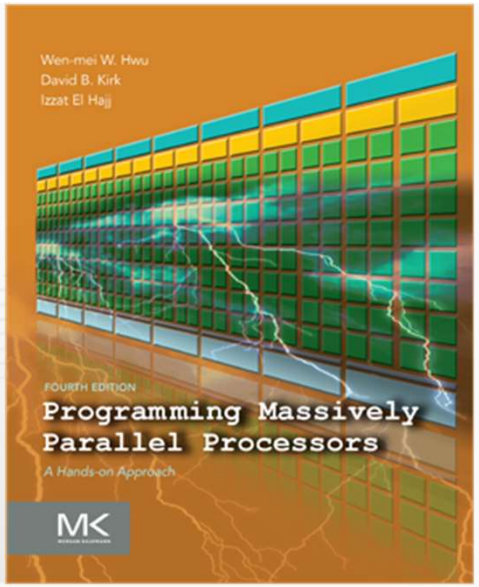
MORGAN KAUFMANN

Programming Massively Parallel Processors

A Hands-on Approach

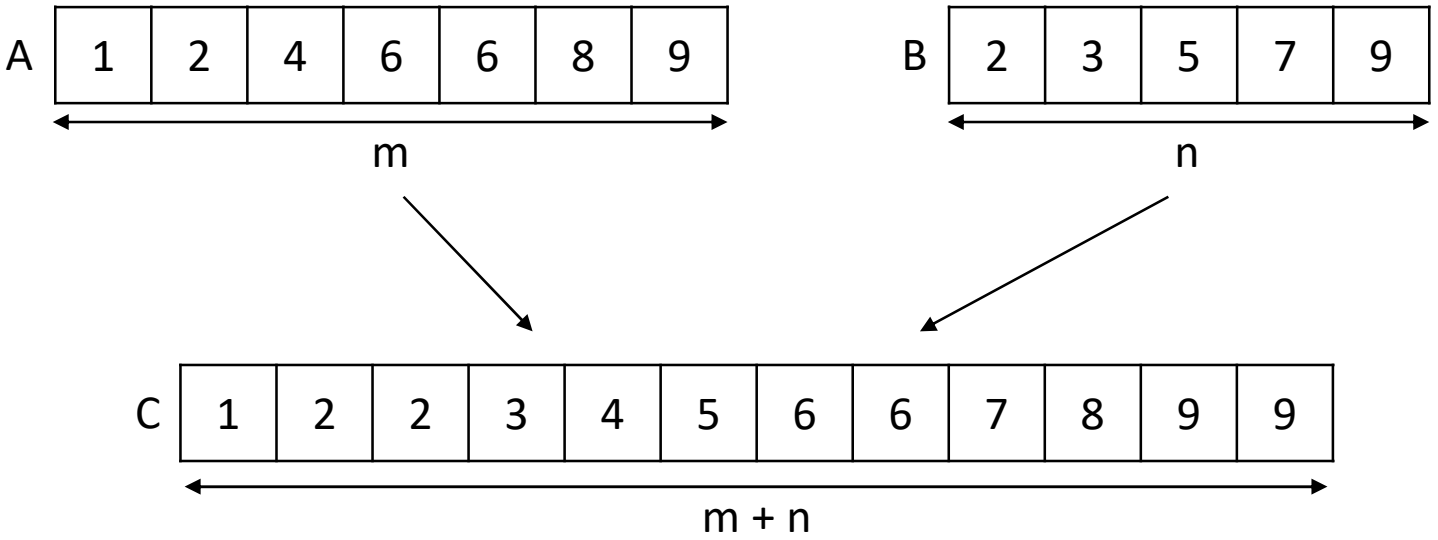
CHAPTER 12 › Merge



Merge

- An **ordered merge** operations takes two ordered lists and combines them into a single ordered list

