Chapter-11
Parallel patterns: merge sort
An introduction to tiling with dynamic input data identification

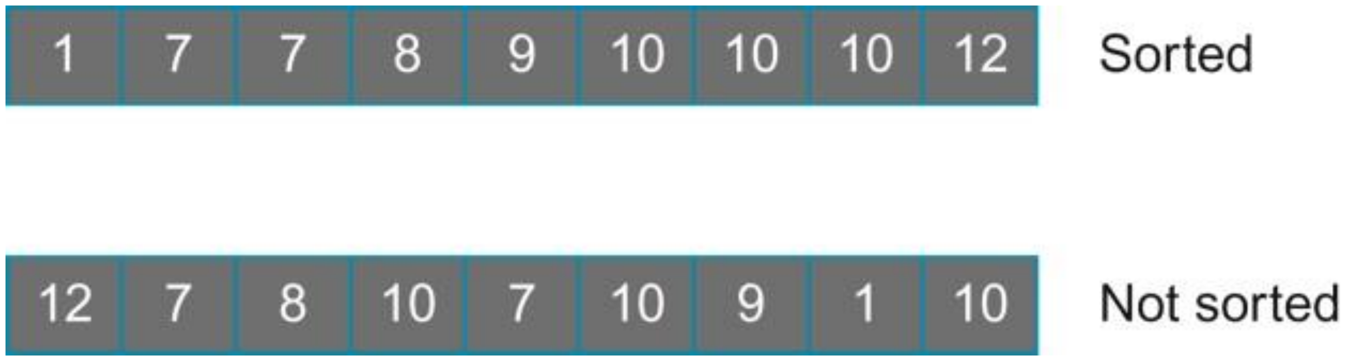

FIGURE 11.1: Examples of sorted versus unsorted lists.

