**Documentation**

**1. Java**

The results for Java took a bit, and from my analysis, they were pretty much very inconsistent for most of the given tests (jumping from 222ms to 43ms and so on), so these couldn’t be the best results, but nonetheless, they give us some insights.

For the test N=M=10, K=3, we can observe that the best execution time is the sequential one, this is probably because Java takes some time to get the Threads from the Operating System, so we can’t say much about this case because the inputs are so small, and the results are similar.

For the test N=M=1000, K=5, we can see some real differences. The best performing algorithm was the cyclic linearization with 4 threads, and the worst one the cyclic linearization with 16 threads (interesting). Overall, there we can clearly see that the 2 and 4 threads are performing the best in each test, but the cyclic versus batch is inconsistent, and with linearization we obtain the most inconsistent result.

For the test N=10, M=10000, K=5, we also can see some interesting results. The linearization with batch and 8 threads performed the best, and the columns with cyclic and 16 threads performed by far the worst (something more predictable). Having only 10 lines with 10000 elements each, it would be easier for the OS to cache some of the pages with the elements, so the sequential algorithm also performs pretty incredible, and we can also see that the lines algorithm is overall faster than the columns one, that being because each thread would cache most of the elements.

For the test N=10000, M=10, K=5, we obtained one of the fastest bests, the Sequential one by a wide margin, and the worst one was the columns with batch one also by far. In this test there are many lines with 10 elements, so I expected to run slowly for the sequential one, but maybe most of the elements were on the same page.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Java* |  |  |  |  |
| *Inputs* | *Algorithm* | *Parallelism Type* | *Threads* | *Execution time* |
| N = 10 | **Sequential** | **-** | **-** | **0.23415** |
| M = 10 | Lines | Batch | 4 | 3.58304 |
| K = 3 | Lines | Cyclic | 4 | 3.53202 |
|  | Columns | Batch | 4 | 3.69763 |
|  | Columns | Cyclic | 4 | 3.72306 |
|  | **Linearization** | **Batch** | **4** | **4.39763** |
|  | Linearization | Cyclic | 4 | 3.85835 |
| N = 1000 | Sequential | - | - | 70.41027 |
| M = 1000 | Lines | Batch | 2 | 79.12809 |
| K = 5 | Lines | Batch | 4 | 65.14881 |
|  | Lines | Batch | 8 | 113.79845 |
|  | Lines | Batch | 16 | 130.64853 |
|  | Lines | Cyclic | 2 | 79.26721 |
|  | Lines | Cyclic | 4 | 68.19541 |
|  | Lines | Cyclic | 8 | 111.76545 |
|  | Lines | Cyclic | 16 | 157.35632 |
|  | Columns | Batch | 2 | 90.91975 |
