Assignment 2

Konstantin Fritzsch, Christopher Knapp

Problem 1 - Approximating Pi

Description

Like in assignment 1, we use the Bailey-Borwein-Plouffe formula. We again define a number of digits and a number of terms. To implement a parallelization with the mpi strategy, we use the recommended mpi4py and create an instance with `MPI.COMM_WORLD`.

The parallelization works like in assignment1 and splits the number of terms depending on the number of the available mpi instances. Then, every instance does the calculation for the given parameters. After finishing the calculation, the results are collected using `MPI.COMM_WORLD.reduce` with the operation parameter set to `MPI.SUM` so we receive the expected sum.

The time calculation starts before the term seperation and ends when finishing the calculation so we track the effort of both, splitting the work and calculating the sum. The instance with rank 0 then prints the output showing us the execution time for the different amount of terms.

Run Code

To run the code sequentially, you can use the function `approximate_pi` and just simply run the python file.

```
python problem_1.py
```

For the parallel execution with mpi4py you need to use the function `approximate_pi_parallel` and run it with the following command:

```
mpiexec -n 4 problem_1.py
```

MPI has to be installed for using this command. The parameter n defines the number of instances to use for the calculation.

Results

The results of the calculation can be seen in the following table. We compare the sequential runtime to the runs using mpi
with different values for the n parameter that defines the number of mpi instances.

n	10.000 (ms)	100.000 (ms)	500.000 (ms)	1.000.000 (ms)	2.000.000 (ms)
seq	5.02	47.28	234.45	473.47	915.98
1	4.69	48.30	228.57	451.78	912.84
2	4.36	35.44	113.93	228.05	467.68
4	2.88	17.76	86.27	172.68	301.29
8	1.82	10.33	49.88	98.68	193.53
12	7.17	11.07	44.65	75.45	137.12
16	45.75	19.93	46.42	102.83	143.71

Running the calculation sequentially has nearly the same runtime results as the calculation using mpi with `n=1` which of course makes sense because in both cases there is only one worker doing the calculation.

The best results are calculated when using 12 instances. This might correlate with the computer that was used for the calculation, because it has 12 kernels. For the small amount of terms, the runtime increases when using more than 8 instances. The cost for creating the mpi instances is significantly higher than the improvement with the parallelization. When it comes to a higher amount of terms, we can see a good decrease of runtime. For example for 2 million terms using 12 mpi instances, we get a 6.68 times improvement which is a good improvement in general but not as good as expected, if we have 12 parallel runners.

It is very interesting that the runtime for 10.000 terms with `n=16` is more than double the runtime of 100.000 terms. This could be seen repeatedly on every run. It seems that there is a high initial cost for creating all the mpi instances in the first loop iteration. Although all the different term sizes are calculated repeatedly in the loop, there might be some internal caching process by mpi4py so that the next iterations are faster than the first one.

Problem 2 - Applying Filters on an Image

Description

Gaussian Blur

As we decided to implement the MPI problems in python, first we implemented the gaussian blur sequentially without parallelization. Then, for curiosity, we also implemented a shared memory version using numba. Compared to the java implementation from assignment 1 it performed way better and that shows how well optimized the numba module is.
The parallel version with the MPI approach works the following: The image is read, a numpy array is being created from that and then we have to split the image into chunks in order to distribute fractions of the image to different workers. Even though a worker only gets a specific area of the image, still those areas need to overlap with other areas to compute the ideal blurred image. That's what the `distribute_chunks_with_overlap` method takes care of. The chunks are being distributed through `comm.scatter(chunks, root=0)` and the blur is applied to each single chunk. The results from each worker are being collected through `comm.gather(blurred_chunk, root=0)` and afterwards the overlapping parts need to be trimmed. Finally, the image needs to be created from the array and written to the target destination.