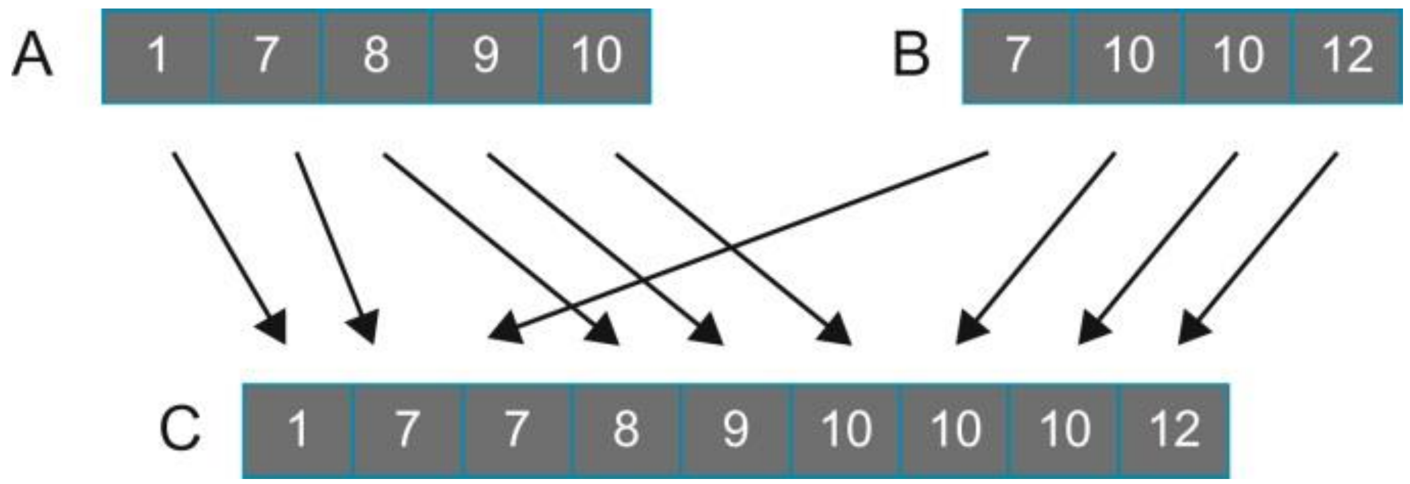

FIGURE 11.2: Example of a merge operation.

```
void merge_sequential(int *A, int m, int *B,
                      int n, int *C) {
  int i = 0; //index into A
  int j = 0; //index into B
  int k = 0; //index into C

              // handle the start of A[] and B[]
  while ((i < m) && (j < n)) {
    if (A[i] <= B[j]) {
      C[k++] = A[i++];
    } else {
      C[k++] = B[j++];
    }
  }
  if (i == m) {
        //done with A[] handle remaining B[]
    for (; j < n; j++) {
      C[k++] = B[j];
    }
  } else {
        //done with B[], handle remaining A[]
    for (; i <m; i++) {
      C[k++] = A[i];
    }
  }
}
```

FIGURE 11.3: A sequential merge function.

Parallel Merge Sort

- parallelization requires each thread to dynamically identify its input position ranges.
- Input ranges are data dependent
- which takes two ordered lists and generates a combined, ordered sort.

Parallel Merge Sort

- A parallel merge sort function divides up the input list into multiple sections and distributes them to parallel threads.
- The threads sort the individual section(s) and then cooperatively merge the sorted sections.

Parallel Mergesort

- each thread first calculates the range of output positions (output range) that it is going to produce, and uses that range as the input to a co-rank function to identify the corresponding input ranges that will be merged to produce the output range.
- Once the input and output ranges are determined, each thread can independently access its two input subarrays and one output subarray.
- Such independence allows each thread to perform the sequential merge function on their subarrays to do the merge in parallel.

CO-RANK

- Let A and B be two sorted input arrays with m and n elements respectively.
- Observation 1: For any k such that $0 \le k < m+n$, there is either (case 1) an i such that $0 \le i < m$ and C[k] receives its value from A[i], or (case 2) a j such that $0 \le j < n$ and C[k] receives its value from B[j] in the merge process.

CO-RANK

- In the first case, we find i and derive j as k-i. In the second case, we find j and derive i as k-j. We can take advantage of the symmetry and summarize the two cases into one observation:

CO-RANK

- Observation 2: For any k such that $0 \le k < m+n$, we can find i and j such that $k=i+j$, $0 \le i < m$ and $0 \le j < n$ and the subarray C[0]-C[k-1] is the result of merging subarray A[0]-A[i-1] and subarray B[0]-B[j-1].
- the index k is referred to as its rank. The unique indices i and j are referred to as its co-ranks.

CO-RANK

- We can divide the work among threads by dividing the output array into subarrays and assign the generation of one subarray to each thread. Once the assignment is done, the rank of output elements to be generated by each thread is known.
- Each thread then uses the co-rank function to determine the subarrays of the input arrays that it needs to merge into its output subarray.

