or newer. The file needs to be compiled and can then be run, similar to problem 1.

```
javac problem_2.java && java problem_2
```

The main method has 3 methods to choose between the two serial implementaions of the algorithms or run the scalability test to get the console logs which can be seen below.

Also you can just access the static methods from the package after you compile it. The method `createBlurredImage()` let's you blur any image from the assets folder with available parameters. The same applies to the `createImageWithSharpEdges()` method. If the parallel algorithm should not be executed, then simply pass a `null` ForkJoinPool. Otherwise create it, set the amount of cores to utilize and then pass the parameter to the methods.

Results

The execution times for the Gaussian blur with different radius values were recorded, highlighting the performance benefits of parallel processing. Here's a summary of the observed results:

```
Testing parallel gaussian blur with 1 threads...
   Radius: 1, Execution time: 0,68 seconds
   Radius: 5, Execution time: 4,31 seconds
   Radius: 7, Execution time: 7,60 seconds
   Radius: 10, Execution time: 14,46 seconds
   Radius: 20, Execution time: 53,81 seconds
Testing parallel gaussian blur with 2 threads...
   Radius: 1, Execution time: 0,55 seconds
   Radius: 5, Execution time: 2,46 seconds
   Radius: 7, Execution time: 4,16 seconds
   Radius: 10, Execution time: 7,82 seconds
   Radius: 20, Execution time: 28,44 seconds
Testing parallel gaussian blur with 4 threads...
   Radius: 1, Execution time: 0,56 seconds
   Radius: 5, Execution time: 1,46 seconds
   Radius: 7, Execution time: 2,36 seconds
   Radius: 10, Execution time: 4,19 seconds
   Radius: 20, Execution time: 14,72 seconds
Testing parallel gaussian blur with 8 threads...
   Radius: 1, Execution time: 0,53 seconds
   Radius: 5, Execution time: 1,30 seconds
   Radius: 7, Execution time: 1,65 seconds
   Radius: 10, Execution time: 3,00 seconds
   Radius: 20, Execution time: 10,41 seconds
```

The results clearly demonstrate the efficiency of parallelizing the Gaussian blur operation. We see that stepping up to utilizing 2 and also 4 cores brings significant performance improvements. However, at the same time, even though the numbers are lower in all cases, we notice that the numbers at 8 threads are not as impressive as the other two thread bumps. Still running the algorithm in this implementation in parallel always brings us a benefit. It might be interesting to see the performance on a machine with more than 8 cores to see how it scales there.

For the Sobel filter, running the calculations in parallel results in only slight improvements. This is primarily due to the inherently lower complexity of the Sobel filter algorithm compared to the Gaussian blur as can be seen in the following numbers.

```
Testing parallel sobel edge detection with 1 threads...
   Execution time: 0,66 seconds
Testing parallel sobel edge detection with 2 threads...
   Execution time: 0,56 seconds
Testing parallel sobel edge detection with 4 threads...
   Execution time: 0,54 seconds
Testing parallel sobel edge detection with 8 threads...
   Execution time: 0,54 seconds
```

The Gaussian blur's computational load can be significantly increased by parameters such as `radius`, which directly impacts the size of the convolution kernel and, consequently, the number of calculations required per pixel. In contrast, the Sobel filter uses a fixed-size kernel (typically 3x3), leading to a relatively consistent and lower computational load regardless of the image size. Therefore the overhead associated with managing parallel tasks can offset the gains from distributing this workload across multiple processors.

Problem 3 - Sorting

Description

To sort numbers sequentially, we decided to use quicksort. The selection of the pivot value is simplified and uses always the list element located at the index equalling half of the size of the whole list. Afterwards, we build three lists: on for the elements less than the pivot value, one for the elements equalling the pivot value and one for all greater values. For the result list, we call the method recursive and add the results in the order less-equal-greater to the result list.

