**C++**

**Gestionarea Bordurii**

Pentru a aplica convoluția, matricea de intrare trebuie să fie extinsă cu o bordură (padding) pentru a permite kernel-ului să fie aplicat pe toate elementele matricei originale. Funcția padMatrix din clasa Utils adaugă această bordură:

*int padded\_m = original\_m + 2 \* padding;  
int padded\_n = original\_n + 2 \* padding;  
int \*\*padded\_original = new int \*[padded\_m];  
for (int i = 0; i < padded\_m; i++) {  
 padded\_original[i] = new int[padded\_n]();  
}  
// Copy the original matrix into the center of the padded matrix  
for (int i = 0; i < original\_m; i++) {  
 for (int j = 0; j < original\_n; j++) {  
 padded\_original[i + padding][j + padding] = original[i][j];  
 }  
}  
// Clean up the original matrix  
for (int i = 0; i < original\_m; i++) {  
 delete[] original[i];  
}  
delete[] original;  
// Update the original pointer to point to the padded matrix  
original = padded\_original;  
original\_m = padded\_m;  
original\_n = padded\_n;*

**Alocarea Rândurilor Thread-urilor**

Pentru a distribui sarcina între mai multe thread-uri, matricea de intrare este împărțită în rânduri, iar fiecare thread procesează un subset de rânduri. Numărul de rânduri alocate fiecărui thread este calculat astfel:

*int lines\_per\_thread = (original\_m - 2 \* padding) / no\_threads;  
int remainder\_line = (original\_m - 2 \* padding) % no\_threads;  
int line\_start = padding;  
for (int i = 0; i < no\_threads; i++) {  
 int line\_end = line\_start + lines\_per\_thread;  
 if (remainder\_line != 0) {  
 remainder\_line--;  
 line\_end++;  
 }*

Fiecare thread primește rowsPerThread rânduri, iar rândurile rămase (extraRows) sunt distribuite uniform între primele thread-uri.

**Utilizarea far amatrice auxiliara**

In the paralel\_h method, auxiliary vectors are used to handle boundary conditions and store intermediate results. Here is a brief explanation of how these vectors are used:

* last\_modified: This vector stores the values of the row above the current row being processed. It is initialized with zeros and updated as the processing moves to the next row.
* boundary\_down: This vector stores the values of the row below the current row being processed. It is also initialized with zeros and updated accordingly.
* curent: This vector stores the intermediate results for the current row being processed. It is initialized with zeros and updated with the convolution results.

*void work\_lines(int id, int line\_start, int line\_end, std::barrier<> &barrier) {  
 int boundary\_down[original\_n];  
 int last\_modified[original\_n], curent[original\_n];  
  
 for (int j = 0; j < original\_n; j++) {  
 last\_modified[j] = original[line\_start - 1][j];  
 boundary\_down[j] = original[line\_end][j];  
 }  
  
 // printf("Thread arrived: %d\n", id);  
 barrier.arrive\_and\_wait();  
 // printf("Thread released: %d\n", id);  
  
 for (int i = line\_start; i < line\_end; i++) {  
 for (int j = padding; j < original\_n - padding; j++) {  
 int temp = 0;  
  
 for (int k = 0; k < conv\_n; k++) {  
 for (int l = 0; l < conv\_n; l++) {  
 int ki = i + (k - conv\_n / 2);  
 int kj = j + (l - conv\_n / 2);  
  
 if (ki < line\_start) {  
 temp += last\_modified[kj] \* conv[k][l];  
 } else if (ki >= line\_end) {  
 temp += boundary\_down[kj] \* conv[k][l];  
 } else {  
 temp += original[ki][kj] \* conv[k][l];  
 }  
 }  
 }  
  
 curent[j] = temp;  
 }  
  
 if (i > line\_start) {  
 for (int j = padding; j < original\_n - padding; j++)  
 original[i - 1][j] = last\_modified[j];  
 }  
  
 for (int j = padding; j < original\_n - padding; j++)  
 last\_modified[j] = curent[j];  
  
 if (i == line\_end - 1) {  
 for (int j = padding; j < original\_n - padding; j++)  
 original[i][j] = curent[j];  
 }  
 }  
}*

**Utilizarea Barierei**

O barieră (std::barrier) este utilizată pentru a sincroniza thread-urile, asigurându-se că toate thread-urile au terminat procesarea înainte de a continua. Fiecare thread apelează barrier.arrive\_and\_wait() după ce își termină sarcina:

*barrier.arrive\_and\_wait();*

**Java**

**Gestionarea Bordurii**

Pentru a aplica convoluția, matricea de intrare trebuie să fie extinsă cu o bordură (padding) pentru a permite kernel-ului să fie aplicat pe toate elementele matricei originale. Funcția padMatrix din clasa Utils adaugă această bordură:

*int[][] paddedMatrix = new int[original\_m + 2 \* padding][original\_n + 2 \* padding];  
for (int i = 0; i < original\_m; i++) {  
 for (int j = 0; j < original\_n; j++) {  
 paddedMatrix[i + padding][j + padding] = original[i][j];  
 }}  
original = paddedMatrix;  
original\_m += 2 \* padding;  
original\_n += 2 \* padding;*

**Alocarea Rândurilor Thread-urilor**

Pentru a distribui sarcina între mai multe thread-uri, matricea de intrare este împărțită în rânduri, iar fiecare thread procesează un subset de rânduri. Numărul de rânduri alocate fiecărui thread este calculat astfel:

*int rowsPerThread = (n - 2) / numThreads;*

*int extraRows = (n - 2) % numThreads;*

Fiecare thread primește rowsPerThread rânduri, iar rândurile rămase (extraRows) sunt distribuite uniform între primele thread-uri.

**Utilizarea far amatrice auxiliara**

In the paralel\_h method, auxiliary vectors are used to handle boundary conditions and store intermediate results. Here is a brief explanation of how these vectors are used:

* last\_modified: This vector stores the values of the row above the current row being processed. It is initialized with zeros and updated as the processing moves to the next row.
* boundary\_down: This vector stores the values of the row below the current row being processed. It is also initialized with zeros and updated accordingly.
* curent: This vector stores the intermediate results for the current row being processed. It is initialized with zeros and updated with the convolution results.

*public static void paralel\_h() {  
 ArrayList<Thread> activeThreads = new ArrayList<>();  
 int lines\_per\_thread = (original\_m - 2 \* padding) / no\_threads;  
 int remainder\_line = (original\_m - 2 \* padding) % no\_threads;  
 int line\_start = padding;  
 //Alocarea randurilor pentru fiecare thread  
 for (int i = 0; i < no\_threads; i++) {  
  
 int line\_end = line\_start + lines\_per\_thread;  
  
 if (remainder\_line != 0) {  
 remainder\_line--;  
 line\_end++;  
 }  
 if (line\_end > original\_m - padding)  
 line\_end = original\_m - padding;  
  
 if (line\_start >= original\_m - padding)  
 break;  
 //Pornirea thread-urilor  
 WorkLines work\_lines = new WorkLines(original, conv, line\_start, line\_end, conv\_n, original\_n, padding);  
 work\_lines.start();  
 activeThreads.add(work\_lines);  
  
 line\_start = line\_end;  
 }  
 for (Thread thread : activeThreads) {  
 try {  
 thread.join();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e); }}}*

**Utilizarea Barierei**

O barieră (CyclicBarrier) este utilizată pentru a sincroniza thread-urile, asigurându-se că toate thread-urile au terminat procesarea înainte de a continua. Fiecare thread apelează barrier.await() după ce își termină sarcina:

*barrier.await();*