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

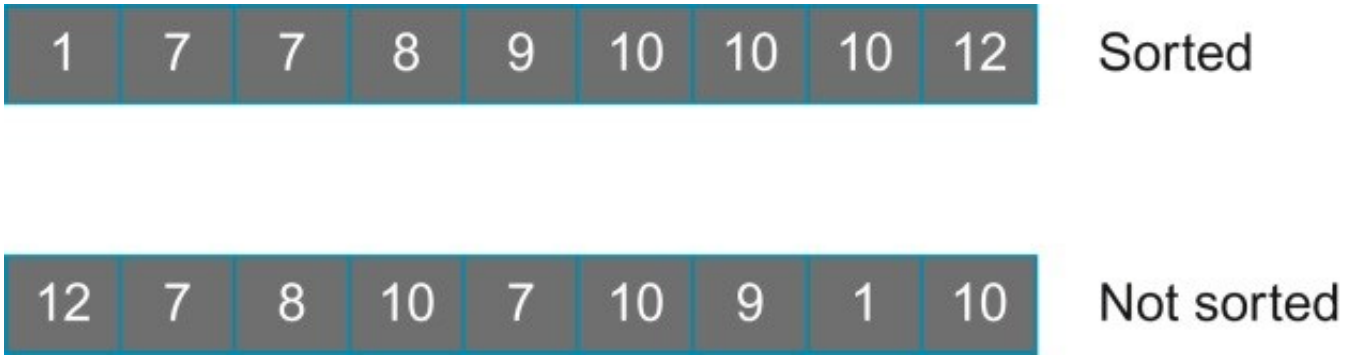

FIGURE 11.1: Examples of sorted versus unsorted lists.

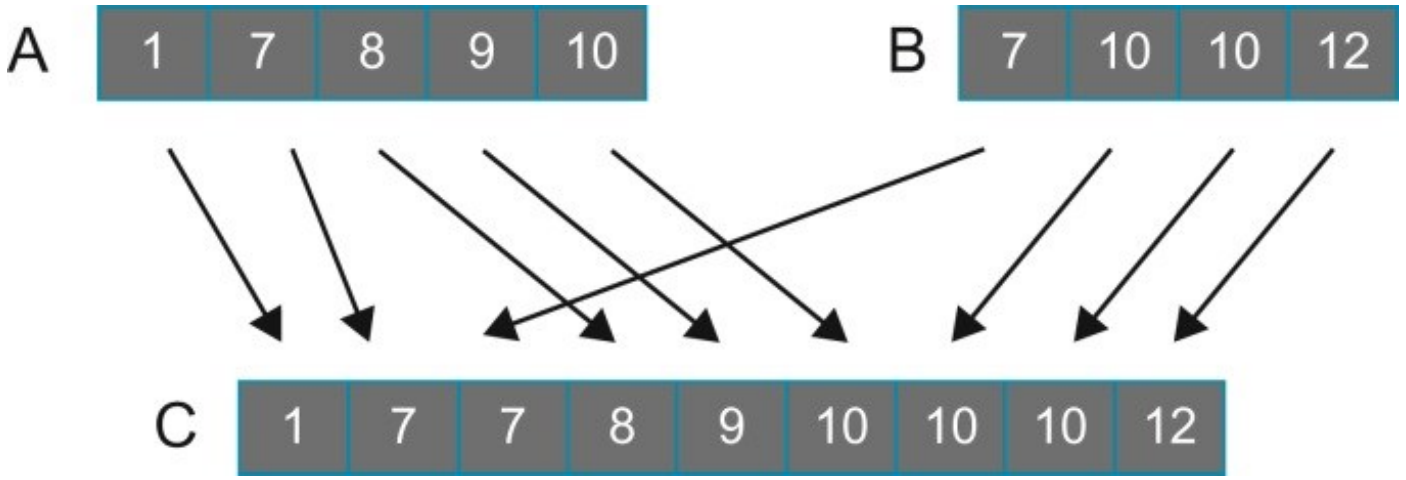

FIGURE 11.2: Example of a merge operation.