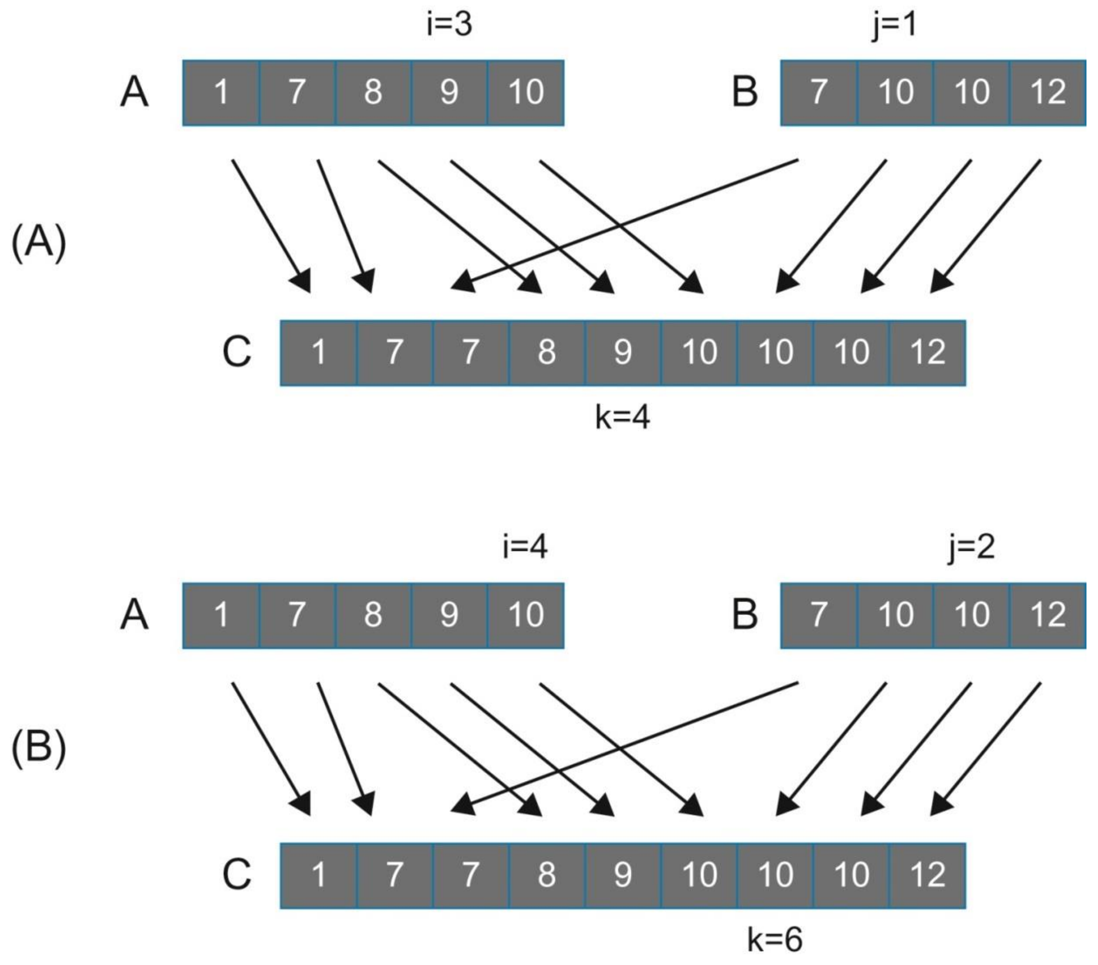

FIGURE 11.4: Examples of observation (1). (A) shows case 1 and (B) shows case 2.

CO-RANK

- We define the co-rank function as a function that takes the rank (k) of a C array element and information about the two input arrays and returns the i co-rank value: int co_rank(int k, int * A, int m, int * B, int n)
- The caller can then derive the j co-rank value as k-i.