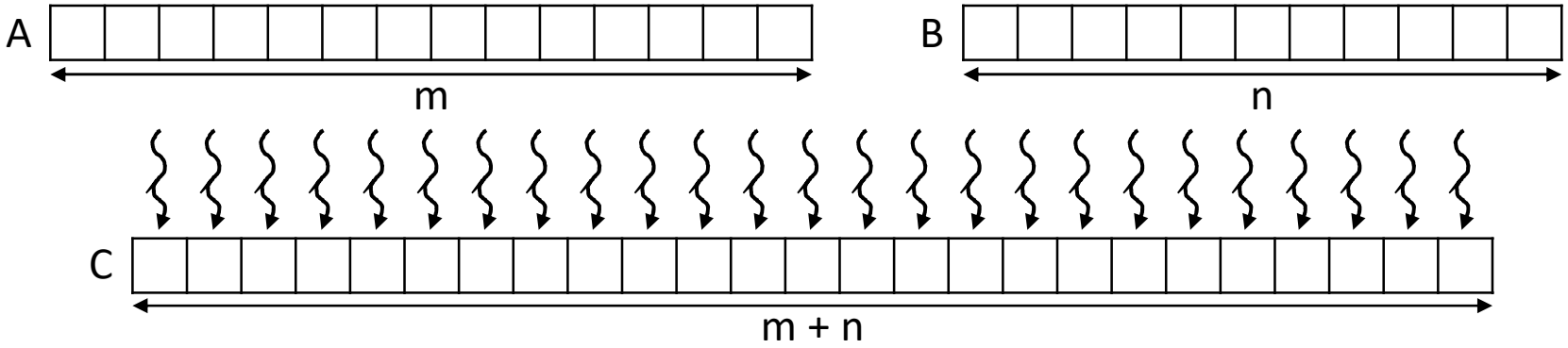
- Example:



Sequential Merge Code

```
void merge(float* A, float* B, float* C, unsigned int m, unsigned int n) {
    unsigned int i = 0;
    unsigned int j = 0;
    for(unsigned int k = 0; k < m + n; ++k) {
        if(j == n || i < m && A[i] <= B[j]) {
            C[k] = A[i++];
        } else {
            C[k] = B[j++];
        }
    }
}
```

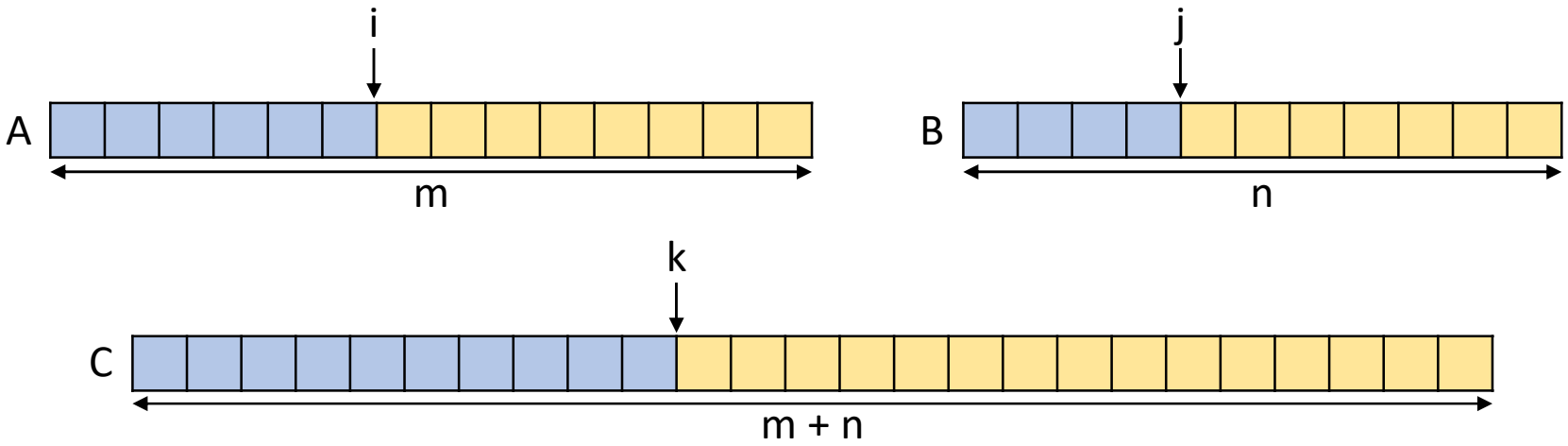
Parallel Merge



Parallelization approach: Assign a thread to each output element and have it fetch the corresponding input element from A or B

Key Challenge: How does each thread find its input element?