The parallel version uses the same algorithm but creates a task for each recursive call except for the "equal" list. Like that, the branching has to reach a specific amount of recursive calls that all threads can be used efficiently. Depending on the size of the ordered list, this can be an issue.

Run code

To run the code, execute the main method of the file problem_3.java. The example code sorts all the instances located in the `inputs` folder. To run your own instances simply add them to the folder and adjust the loops in the main method.

The main method executes both sequential quicksort and parallel quicksort for all of the json files and produces the output you can see in the `Results` section.

Results

The intention of using quicksort in parallel was mainly the easy implementation. After implementing sequential quicksort it wasn't hard to add the recursive calls as a task to a threadpool. We chose to generate different number arrays, especially with different amounts of numbers. To decrease the randomness of the results, we also created multiple instances of the same size.

We tested number arrays of the sizes 1.000, 10.000, 100.000, 1.000.000 and 10.000.000. The results are displayed below. Note that every line uses a different number array of the specified size so the computing times may differ for the same instance size.

```
Instance size 10^3

Quicksort: 4.008143 ms,  Parallel quicksort: 6.940235 ms
Quicksort: 1.244997 ms,  Parallel quicksort: 3.111935 ms
Quicksort: 1.29242 ms,  Parallel quicksort: 2.514369 ms
Quicksort: 1.479525 ms,  Parallel quicksort: 4.888776 ms
Quicksort: 0.759597 ms,  Parallel quicksort: 2.508852 ms

Instance size 10^4

Quicksort: 11.109433 ms,  Parallel quicksort: 34.553322 ms
Quicksort: 9.438751 ms,  Parallel quicksort: 32.310287 ms
Quicksort: 9.688016 ms,  Parallel quicksort: 32.958697 ms
Quicksort: 9.614682 ms,  Parallel quicksort: 35.071267 ms
Quicksort: 6.95148 ms,  Parallel quicksort: 34.046481 ms

Instance size 10^5

Quicksort: 90.101464 ms,  Parallel quicksort: 48.135499 ms
Quicksort: 71.562276 ms,  Parallel quicksort: 23.274174 ms
Quicksort: 94.281767 ms,  Parallel quicksort: 27.950352 ms
Quicksort: 46.173227 ms,  Parallel quicksort: 20.438602 ms
Quicksort: 43.050188 ms,  Parallel quicksort: 19.026892 ms

Instance size 10^6

Quicksort: 899.456967 ms,  Parallel quicksort: 417.058267 ms
Quicksort: 679.049409 ms,  Parallel quicksort: 393.423572 ms
Quicksort: 857.422417 ms,  Parallel quicksort: 405.725551 ms
Quicksort: 918.313865 ms,  Parallel quicksort: 321.428626 ms
Quicksort: 769.352086 ms,  Parallel quicksort: 360.956001 ms

Instance size 10^7

Quicksort: 9367.066654 ms,  Parallel quicksort: 5834.915436 ms
Quicksort: 9699.654091 ms,  Parallel quicksort: 6326.136813 ms
Quicksort: 9545.941334 ms,  Parallel quicksort: 5719.366675 ms
Quicksort: 9157.800585 ms,  Parallel quicksort: 5346.055685 ms
Quicksort: 9665.90311 ms,  Parallel quicksort: 5485.32847 ms
```

For the small instances (10^3 and 10^4), the parallel implementation of quicksort has a worse runtime that is about 3-4 times the runtime of sequential quicksort. That's because it takes some time for the parallel algorithm to use all available threads and also to start and stop each thread.

If the instances get bigger than 10^5 the parallel algorithm is able to perform better than the sequential one. For our biggest test instances the runtime is reduced by about 40% compared to the runtime of non-parallel quicksort.