Sobel Edge Detection

For this filter we also implemented the serial version, as well as a parallel numba version and a parallel MPI version.
We could reuse a lot of the code used for the image blur and the only differences are actually that we do not have a kernel for this filter and have a different filter calculation per chunk. Further, the filter has no variable like the radius. Therefore, we still overlap the chunks, but keep it to the absolute minimum which is needed. Otherwise two pixels between the chunks would not receive the filter application.

Run Code

The script contains a sequential implementation, as well as a parallel numba implementation and a parallel MPI implementation for both, the gaussian blur and the sobel edge detection.

The methods are all publicly available in the script and the main block also contains many commented out methods for executing the filters or testing the scalability.

To run the sequential code or the numba code just run it through

```
python problem_2.py
```

To run the MPI version though it's important to have `open-mpi` installed. Then it can be run through the code below, while flag `-n` sets the number of cores to use.

```
mpiexec -n 8 python problem_2.py
```

Results

Gaussian Blur

In the sequential approach, where the code runs synchronously, the execution time increases modestly as the radius increases, starting from 15.17 seconds for a radius of 1 and going up to 23.41 seconds for a radius of 20. This gradual increase is expected because a larger radius means more data points are involved in the calculations for each pixel's blur effect, leading to slightly longer execution times.

We have to note that compared to the implementation in Java from assignment 1, the execution times start being quite long with 15.17 seconds compared to 0,68 seconds. However, the execution times increase significantly less with higher radii. With a radius of 20 in the python implementation we get an execution time of 23.41 seconds, while it is 53,81 seconds in the Java implementation, what is very surprising.

Execution Mode	Threads/Processes	Radius	Execution Time (s)
Sequential	N/A	1	15.17

Execution Mode	Threads/Processes	Radius	Execution Time (s)
Sequential	N/A	5	16.02
Sequential	N/A	7	16.35
Sequential	N/A	10	17.51
Sequential	N/A	20	23.41

Since the execution times in the parallel MPI version are substantually higher, we chose to use lower radii than the ones above for the comparison.
The use of Numba for parallel execution dramatically improves performance, showcasing the benefits of parallel computing. With just 1 thread, the execution times are significantly reduced compared to the sequential execution, starting at 0.75 seconds for radius 1 and reaching up to 2.05 seconds for radius 9. This already represents a substantial improvement. Increasing the number of threads further decreases the execution time. With 2 threads, the execution time for radius 1 drops to 0.07 seconds, and with 8 threads, it remains low at 0.05 seconds for the same radius. For larger radii, the execution times also decrease with more threads, demonstrating excellent scalability with the number of threads up to 8, where execution time for radius 9 is just 0.38 seconds.

The performance of the Gaussian blur using MPI for parallel execution, surprisingly, does not follow the trend seen with Numba. Despite parallelizing the computation across multiple processes, the execution times are significantly longer than both the sequential and Numba parallel executions.
For instance, running the operation with a single process (which essentially simulates a sequential execution in an MPI framework) yields a much higher execution time of 31.83 seconds for radius 1 and even longer for radius 3, at 158.23 seconds, before the process was manually aborted due to excessive duration.
Even when increasing the number of processes to 2 and 4, the execution times, although improved, remain much higher than the Numba parallel execution. For example, with 4 processes, the execution time for radius 1 is 8.39 seconds, and for radius 9, it escalates to 301.70 seconds. The performance does improve as the number of processes increases to 8, but the execution times are still not competitive with the Numba implementation, marking 5.29 seconds for radius 1 and 195.39 seconds for radius 9.