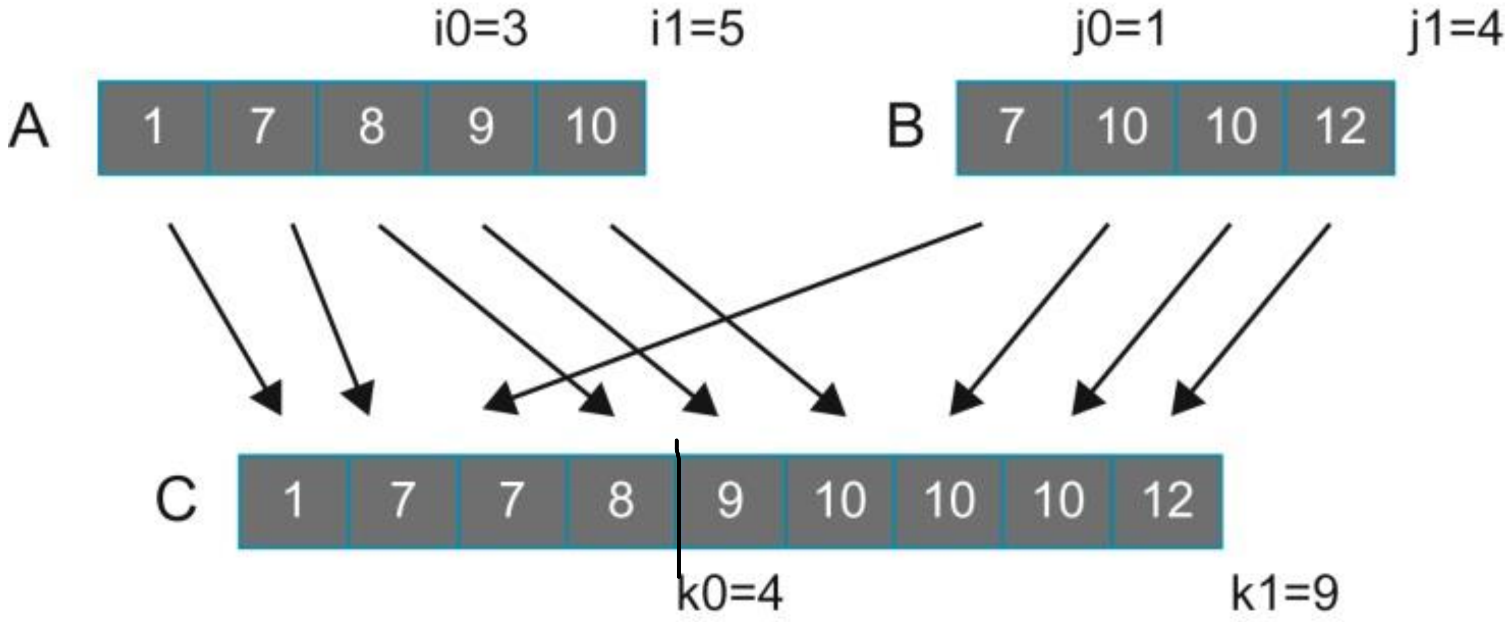

FIGURE 11.5: Example of co-rank function execution.

CO-RANK

- Thread-0 generates C[0]-C[3] and Thread-1 generates C[4]-C[8].
- Thread 0 calls the co-rank function with parameters (4, A, 5, B, 4). The goal of the co-rank function for thread 0 is to identify the co-rank values i0 =3 and j0 =1.

CO-RANK

- That is, the prefix subarray of C[4] is to be generated by merging the prefix subarrays of A[3] (A[0]-A[2]) and B[1] (B[0]). Intuitively, we are looking for a total of 4 elements from A and B that will fill the first 4 elements of the output array. By visual inspection, we see that the choice of i0 =3 and j0 =1 meets our need.

CO-RANK

- Thread 1 calls the co-rank function with parameters (9, A, 5, B, 4).
- the co-rank function should produce co-rank values i1 =5 and j1 =4.
- Note that the input subarrays to be used by thread 1 are actually defined by the co-rank values of thread 0 and those of thread 1: A[3]-A[4] and B[1]-B[3].
- That is, the starting index of the A subarray for thread 1 is actually thread 0's co-rank i value. The starting index of the B subarray for thread 1 is thread 0's co-rank j value.

CO-RANK

- The co-rank function uses two pairs of marker variables to delineate the range of A array indices and the range of B array indices being considered for the co-rank values. Variables i_low and j_low are the smallest possible co-rank values that could be generated by the function.

CO-RANK

- Variables i and j are the candidate co-rank return values being considered in the current iteration.

```
int co_rank(int k, int* A, int m, int* B, int n) {
  int i= k<m ? k : m;  //i = min(k,m)
  int j = k- i;
  int i_low = 0>(k-n) ? 0 : k-n;  //i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0 : k-m;  //i_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active)    {
     if (i > 0 && j < n && A[i-1] > B[j]) {
        delta = ((i - i_low +1) >> 1);  // ceil(i-i_low)/2)
        j_low = j;
        j = j + delta;
        i = i - delta;
     } else if (j > 0 && i < m && B[j-1] >= A[i]) {
        delta = ((j - j_low +1) >> 1);
        i_low = i;
        i = i + delta;
        j = j - delta;
     } else {
        active = false;
     }
  }
  return i;
}
```

FIGURE 11.6: A co-rank function based on binary search.