|  | Columns | Batch | 4 | 87.06887 |
|  | Columns | Batch | 8 | 115.14193 |
|  | Columns | Batch | 16 | 123.42222 |
|  | Columns | Cyclic | 2 | 85.74128 |
|  | Columns | Cyclic | 4 | 79.1811 |
|  | Columns | Cyclic | 8 | 120.44972 |
|  | Columns | Cyclic | 16 | 153.39926 |
|  | Linearization | Batch | 2 | 74.697 |
|  | Linearization | Batch | 4 | 69.13448 |
|  | Linearization | Batch | 8 | 88.06956 |
|  | Linearization | Batch | 16 | 161.66285 |
|  | Linearization | Cyclic | 2 | 63.99735 |
|  | **Linearization** | **Cyclic** | **4** | **57.52431** |
|  | Linearization | Cyclic | 8 | 97.66235 |
|  | **Linearization** | **Cyclic** | **16** | **201.49973** |
| N = 10 | Sequential | - | - | 50.92631 |
| M = 10000 | Lines | Batch | 2 | 58.82203 |
| K = 5 | Lines | Batch | 4 | 64.13767 |
|  | Lines | Batch | 8 | 67.39278 |
|  | Lines | Batch | 16 | 71.12815 |
|  | Lines | Cyclic | 2 | 66.29764 |
|  | Lines | Cyclic | 4 | 69.18791 |
|  | Lines | Cyclic | 8 | 74.52551 |
|  | Lines | Cyclic | 16 | 74.69774 |
|  | Columns | Batch | 2 | 60.9933 |
|  | Columns | Batch | 4 | 56.89766 |
|  | Columns | Batch | 8 | 65.46288 |
|  | Columns | Batch | 16 | 90.58032 |
|  | Columns | Cyclic | 2 | 54.65485 |
|  | Columns | Cyclic | 4 | 56.76268 |
|  | Columns | Cyclic | 8 | 58.29451 |
|  | **Columns** | **Cyclic** | **16** | **97.20276** |
|  | Linearization | Batch | 2 | 62.56913 |
|  | Linearization | Batch | 4 | 48.07492 |
|  | **Linearization** | **Batch** | **8** | **44.8279** |
|  | Linearization | Batch | 16 | 71.5996 |
|  | Linearization | Cyclic | 2 | 51.66897 |
|  | Linearization | Cyclic | 4 | 59.09501 |
|  | Linearization | Cyclic | 8 | 70.36774 |
|  | Linearization | Cyclic | 16 | 90.28678 |
| N = 10000 | **Sequential** | **-** | **-** | **20.62015** |
| M = 10 | Lines | Batch | 2 | 35.25913 |
| K = 5 | Lines | Batch | 4 | 31.05213 |
|  | Lines | Batch | 8 | 36.45505 |
|  | Lines | Batch | 16 | 40.95883 |
|  | Lines | Cyclic | 2 | 32.85674 |
|  | Lines | Cyclic | 4 | 37.84178 |
|  | Lines | Cyclic | 8 | 43.70375 |
|  | Lines | Cyclic | 16 | 45.85408 |
|  | **Columns** | **Batch** | **2** | **77.73998** |
|  | Columns | Batch | 4 | 54.07105 |
|  | Columns | Batch | 8 | 61.26326 |
|  | Columns | Batch | 16 | 63.39962 |
|  | Columns | Cyclic | 2 | 73.64938 |
|  | Columns | Cyclic | 4 | 68.89919 |
|  | Columns | Cyclic | 8 | 50.99933 |
|  | Columns | Cyclic | 16 | 51.69331 |
|  | Linearization | Batch | 2 | 30.57736 |
|  | Linearization | Batch | 4 | 35.27326 |
|  | Linearization | Batch | 8 | 39.24612 |
|  | Linearization | Batch | 16 | 34.56968 |
|  | Linearization | Cyclic | 2 | 34.48712 |
|  | Linearization | Cyclic | 4 | 42.39018 |
|  | Linearization | Cyclic | 8 | 54.73176 |
|  | Linearization | Cyclic | 16 | 65.95068 |