Parallel Execution Efficiency Comparison

Configuration	Radius 1	Radius 3	Radius 5	Radius 7	Radius 9
Sequential	14.90s	15.30s	15.67s	16.01s	16.52s
Numba 1 Thread	0.75s (-94.96%)	0.32s (-97.91%)	0.72s (-95.41%)	1.28s (-92.00%)	2.05s (-87.56%)
Numba 2 Threads	0.07s (-99.53%)	0.17s (-98.89%)	0.39s (-97.51%)	0.67s (-95.81%)	1.06s (-93.59%)
Numba 4 Threads	0.05s (-99.66%)	0.11s (-99.28%)	0.21s (-98.66%)	0.38s (-97.63%)	0.57s (-96.55%)
Numba 8 Threads	0.05s (-99.66%)	0.09s (-99.41%)	0.16s (-98.98%)	0.26s (-98.38%)	0.38s (-97.70%)
MPI 1 Process	31.83s (+113%)	158.23s (+933%)	-	-	-
MPI 2 Processes	16.47s (+10.48%)	81.36s (+431%)	196.96s (+1153%)	-	-
MPI 4 Processes	8.39s (-43.69%)	42.03s (+174%)	100.80s (+542%)	187.10s (+1067%)	301.70s (+1725%)
MPI 8 Processes	5.29s (-64.50%)	25.76s (+68.56%)	62.57s (+298%)	120.31s (+651%)	195.39s (+1082%)

- "-" indicates the execution was manually aborted due to excessive duration.
- Percentage changes are calculated relative to the sequential execution time for the same radius.

The contrast in performance between Numba and MPI implementations for parallel computing in this case study is noteworthy. Numba's parallel execution with threads demonstrates exceptional efficiency and scalability for the Gaussian blur operation, significantly outperforming the MPI approach, which struggles to achieve similar efficiency gains. This might be surprising given MPI's widespread use in high-performance computing for distributed memory systems. The poor performance of MPI in this context could be attributed to overheads associated with inter-process communication, which become pronounced for operations like Gaussian blur that require intensive data exchange. This analysis underscores the importance of choosing the right parallel computing approach based on the specific nature of the task and the computational resources available.

> Even though we introduce overheads with inter-process communication, we should still note that we get almost double the execution time when executing the mpi code with just one process. The execution times decrease with a higher number of processes, but the number for just one process suggests that the implementation contains a significant problem which introduces quite big overheads.

Sobel Edge Detection

The sequential (or serial) execution of the Sobel edge detection took 9.97 seconds. This sets the baseline for comparing the efficiency gains of parallel implementations. Unlike the Gaussian blur, where execution time increases with radius, Sobel edge detection's complexity primarily depends on the image size, not the variable parameters of the operation.

Numba again proves to be highly optimized as it even only takes 0.94 seconds with just one thread. Bumping up to 2 threads leads to an execution time of only 0.04 seconds. The same applies to thread counts of 4 and 8.

Running the filter in parallel by using MPI with 1 process essentially simulates a sequential execution in an MPI framework, resulting in 11.05 seconds, which is interestingly longer than the serial implementation. With 2 processes, the execution time decreases to 5.75 seconds, taking only halv of the time. Increasing to 4 processes decreases the execution time to 2.90 seconds, and is again considerably faster. Finally, with 8 processes, the execution time did decrease again, even though not significantly, but still bringing a considerable benefit.

Sobel Edge Detection Execution Time Comparison

Method	Configuration	Execution Time (s)	Improvement over Sequential
Sequential	N/A	9.97	N/A
Numba	1 thread	0.95	Improved by 9.02s
Numba	2 threads	0.04	Improved by 9.93s

Method	Configuration	Execution Time (s)	Improvement over Sequential
Numba	4 threads	0.04	Improved by 9.93s
Numba	8 threads	0.04	Improved by 9.93s
MPI	1 process	11.05	Slower by 1.08s
MPI	2 processes	5.75	Improved by 4.22s
MPI	4 processes	2.90	Improved by 7.07s
MPI	8 processes	2.09	Improved by 7.88s

This table highlights the effectiveness of parallel computing in reducing execution times for image processing tasks like Sobel edge detection. Numba's multi-threading capability shows an impressive performance, drastically reducing execution times to near-instantaneous levels with minimal overhead for increasing threads. In contrast, MPI shows a more gradual improvement as the number of processes increases, with diminishing returns indicating the overhead associated with inter-process communication.