Finding Co-ranks



Objective: Given k, find i and j

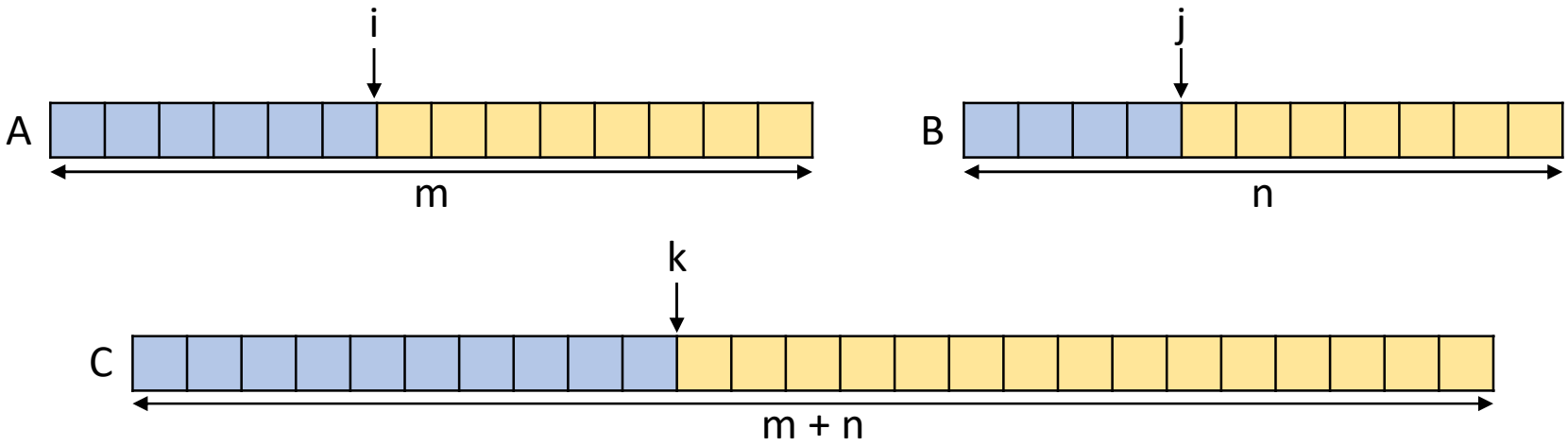
- We refer to i and j as the **co-ranks** of k

We observe that:

$$k = i + j$$

Therefore, it is sufficient to find i, then use j = k - i.

Finding Co-ranks



Objective: Given k, find i **To find j:** j = k - i

Let's set a bounds on i. We observe that:

$0 \le i \le m$ and $0 \le j \le n$

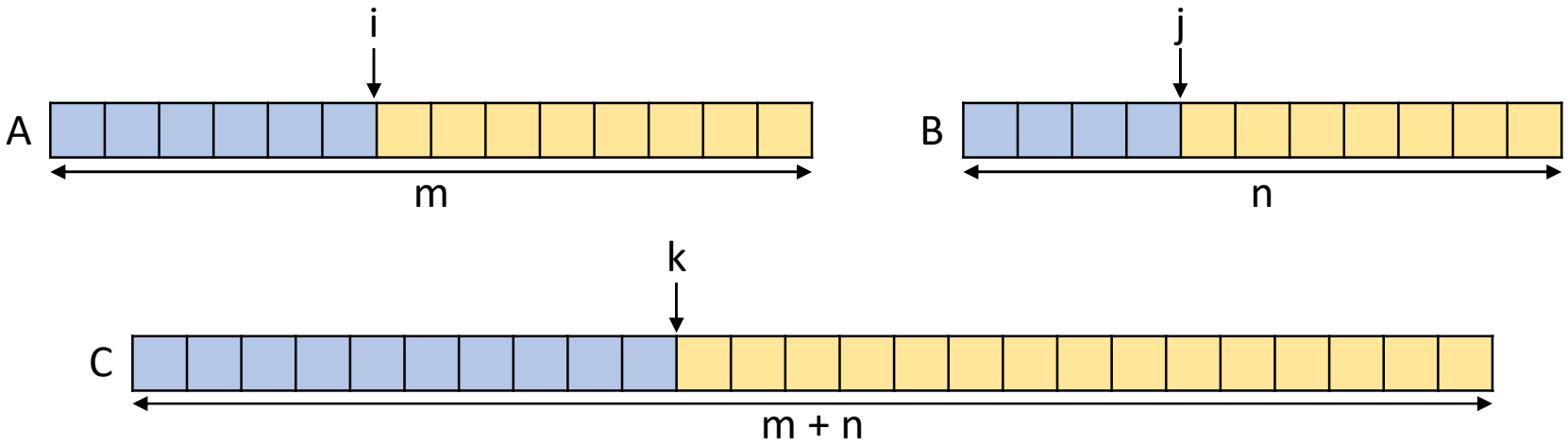
$0 \le k - i \le n$

$-k \le -i \le -k + n$

$k - n \le i \le k$

Therefore: $\max(0, k - n) \le i \le \min(m, k)$

Finding Co-ranks



Objective: Given k, find i **To find j:** j = k - i

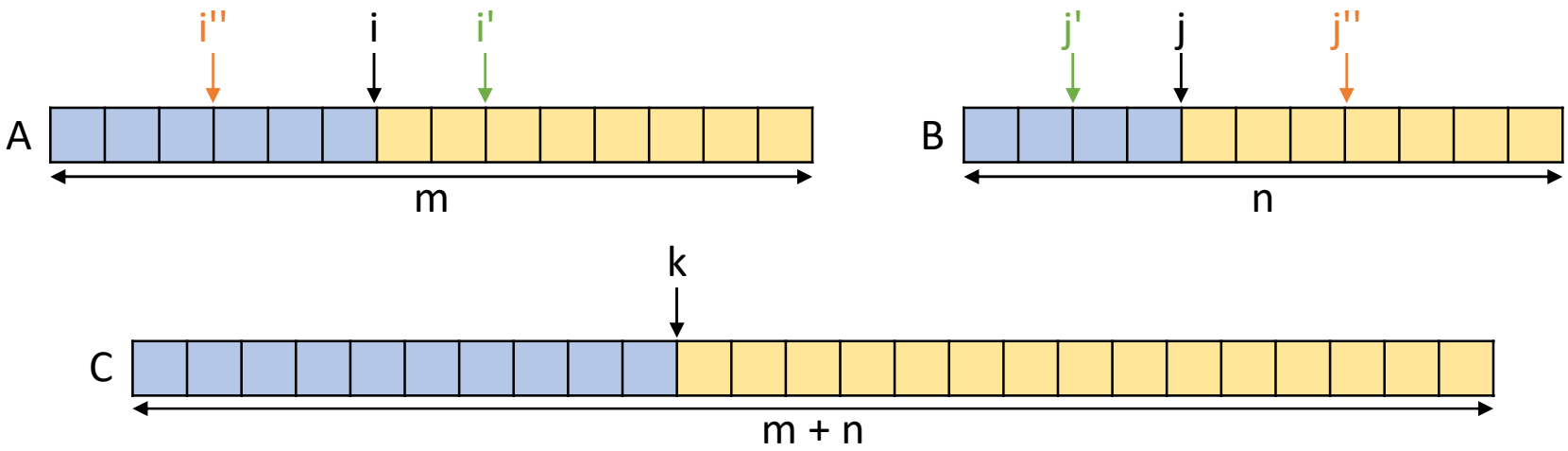
Bound on i: $\max(0, k - n) \leq i \leq \min(m, k)$

Strategy: Perform a binary search within the bound

How do we know when we found i? We observe that:

$A[i - 1] \leq B[j]$ and $B[j - 1] < A[i]$

Finding Co-ranks



Objective: Given k, find i **To find j:** j = k - i

Bound on i: $\max(0, k - n) \leq i \leq \min(m, k)$

Strategy: Perform a binary search within the bound

- Guess is correct: $A[i - 1] \leq B[j]$ && $B[j - 1] < A[i]$
- Guess is **too high**: $A[i' - 1] > B[j']$
- Guess is **too low**: $B[j'' - 1] \geq A[i'']$