**2. C++ (Static)**

Unlike Java, there the results seemed to be more consistent (and they are a lot greater too, which proves C++ is better than Java :D). Similarly, from the first task we can see much besides the fact that the sequential one is a lot faster because of the same reason, the program trying to allocate some threads from the SO taking some time.

We can see, compared to Java, that the sequential one is one of the worst ones in all the rest of the tests (probably Java’s compiler do some optimizations related to this, achieving some great results). But one approach that is standing out from the rest is the linearization one, having the best results in almost all the cases.

We also can see that the lines distribution tends to do better when there are more lines, while the columns distribution tends to do better when there are more columns, this is probably because some threads will be assigned to do more tasks than the others, a line / a column in plus being a lot of elements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *C++ (Static)* |  |  |  |  |
| *Inputs* | *Algorithm* | *Parallelism Type* | *Threads* | *Execution time* |
| N = 10 | **Sequential** | **-** | **-** | **0.01805** |
| M = 10 | Lines | Batch | 4 | 1.0340125 |
| K = 3 | Lines | Cyclic | 4 | 1.02695 |
|  | **Columns** | **Batch** | **4** | **1.0674375** |
|  | Columns | Cyclic | 4 | 0.92425 |
|  | Linearization | Batch | 4 | 1.0553625 |
|  | Linearization | Cyclic | 4 | 1.06485 |
| N = 1000 | **Sequential** | **-** | **-** | **198.202125** |
| M = 1000 | Lines | Batch | 2 | 109.3906875 |
| K = 5 | Lines | Batch | 4 | 68.9315125 |
|  | Lines | Batch | 8 | 56.6327375 |
|  | Lines | Batch | 16 | 65.8404 |
|  | Lines | Cyclic | 2 | 112.7637875 |
|  | Lines | Cyclic | 4 | 71.5920625 |
|  | Lines | Cyclic | 8 | 79.308525 |
|  | Lines | Cyclic | 16 | 52.3308875 |
|  | Columns | Batch | 2 | 130.77375 |
|  | Columns | Batch | 4 | 75.2793625 |
|  | Columns | Batch | 8 | 70.0166 |
|  | Columns | Batch | 16 | 73.5599625 |
|  | Columns | Cyclic | 2 | 132.1035 |
|  | Columns | Cyclic | 4 | 94.867125 |
|  | Columns | Cyclic | 8 | 59.2690375 |
|  | Columns | Cyclic | 16 | 68.0560875 |
|  | Linearization | Batch | 2 | 109.216625 |
|  | Linearization | Batch | 4 | 72.6709 |
|  | **Linearization** | **Batch** | **8** | **46.6711625** |
|  | Linearization | Batch | 16 | 51.10605 |
|  | Linearization | Cyclic | 2 | 107.430375 |
|  | Linearization | Cyclic | 4 | 69.480325 |
|  | Linearization | Cyclic | 8 | 49.57155 |
|  | Linearization | Cyclic | 16 | 64.132375 |
| N = 10 | **Sequential** | **-** | **-** | **19.050725** |
| M = 10000 | Lines | Batch | 2 | 12.7392125 |
| K = 5 | Lines | Batch | 4 | 8.6864 |
|  | Lines | Batch | 8 | 7.7088 |
|  | Lines | Batch | 16 | 7.3035875 |
|  | Lines | Cyclic | 2 | 11.58555 |
|  | Lines | Cyclic | 4 | 9.2566 |
|  | Lines | Cyclic | 8 | 8.037325 |
|  | Lines | Cyclic | 16 | 8.3503625 |
|  | Columns | Batch | 2 | 11.4257125 |
|  | Columns | Batch | 4 | 8.8830625 |
|  | Columns | Batch | 8 | 6.630175 |
|  | Columns | Batch | 16 | 7.3091375 |
|  | Columns | Cyclic | 2 | 11.153325 |
|  | Columns | Cyclic | 4 | 8.8784 |
|  | Columns | Cyclic | 8 | 6.3754625 |
|  | Columns | Cyclic | 16 | 7.1200875 |
|  | Linearization | Batch | 2 | 11.867825 |
|  | Linearization | Batch | 4 | 8.2174375 |
|  | Linearization | Batch | 8 | 6.5206875 |
|  | Linearization | Batch | 16 | 6.8667875 |
|  | Linearization | Cyclic | 2 | 11.0994 |
|  | Linearization | Cyclic | 4 | 9.05045 |
|  | Linearization | Cyclic | 8 | 7.887525 |
|  | **Linearization** | **Cyclic** | **16** | **6.166875** |
| N = 10000 | Sequential | - | - | 29.8862 |
| M = 10 | Lines | Batch | 2 | 19.3439375 |
| K = 5 | **Lines** | **Batch** | **4** | **12.0207625** |
|  | Lines | Batch | 8 | 15.19865 |
|  | Lines | Batch | 16 | 13.8656125 |
|  | Lines | Cyclic | 2 | 19.374925 |
|  | Lines | Cyclic | 4 | 12.4437625 |
|  | Lines | Cyclic | 8 | 16.1040125 |
|  | Lines | Cyclic | 16 | 15.709125 |
|  | **Columns** | **Batch** | **2** | **36.3432875** |
|  | Columns | Batch | 4 | 30.9856375 |
|  | Columns | Batch | 8 | 27.4364375 |
|  | Columns | Batch | 16 | 26.8299375 |
|  | Columns | Cyclic | 2 | 37.252425 |
|  | Columns | Cyclic | 4 | 27.9929125 |
|  | Columns | Cyclic | 8 | 27.5198875 |
|  | Columns | Cyclic | 16 | 25.8718875 |
|  | Linearization | Batch | 2 | 19.677675 |
|  | Linearization | Batch | 4 | 13.5802 |
|  | Linearization | Batch | 8 | 13.2689125 |
|  | Linearization | Batch | 16 | 17.0181125 |
|  | Linearization | Cyclic | 2 | 26.5246 |
|  | Linearization | Cyclic | 4 | 23.562975 |
|  | Linearization | Cyclic | 8 | 25.035075 |
|  | Linearization | Cyclic | 16 | 25.3398875 |

**3. C++ (Dynamic)**

If I stated previously that C++ with static allocation was a lot better than Java, C++ with dynamic allocation is destroying both. For this test, the times were a lot smaller than previously, and the differences were almost insignificantly, that’s maybe because most of the memory is in the heap, and the threads shouldn’t require finding the data in the cache to perform the operations faster.