Conclusion

The use of MPI for parallel processing demonstrated significant benefits for the Sobel edge detection, effectively reducing the execution time with increasing numbers of processes. This improvement contrasts sharply with its performance on the Gaussian blur, where MPI execution times were unexpectedly high, even with multiple processes. This discrepancy can be attributed to the nature of the algorithms and their parallelization potential. Sobel edge detection, being a more straightforward pixel-wise operation, scales better with MPI due to less inter-process communication overhead. On the other hand, Gaussian blur involves more extensive data dependencies and communication overheads, making it less suited for MPI's process-based parallelism. This analysis underscores the importance of algorithm characteristics in selecting an appropriate parallelization strategy.

Problem 3 - Sorting

Description

We used our algorithm of assignment 1 and translated it to python to use mpi4py. The sequential algorithm is still quicksort with the pivot value at the index equalling half the list size.

To parallelize this algorithm we have to keep track of the communication between the single instances. That's why we decided to distribute the list into equal parts and give each instance a part of the original list to sort. After that sorting, the mpi instance with rank 0 gathers the results and merges the single sorted lists together by searching for the smallest first item of all the sorted lists and then adding it to the result. This can be done efficiently using a stack data structure for the single lists.

Run code

Like in assignment 1, we use random generated lists with different sizes to measure the runtime. The lists are not provided because they take a lot of storage space but they have to be located in the inputs folder and look like the file `random_integers.json` (located also inside this folder) to get the program running.

The main function of the file iterates over multiple of these files with special names so it might be necessary to either create files with the same names or change the logic of the loops.

To run the code use this command again

```
mpiexec -n 4 problem_3.py
```

and adjust the number of instances by changing the n parameter.

Results

We get the different results by outputs looking like that:

```
Instance size 10^3

Quicksort: 0.11 ms, Parallel quicksort: 21.39 ms
Quicksort: 0.08 ms, Parallel quicksort: 7.80 ms
Quicksort: 0.08 ms, Parallel quicksort: 15.55 ms
Quicksort: 0.09 ms, Parallel quicksort: 8.30 ms
Quicksort: 0.08 ms, Parallel quicksort: 14.90 ms
```

To keep track of all the different results also with multiple values for the n parameter, we calculated the averages of all five results and put them into the following table.

n	10^3 (ms)	10^4 (ms)	10^5 (ms)
Sequential	1.05	12.33	168.55
Sorted	0.07	0.95	14.44
1	1.67	22.38	831.77
2	1.13	14.12	401.07
4	0.97	9.42	217.74
8	1.59	13.04	216.68
12	2.25	15.51	212.19
16	14.12	48.73	356.20

The results of this parallelization is not really good. For small number of instances, the effort for the list splitting and merging is bigger than the improvements of the parallel work so we just increase runtime. We have an optimum with slight improvements for the smaller list at `n=4`. With higher n values we keep increasing the runtime, also because it gets more complicated to merge the more lists when finishing the sorting. If we compare the runtime to the already predefined `sorted` function in python, there is no reason to parallelize the sorting algorithm because the predefined version is so much faster.

Compared to assignment 1 and using a threadpool to add the single recursive tasks to this pool, the parallel version of quicksort using mpi is just not meaningful. The communication inside a threadpool is so much easier and a new task can be instantly started after finishing one. With mpi, we firstly needed to simplify the parallelization of the algorithm, and then do the not optimal merging between the lists that only one mpi instance can do, so we have a lot of effort for the branch and bound of the task and wait for single instances to finish their work which slows down the whole algorithm.

Problem 4 - Iterative Solver

Description

This problem was very hard to get parallelized. With the chosen strategy being the chessboard strategy or even just looping through the cells, it should not really be possible to parallelize the algorithm for one iteration and get accurate results. However, the parallelization may find justification if the use case is just to get a rough estimate of the needed iteration counts and if that's the goal, we actually have surprising results which suggest that you might find a benefit using the MPI parallelization strategy, but only with specific configurations.