CO-RANK

- Line 2 initializes i to its largest possible value. If the k value is greater than m, line 2 initializes i to m, since the co-rank i value cannot be larger than the size of the A array. Otherwise, line 2 initializes i to k, since i cannot be larger than k.
- The co-rank j value is initialized as k-i (line 3).

CO-RANK

- Throughout the execution, the co-rank function maintains this important invariant relation. The sum of the i and j variable is always equal to the value of the input variable k (the rank value).
- when k is larger than n, we know that the i value cannot be less than k-n. The reason is that the most number of C[k] prefix subarray elements that can come from the B array is n.

CO-RANK

- Therefore, a minimal of k −n elements must come from A. Therefore, the i value can never be smaller than k −n; we may as well set i_low to k −n.

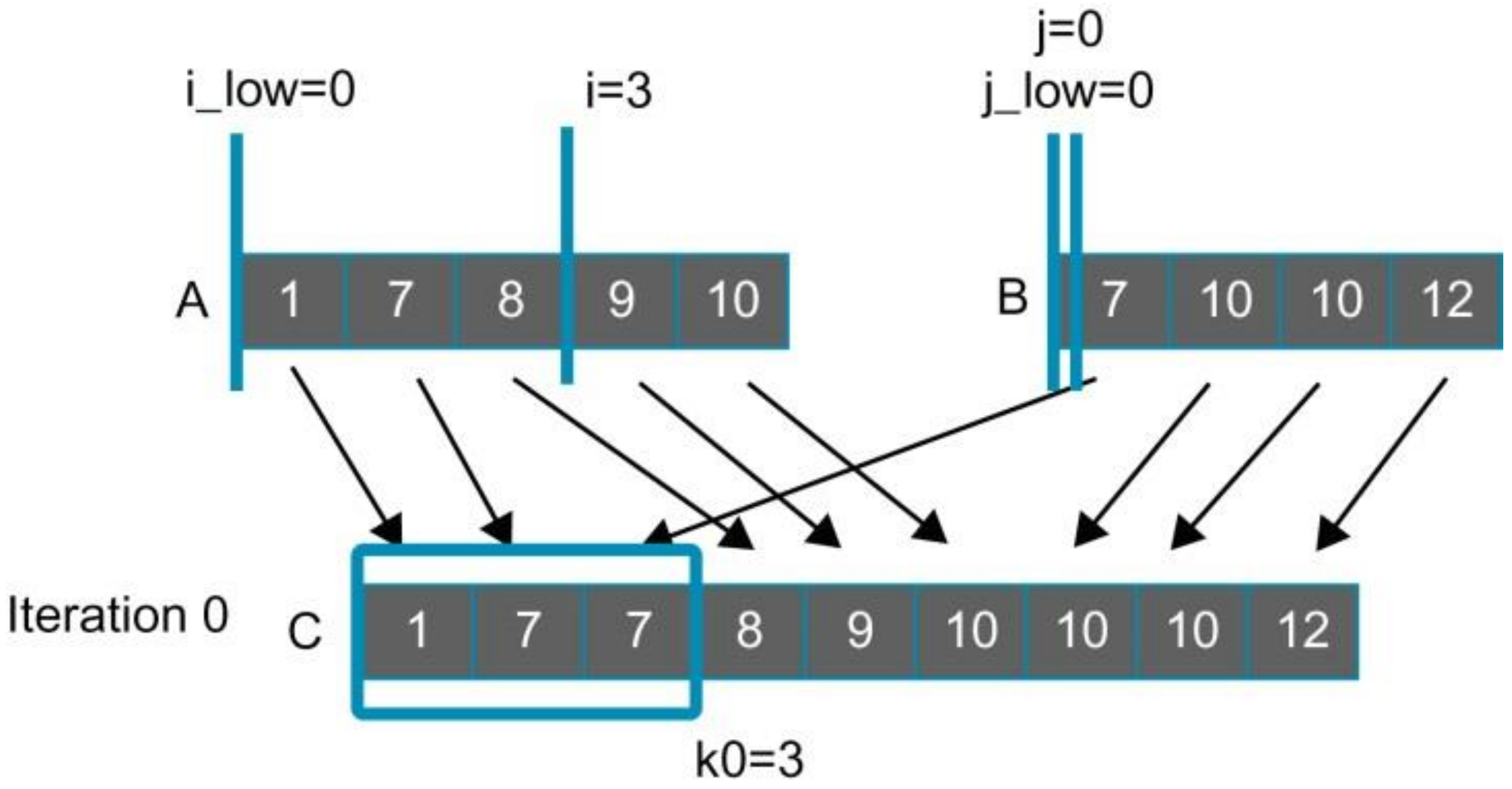

FIGURE 11.7: Iteration 0 of the co-rank function operation example for thread 0.

CO-RANK

- thread 0 calls the co-rank function with parameters (3, A, 5, B, 4).
- The main body of the co-rank function is a while-loop (line 8) that iteratively zooms into the final co-rank i and j values. The goal is to find a pair of i and j that result in $A[i-1] \le B[j]$ and $B[j-1] < A[i]$.

CO-RANK

- largest A element in the current subarray could be equal to the smallest element in the next B subarray since the A elements take precedence in placement into the output array whenever a tie occurs between an A element and a B element in our definition of the merge process.

CO-RANK

- In Fig. 11.6, the first if-construction in the while-loop (line 9) tests if the current i value is too high. If so, it will adjust the marker values so that it reduces the search range for i by about half toward the smaller end. This is done by reducing the i value by about half the difference between i and i_low.

CO-RANK

- In Fig. 11.7, for iteration 0 of the while-loop, the if-construct finds that the i value (3) is too high since A[i −1], whose value is 8, is greater than B[j], whose value is 7. The next few statements proceed to reduce the search range for i by reducing its value by delta =(3-0+1)>>1 =2 (lines 10 and 13) while keeping the i_low value unchanged. Therefore, the i_low and i values for the next iteration will be 0 and 1.