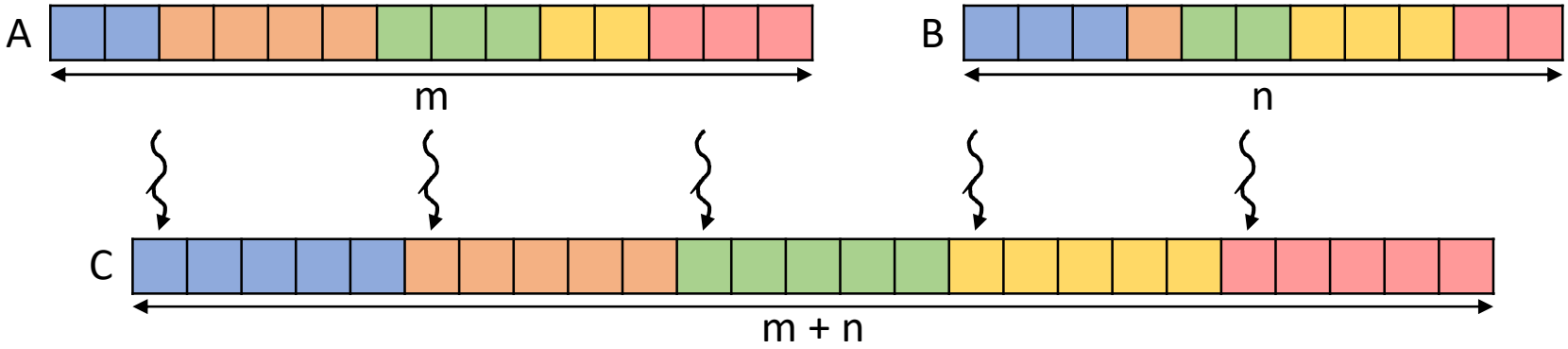
Parallel Merge Code

```
__device__ unsigned int coRank(float* A, float* B, unsigned int m, unsigned int n, unsigned int k) {

    // Initialize bounds
    unsigned int iLow = (k > n)?(k - n):0;
    unsigned int iHigh = (k < m)?k:m;

    // Binary search
    while(true) {
        unsigned int i = (iLow + iHigh)/2;
        unsigned int j = k - i;
        if(i > 0 && j < n && A[i - 1] > B[j]) {
            iHigh = i - 1;
        } else if(j > 0 && i < m && B[j - 1] >= A[i]) {
            iLow = i + 1;
        } else {
            return i;
        }
    }

}
__global__ void merge_kernel(float* A, float* B, float* C, unsigned int m, unsigned int n) {
    unsigned int k = blockIdx.x*blockDim.x + threadIdx.x;
    if(k < m + n) {
        unsigned int i = coRank(A, B, m, n, k);
        unsigned int j = k - i;
        if(j == n || i < m && A[i] <= B[j]) {
            C[k] = A[i];
        } else {
            C[k] = B[j];
        }
    }
}
```

Work Efficiency

- Sequential merge performs *O(N)* operations
- Parallel merge performs *O(N log(N))* operations
  - Launches *N* threads
  - Each thread performs a binary search which is *O(log(N))*
- Use thread coarsening to improve work efficiency

Applying Thread Coarsening



Assign threads to fixed-length output segments

Each thread calls the coRank function to find the bounds of its input segments

Each thread then performs a sequential merge of its input segments

Applying Thread Coarsening Code

```
__device__ void mergeSequential(float* A, float* B, float* C, unsigned int m, unsigned int n) {
    unsigned int i = 0;
    unsigned int j = 0;
    for(unsigned int k = 0; k < m + n; ++k) {
        if(j == n || i < m && A[i] <= B[j]) {
            C[k] = A[i++];
        } else {
            C[k] = B[j++];
        }
    }
}

__global__ void merge_kernel(float* A, float* B, float* C, unsigned int m, unsigned int n) {
    unsigned int k = (blockIdx.x*blockDim.x + threadIdx.x)*COARSE_FACTOR;
    if(k < m + n) {
        unsigned int i = coRank(A, B, m, n, k);
        unsigned int j = k - i;
        unsigned int kNext = (k + COARSE_FACTOR < m + n)?(k + COARSE_FACTOR):(m + n);
        unsigned int iNext = coRank(A, B, m, n, kNext);
        unsigned int jNext = kNext - iNext;
        mergeSequential(&A[i], &B[j], &C[k], iNext - i, jNext - j);
    }
}
```

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.