To parallelize the algorithm, ideally you would instead look at another strategy. As one iteration should run sequentially because cells need their updated corresponding neighbours from top and left and therefore we would instead look to something different to parallelize. For example we could run multiple iterations in parallel instead and set specific boundaries which need to be passed in order for another work to start its calculation. That would require much more communication between the workers and was considered overkill for the sake of this exercise. But in theory, we should see quite some performance improvements while maintaining the same accuracy as the sequential algorithm.

We implemented two versions to solve the problem, or at least get near to an accurate solution.
In method `gauss_seidel_mpi()` we don't sync the colors with other workers after calculating the residual and updating the grid. In the method `gauss_seidel_mpi_chessboard_with_color_sync()` we do exactly that in hope to get a more accurate calculation.

Chessboard Strategy without color sync

So what we did, was to initialize the grid through the `init()` method from the given `heat.py` script and then divide the grid vertically into as many sections as we have workers.

The rows are overlapping, so that we apply the calculation to every cell as it would be in the sequential code.
We avoided initializing the grid just on rank 0 and scattering the sections because it was taking a lot of time. In our use case we always run the code on just one machine and therefore we decided to not waste more time with something having less relevance for the end result.
We loop as long as the iteration count is smaller than the maximum iteration count and the calculated summed residual is smaller than the aimed tolerance.
In every loop, we first send and receive the overlapping ghost cells from the neighbouring workers to have a more accurate calculation. Then we collect the results from all workers and sum them up. Finally, we check if we are done and communicate that to all workers.

Chessboard Strategy with color sync

In this method most things from the just mentioned approach also apply here. The only difference is, that for each color of the chessboard, we send and receive the ghost cells of the neighbouring workers and then calculate the residual to achieve some extra accuracy. The rest is pretty much identical.

Run code

The script contains a sequential implementation and a parallel MPI implementation.

The methods are all publicly available in the script and the main block also contains many commented out methods for executing or testing the scalability.

To run the sequential code or the numba code just run it through

```
python problem_4.py
```

To run the MPI version though it's important to have `open-mpi` installed. Then it can be run through the code below, while flag `-n` sets the number of cores to use.

```
mpiexec -n 8 python problem_4.py
```

Results

Comparing the code to the sequential approach, we have quite mixed results.
As mentioned before, with the chosen parallelization approach, we have decreased accuracy but can get quite some performance improvements.

```
Sequential Code
  Grid Size: 50x50, Execution Time: 2.48 seconds, after 920 iterations, residual = 4.9884
  Grid Size: 75x75, Execution Time: 11.02 seconds, after 1810 iterations, residual = 4.99
  Grid Size: 100x100, Execution Time: 31.01 seconds, after 2904 iterations, residual = 4.
  Grid Size: 150x150, Execution Time: 133.80 seconds, after 5562 iterations, residual = 4
```

Chessboard Strategy without color sync

As we would expect, running the code with only one process, being also a sequential execution, we get the anticipated overhead from the additional MPI code.

```
Testing with 1 processes...
  Grid Size: 50x50, Execution Time: 2.53 seconds, after 921 iterations, residual = 4.9884
  Grid Size: 75x75, Execution Time: 11.02 seconds, after 1811 iterations, residual = 4.99
  Grid Size: 100x100, Execution Time: 31.35 seconds, after 2905 iterations, residual = 4.
  Grid Size: 150x150, Execution Time: 134.21 seconds, after 5563 iterations, residual = 4
```

Then, with a parallelization with 2 processes, we almost halve the execution times and achieve very good parallelization while our iteration counts are slightly off.

```
Testing with 2 processes...
  Grid Size: 50x50, Execution Time: 1.29 seconds, after 917 iterations, residual = 4.9968
  Grid Size: 75x75, Execution Time: 5.68 seconds, after 1806 iterations, residual = 4.990
  Grid Size: 100x100, Execution Time: 15.71 seconds, after 2897 iterations, residual = 4.
  Grid Size: 150x150, Execution Time: 67.36 seconds, after 5552 iterations, residual = 4.
```