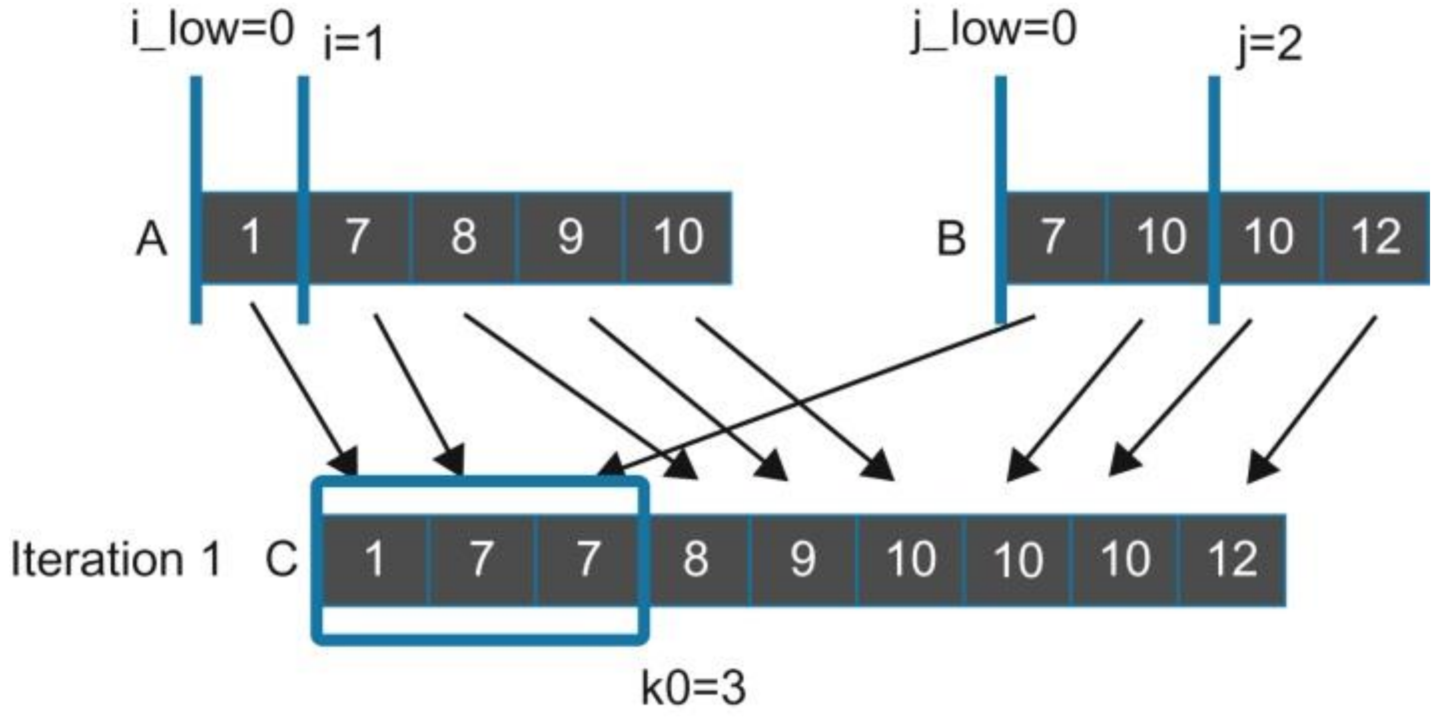

FIGURE 11.8: Iteration 1 of the co-rank function operation example for thread 0.