If we compare our parallel algorithm to the `ArrayList.sort()` method provided by java, we get the following results:

```
Instance size 10^4

Quicksort: 9.701575 ms,  Parallel quicksort: 26.231786 ms
Quicksort: 4.405686 ms,  Parallel quicksort: 24.644772 ms
Quicksort: 4.591745 ms,  Parallel quicksort: 23.214903 ms
Quicksort: 4.073868 ms,  Parallel quicksort: 25.01207 ms
Quicksort: 2.756654 ms,  Parallel quicksort: 22.5139 ms

Instance size 10^5

Quicksort: 58.132135 ms,  Parallel quicksort: 53.088445 ms
Quicksort: 48.505361 ms,  Parallel quicksort: 35.292074 ms
Quicksort: 23.492104 ms,  Parallel quicksort: 29.336949 ms
Quicksort: 18.581951 ms,  Parallel quicksort: 20.434083 ms
Quicksort: 17.661437 ms,  Parallel quicksort: 24.124451 ms

Instance size 10^6

Quicksort: 401.449706 ms,  Parallel quicksort: 320.617472 ms
Quicksort: 349.680641 ms,  Parallel quicksort: 276.659366 ms
Quicksort: 345.675776 ms,  Parallel quicksort: 308.186272 ms
Quicksort: 344.035474 ms,  Parallel quicksort: 353.151076 ms
Quicksort: 339.278134 ms,  Parallel quicksort: 334.684993 ms

Instance size 10^7

Quicksort: 5633.891551 ms,  Parallel quicksort: 4455.017274 ms
Quicksort: 5669.133478 ms,  Parallel quicksort: 4378.58507 ms
Quicksort: 5599.318218 ms,  Parallel quicksort: 4953.55837 ms
Quicksort: 5625.694436 ms,  Parallel quicksort: 4250.218694 ms
Quicksort: 5593.994184 ms,  Parallel quicksort: 4842.877945 ms
```

We can see that the algorithm is faster than our implementation of sequential quicksort. Nevertheless, the parallel algorithm performs better for the biggest lists of numbers. The sequential algorithm is better for some instances until the size of 10^6 but loses more and more of its advantage until its runtime is worse for every single one of the given test instances for the size 10^7. There we get a performance boost of about 15-20%.

Problem 4 - Iterative Solver

Description

For this task we decided to switch to python since the provided code was in python, but also because the numpy module in python just makes it easier to work with matrices. First we took a look at the provided code and tried to parallelize that one by simply utilizing the numba library. Decorating the `gauss_seidel_step` method with the `@njit` annotation setting the `parallel` flag to `True` and using `prange` for the outer loop improved the performance significantly.

During the initial examination of the provided code, we identified a potential optimization in the gauss_seidel_step method. The original increment for updating the grid was modified from `diff**2 * diff**2` to `diff**2`, aligning with standard practices for calculating residuals in iterative solvers. This correction ensures that the algorithm accurately measures convergence towards the solution.

Then we implemented the red-black aka. chessboard strategy from the lecture. We first calculate all the red cells and then all the black cells and add the accumulated results together. For each color we iterate over the rows in parallel and over the columns in the same thread, and calculate the partial results.

Numba's JIT compliation takes quite some time when a method is executed for the first time. For that reason, we execute the calculation first with minimal numbers only for the sake of triggering the JIT compilation for numba. Thus, no numbers are influenced by that.

Run code

To run the code, just execute the files main method, or call the modules `gauss_seidel_step` method for a parallelized version of the provided code or `gauss_seidel_step_chessboard` for the chessboard strategy. It takes a two dimensional matrix and returns the numeric residual value.

Results

Overall, we can see significant improvements with the parallelization. However, the results might not be what we expect at first thought.

Below we can see the execution times with different thread numbers and grid sizes. We get consistent results ±1sec running the code on a MacBook Pro with M1 Pro chip.

Chessboard Strategy

We can clearly see that if our grid size is ≤ 500, then an increased number of threads is not beneficial and even leads to increased execution times. But if the grid size is higher than that, then execution times get significantly lower. The reason for this is that a thread should have enough work to compute in order to overshadow the cost of creating a new thread.