Unlike the other approaches, the sequential is more visibly worst than all the other ones, which makes sense because the advantage of the sequential one was the caching, but with the data being stored in the heap, that advantage is gone, because all the threads benefit from that now.

All the previous observations are still being seen in this case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *C++ (Dynamic)* |  |  |  |  |
| *Inputs* | *Algorithm* | *Parallelism Type* | *Threads* | *Execution time* |
| N = 10 | **Sequential** | **-** | **-** | **0.00821** |
| M = 10 | Lines | Batch | 4 | 0.92753 |
| K = 3 | Lines | Cyclic | 4 | 0.96783 |
|  | Columns | Batch | 4 | 0.90522 |
|  | **Columns** | **Cyclic** | **4** | **1.00051** |
|  | Linearization | Batch | 4 | 0.93217 |
|  | Linearization | Cyclic | 4 | 0.94243 |
| N = 1000 | **Sequential** | **-** | **-** | **204.0008** |
| M = 1000 | Lines | Batch | 2 | 118.2783 |
| K = 5 | Lines | Batch | 4 | 71.51566 |
|  | Lines | Batch | 8 | 71.54069 |
|  | Lines | Batch | 16 | 58.22028 |
|  | Lines | Cyclic | 2 | 116.2804 |
|  | Lines | Cyclic | 4 | 82.20552 |
|  | Lines | Cyclic | 8 | 74.1338 |
|  | Lines | Cyclic | 16 | 65.59637 |
|  | Columns | Batch | 2 | 121.2369 |
|  | Columns | Batch | 4 | 82.61576 |
|  | Columns | Batch | 8 | 67.01784 |
|  | **Columns** | **Batch** | **16** | **52.89504** |
|  | Columns | Cyclic | 2 | 130.7165 |
|  | Columns | Cyclic | 4 | 87.58329 |
|  | Columns | Cyclic | 8 | 83.51553 |
|  | Columns | Cyclic | 16 | 70.94138 |
|  | Linearization | Batch | 2 | 117.4614 |
|  | Linearization | Batch | 4 | 74.86469 |
|  | Linearization | Batch | 8 | 67.1994 |
|  | Linearization | Batch | 16 | 58.4499 |
|  | Linearization | Cyclic | 2 | 118.7508 |
|  | Linearization | Cyclic | 4 | 74.8271 |
|  | Linearization | Cyclic | 8 | 64.79547 |
|  | Linearization | Cyclic | 16 | 57.66596 |
| N = 10 | **Sequential** | **-** | **-** | **22.30387** |
| M = 10000 | Lines | Batch | 2 | 11.08354 |
| K = 5 | Lines | Batch | 4 | 9.85845 |
|  | Lines | Batch | 8 | 7.38372 |
|  | Lines | Batch | 16 | 7.56233 |
|  | Lines | Cyclic | 2 | 12.4654 |
|  | Lines | Cyclic | 4 | 9.02766 |
|  | Lines | Cyclic | 8 | 7.82771 |
|  | **Lines** | **Cyclic** | **16** | **6.89747** |
|  | Columns | Batch | 2 | 12.44401 |
|  | Columns | Batch | 4 | 9.43857 |
|  | Columns | Batch | 8 | 7.44861 |
|  | Columns | Batch | 16 | 7.54114 |
|  | Columns | Cyclic | 2 | 10.80383 |
|  | Columns | Cyclic | 4 | 9.02052 |
|  | Columns | Cyclic | 8 | 7.01599 |
|  | Columns | Cyclic | 16 | 7.47444 |
|  | Linearization | Batch | 2 | 12.16632 |
|  | Linearization | Batch | 4 | 9.14973 |
|  | Linearization | Batch | 8 | 7.05376 |
|  | Linearization | Batch | 16 | 9.37069 |
|  | Linearization | Cyclic | 2 | 13.30638 |
|  | Linearization | Cyclic | 4 | 9.20933 |
|  | Linearization | Cyclic | 8 | 7.39564 |
|  | Linearization | Cyclic | 16 | 7.45787 |
| N = 10000 | **Sequential** | **-** | **-** | **19.53629** |
| M = 10 | Lines | Batch | 2 | 13.29274 |
| K = 5 | Lines | Batch | 4 | 11.10984 |
|  | Lines | Batch | 8 | 9.64948 |
|  | Lines | Batch | 16 | 9.02486 |
|  | Lines | Cyclic | 2 | 13.00414 |
|  | Lines | Cyclic | 4 | 9.58121 |
|  | Lines | Cyclic | 8 | 10.72996 |
|  | Lines | Cyclic | 16 | 10.8164 |
|  | Columns | Batch | 2 | 13.56078 |
|  | Columns | Batch | 4 | 11.28136 |
|  | Columns | Batch | 8 | 9.95839 |
|  | Columns | Batch | 16 | 9.04612 |
|  | Columns | Cyclic | 2 | 13.25753 |
|  | Columns | Cyclic | 4 | 9.62427 |
|  | Columns | Cyclic | 8 | 9.5032 |
|  | Columns | Cyclic | 16 | 10.91284 |
|  | Linearization | Batch | 2 | 11.9323 |
|  | Linearization | Batch | 4 | 10.77132 |
|  | **Linearization** | **Batch** | **8** | **7.18736** |
|  | Linearization | Batch | 16 | 8.48294 |
|  | Linearization | Cyclic | 2 | 13.1151 |
|  | Linearization | Cyclic | 4 | 11.20009 |
|  | Linearization | Cyclic | 8 | 9.48705 |
|  | Linearization | Cyclic | 16 | 9.6387 |