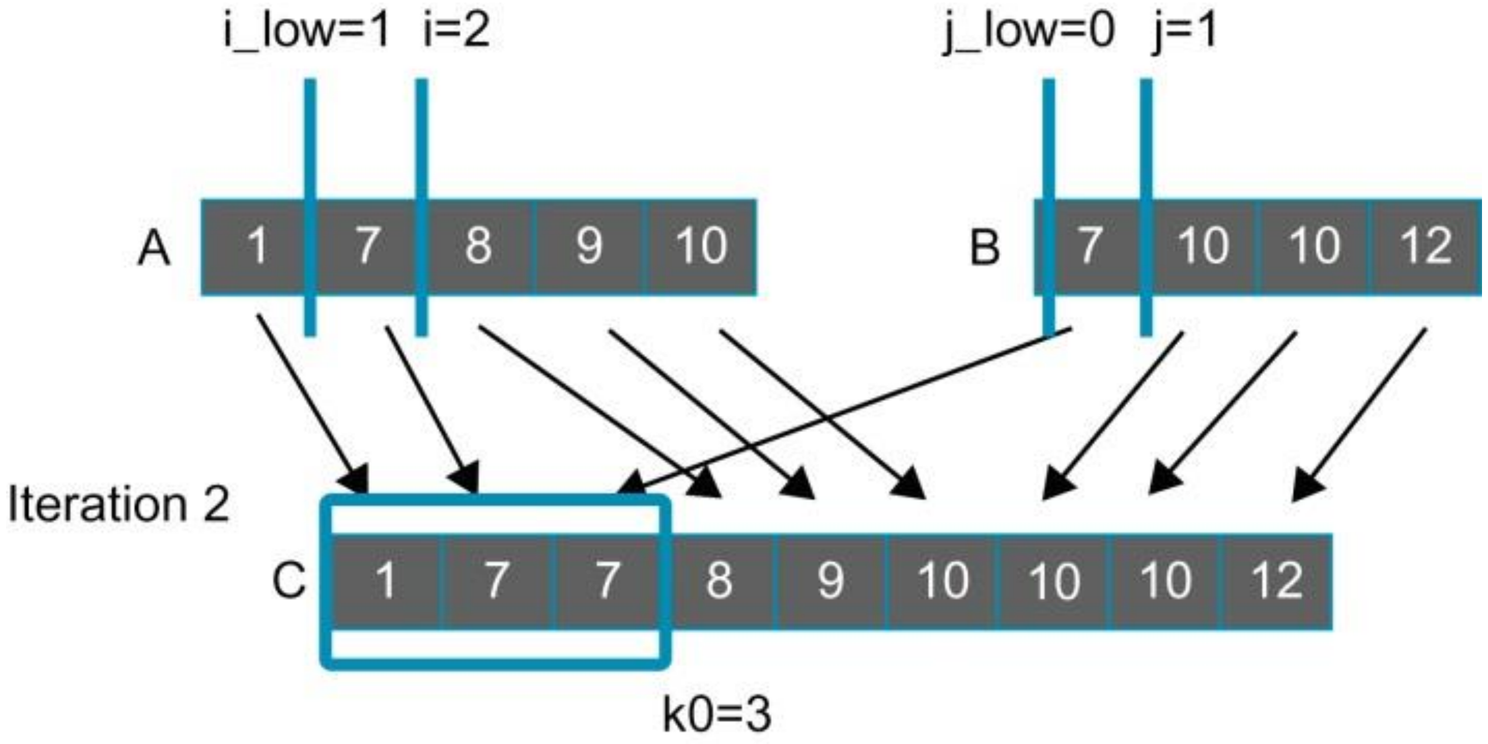

FIGURE 11.9: Iteration 2 of the co-rank function operation example for thread 0.

```
__global__void merge_basic_kernel(int* A, int m, int* B, int n, int* C)


{
    int tid= blockIdx.x*blockDim.x + threadIdx.x;
    int k_curr = tid*ceil((m+n)/(blockDim.x*gridDim.x);                    // start index of output
    int k_next = min((tid+1) * ceil((m+n)/(blockDim.x*gridDim.x)), m+n);   // end index of output
    int i_curr= co_rank(k_curr, A, m, B, n);
    int i_next = co_rank(k_next, A, m, B, n);
    int j_curr = k_curr -i_curr;
    int j_next = k_next-i_next;
            /* All threads call the sequential merge function */
    merge_sequential(&A[i_curr], i_next-i_curr, &B[j_curr], j_next-j_curr, &C[k_curr] );
}
```

FIGURE 11.10: A basic merge kernel.

Parallel Mergesort

- Keep in mind that the total number of output elements may not be a multiple of the number of threads.
- Each thread then makes two calls to the co-rank function. The first call uses k_curr as the rank parameter to get the co-rank values of the first (lowest-indexed) element of the output subarray.

Parallel Mergesort

- The second call uses k_next as the rank parameter to get the co-rank values for the next thread. These co-rank values mark the positions of the lowest-indexed input array elements to be used by the next thread. Therefore, i_next-i_curr and j_next-j_curr give m and n, the sizes of the subarrays of A and B to be used by the current thread.

Parallel Mergesort

- The execution of the basic merge kernel can be illustrated with the example in Fig. 11.9. The k_curr values for the three threads (threads 0, 1, and 2) will be 0, 3, and 6. We will focus on the execution of thread 1 whose k_curr value will be 3. The i_curr and j_curr values returned from the two co-rank function calls are 2 and 1. The k_next value for thread 1 will be 6. The call to the co-rank function gives the i_next and j_next values of 5 and 1. Thread 1 then calls the merge function with parameters (&A[2], 3, &B[1], 0, &C[3]). Note that the 0 value for parameter n indicates that none of the three elements of the output subarray for thread 1 should come from array B. This is indeed the case in Fig. 11.9: output elements C[3]-C[5] come from A[2]-A[4].