In conclusion we can say, that this algorithm and this implementation only scale well with more threads if the grid size also is higher than a respective value

```
Testing with 1 threads...
  Grid Size: 100x100, Execution Time: 0.24 seconds, after 2904 iterations, residual = 4.99
  Grid Size: 250x250, Execution Time: 2.23 seconds, after 12128 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 15.41 seconds, after 24999 iterations, residual = 8
  Grid Size: 1000x1000, Execution Time: 54.77 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 113.55 seconds, after 24999 iterations, residual =
Testing with 2 threads...
  Grid Size: 100x100, Execution Time: 0.20 seconds, after 2904 iterations, residual = 4.99
  Grid Size: 250x250, Execution Time: 1.59 seconds, after 12128 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 9.50 seconds, after 24999 iterations, residual = 8.0
  Grid Size: 1000x1000, Execution Time: 28.89 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 59.32 seconds, after 24999 iterations, residual =
Testing with 4 threads...
  Grid Size: 100x100, Execution Time: 0.22 seconds, after 2904 iterations, residual = 4.99
  Grid Size: 250x250, Execution Time: 1.69 seconds, after 12128 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 8.02 seconds, after 24999 iterations, residual = 8.0
  Grid Size: 1000x1000, Execution Time: 16.91 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 33.38 seconds, after 24999 iterations, residual =
Testing with 8 threads...
  Grid Size: 100x100, Execution Time: 0.29 seconds, after 2904 iterations, residual = 4.99
  Grid Size: 250x250, Execution Time: 1.77 seconds, after 12128 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 7.15 seconds, after 24999 iterations, residual = 8.0
  Grid Size: 1000x1000, Execution Time: 16.85 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 32.55 seconds, after 24999 iterations, residual =
```

Parallelized version of the provided code

Here we don't see such a threshold needed to be past in order to be worth the overhead of the parallelism. For very low grid sizes it performs slightly better than the chessboard approach, but for higher grid sizes it's a lot less performant.

```
Testing with 1 threads...
  Grid Size: 100x100, Execution Time: 0.20 seconds, after 2902 iterations, residual = 4.99
  Grid Size: 250x250, Execution Time: 4.13 seconds, after 12123 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 29.66 seconds, after 24999 iterations, residual = 8
  Grid Size: 1000x1000, Execution Time: 111.54 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 253.31 seconds, after 24999 iterations, residual =
Testing with 2 threads...
  Grid Size: 100x100, Execution Time: 0.16 seconds, after 2902 iterations, residual = 4.97
  Grid Size: 250x250, Execution Time: 2.48 seconds, after 12123 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 17.24 seconds, after 24999 iterations, residual = 8
  Grid Size: 1000x1000, Execution Time: 63.39 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 127.47 seconds, after 24999 iterations, residual =
Testing with 4 threads...
  Grid Size: 100x100, Execution Time: 0.16 seconds, after 2906 iterations, residual = 4.94
  Grid Size: 250x250, Execution Time: 1.85 seconds, after 12123 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 9.63 seconds, after 24999 iterations, residual = 8.0
  Grid Size: 1000x1000, Execution Time: 38.34 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 77.40 seconds, after 24999 iterations, residual =
Testing with 8 threads...
  Grid Size: 100x100, Execution Time: 0.15 seconds, after 2906 iterations, residual = 4.90
  Grid Size: 250x250, Execution Time: 1.84 seconds, after 12117 iterations, residual = 4.9
  Grid Size: 500x500, Execution Time: 8.56 seconds, after 24999 iterations, residual = 8.0
  Grid Size: 1000x1000, Execution Time: 26.33 seconds, after 24999 iterations, residual =
  Grid Size: 1500x1500, Execution Time: 55.79 seconds, after 24999 iterations, residual =
```