**4. Approaches**

A screenshot of a computer program

Description automatically generated

Basically we just need to apply the convolution operation for each element and put the results in a new matrix for each one.

A grid with numbers and symbols

Description automatically generated

For the corner elements, we need to take the closest element in the matrix, this could be done virtually by clipping the row index in the interval [0, rows] and the column index in the interval [0, columns].

The convolution operation should be done like this for the element (i, j):

*sum = 0  
for (int p = 0; p < K; p++):  
 for (int q = 0; q < K; q++):  
 sum += elemInMatrix(i – K / 2 + p, j – K / 2 + q) \* elemInKernel(p, q)  
return sum*

**4.0. Parallelization types**

**4.0.1. Batch approach**

🡪 there we move for each contiguous block.

**4.0.2. Cyclic approach**

🡪 there we jump from elements to elements

**4.1. Sequential approach**

🡪 there we just apply the convolution operation for each element sequentially, no parallelism.

**4.2. Lines approach**

🡪 there we run a thread for some lines, when the distribution is not equal, some threads will take at most 1 line additionally, so it would be disadvantageous for matrices with a big number of lines, there we can run with the batch approach and with the cyclic one, some advantage would be that the threads can cache some lines and be faster.

🡪 for the batch approach, we divide the number of lines by P (the number of threads) and gave each thread N / P lines, +1 if there are lines remaining (we distribute N % P for the first lines), for the cyclic approach we jump from P to P lines

**4.3. Columns approach**

🡪 there we run a thread for some columns, we have the same disadvantage as previous, but for matrices with a big number of columns, there we can run with the batch approach and the cyclic one as well, this approach doesn’t benefit from the cache advantage because it would need to access a page for each element in the column.

🡪 for the batch approach, we divide similarly to the previous approach, but the columns each times, so M / P columns for each thread, +1 if there are columns remaining (we distribute M % P like the last times), for the cyclic approach we jump from P to P columns

**4.4. Linearization approach**

🡪 there we take a mapping function that linearizes the matrix ((i, j) 🡪 i \* M + j, x -> (x / M, x % M)), in advantage of the other approaches, some threads will take at most one element, because the space is contiguous, it would cache some pages with the elements, but only in the batch approach, in the cyclic approach should perform worse.

🡪 taking the mapping into account, for the batch approach we give each thread N \* M / P elements, +1 if there are elements remaining (we distribute N \* M % P for the first lines), for the cyclic approach we jump from P to P elements in the linearization

**4.5. Blocks approach**

🡪 similarly to the linearization approach, but we calculate the start index and the end index before the threads are running, so each thread shouldn’t perform many operations at each step, the advantage is very insignificant in face of the linearization approach.

🡪 similarly to the previous approach, we give each thread N \* M / P elements, calculating the start point (startI, startJ) and the end point (endI, endJ) and going through those elements.