Running with 4 processes improves the performance even more while slightly losing some more accuracy. But we still almost get a 40% improvement in execution time!

```
Testing with 4 processes...
  Grid Size: 50x50, Execution Time: 0.79 seconds, after 930 iterations, residual = 4.99565
  Grid Size: 75x75, Execution Time: 3.13 seconds, after 1822 iterations, residual = 4.9900
  Grid Size: 100x100, Execution Time: 8.55 seconds, after 2917 iterations, residual = 4.99
  Grid Size: 150x150, Execution Time: 35.84 seconds, after 5576 iterations, residual = 4.9
```

However, if we then take a look at the performance of running the code with 8 processes, the implementation performs very poorly on small grids, being even a lot slower than the sequential implementation. Going up to a grid size of 150x150, we then get better execution times than the sequential code.
So, we can expect this implementation to perform better with multiple processes if the grid sizes also get higher.

```
Testing with 8 processes...
  Grid Size: 50x50, Execution Time: 13.37 seconds, after 951 iterations, residual = 4.9998
  Grid Size: 75x75, Execution Time: 22.52 seconds, after 1852 iterations, residual = 4.992
  Grid Size: 100x100, Execution Time: 39.52 seconds, after 2952 iterations, residual = 4.9
  Grid Size: 150x150, Execution Time: 104.73 seconds, after 5619 iterations, residual = 4
```

Chessboard Strategy with color sync

In this approach we overall gain some performance improvements but lower the accuracy, at least a lot more than with the other implementation. Also it is very interesting that even with one process our execution time decreases, what we would definitely not expect. This might suggest that implementation is not a 100% correct because we would expect higher accuracy coming from updating our cells more often.
In summary: MPI might not be the best choice for most problems. It is simply hard to get right and not really intuitive for the developer and you can spend days on finding bugs, as we did.

```
Testing with 1 processes...
  Grid Size: 50x50, Execution Time: 2.08 seconds, after 830 iterations, residual = 4.96858
  Grid Size: 75x75, Execution Time: 8.96 seconds, after 1608 iterations, residual = 4.9999
  Grid Size: 100x100, Execution Time: 25.19 seconds, after 2546 iterations, residual = 4.9
  Grid Size: 150x150, Execution Time: 106.50 seconds, after 4763 iterations, residual = 4
Testing with 2 processes...
  Grid Size: 50x50, Execution Time: 1.07 seconds, after 826 iterations, residual = 4.97707
  Grid Size: 75x75, Execution Time: 4.55 seconds, after 1592 iterations, residual = 4.9863
  Grid Size: 100x100, Execution Time: 12.52 seconds, after 2527 iterations, residual = 4.9
  Grid Size: 150x150, Execution Time: 53.27 seconds, after 4752 iterations, residual = 4.9
Testing with 4 processes...
  Grid Size: 50x50, Execution Time: 0.71 seconds, after 832 iterations, residual = 4.97612
  Grid Size: 75x75, Execution Time: 2.46 seconds, after 1598 iterations, residual = 4.9979
  Grid Size: 100x100, Execution Time: 6.80 seconds, after 2534 iterations, residual = 4.99
  Grid Size: 150x150, Execution Time: 27.34 seconds, after 4748 iterations, residual = 4.9
Testing with 8 processes...
  Grid Size: 50x50, Execution Time: 1.84 seconds, after 853 iterations, residual = 4.98413
  Grid Size: 75x75, Execution Time: 5.66 seconds, after 1595 iterations, residual = 4.9853
  Grid Size: 100x100, Execution Time: 12.99 seconds, after 2581 iterations, residual = 4.9
  Grid Size: 150x150, Execution Time: 44.00 seconds, after 4799 iterations, residual = 4.9
```