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

              // handle the start of A[] and B[]
5     while ((i < m) && (j < n)) {
6       if (A[i] <= B[j]) {
7         C[k++] = A[i++];
8       } else {
9         C[k++] = B[j++];
10      }
11    }
12    if (i == m) {
            //done with A[] handle remaining B[]
13      for (; j < n; j++) {
14        C[k++] = B[j];
15      }
16    } else {
            //done with B[], handle remaining A[]
17      for (; i <m; i++) {
18        C[k++] = A[i];
19      }
20    }
21 }
```

FIGURE 11.3: A sequential merge function.

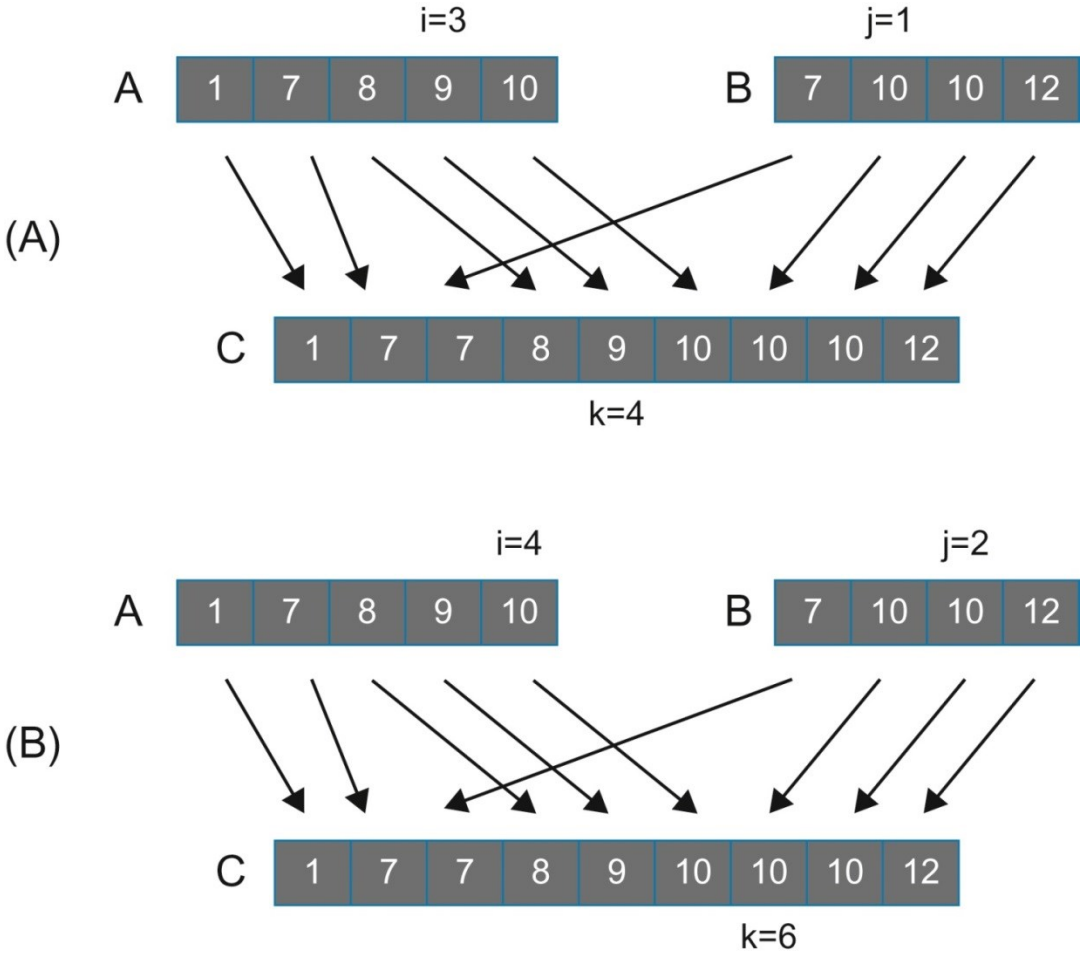

FIGURE 11.4: Examples of observation (1). (A) shows case 1 and (B) shows case 2.

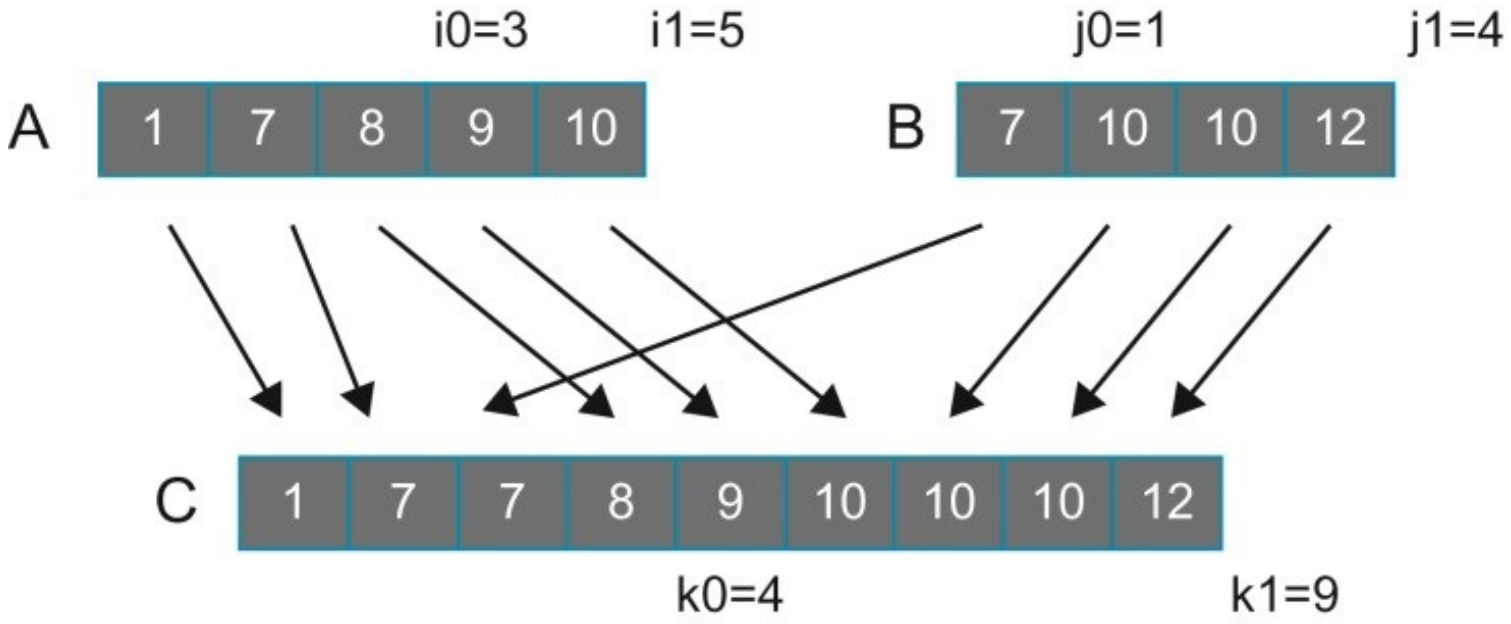

FIGURE 11.5: Example of co-rank function execution.

```
1 int co_rank(int k, int* A, int m, int* B, int n) {
2   int i= k<m ? k : m;  //i = min(k,m)
3   int j = k- i;
4   int i_low = 0>(k-n) ? 0 : k-n;  //i_low = max(0, k-n)
5   int j_low = 0>(k-m) ? 0 : k-m;  //i_low = max(0, k-m)
6   int delta;
7   bool active = true;
8   while(active)    {
9      if (i > 0 && j < n && A[i-1] > B[j]) {
10        delta = ((i - i_low +1) >> 1);  // ceil(i-i_low)/2)
11        j_low = j;
12        j = j + delta;
13        i = i - delta;
14     } else if (j > 0 && i < m && B[j-1] >= A[i]) {
15        delta = ((j - j_low +1) >> 1);
16        i_low = i;
17        i = i + delta;
18        j = j - delta;
19     } else {
20        active = false;
21     }
22  }
23  return i;
24 }
```

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

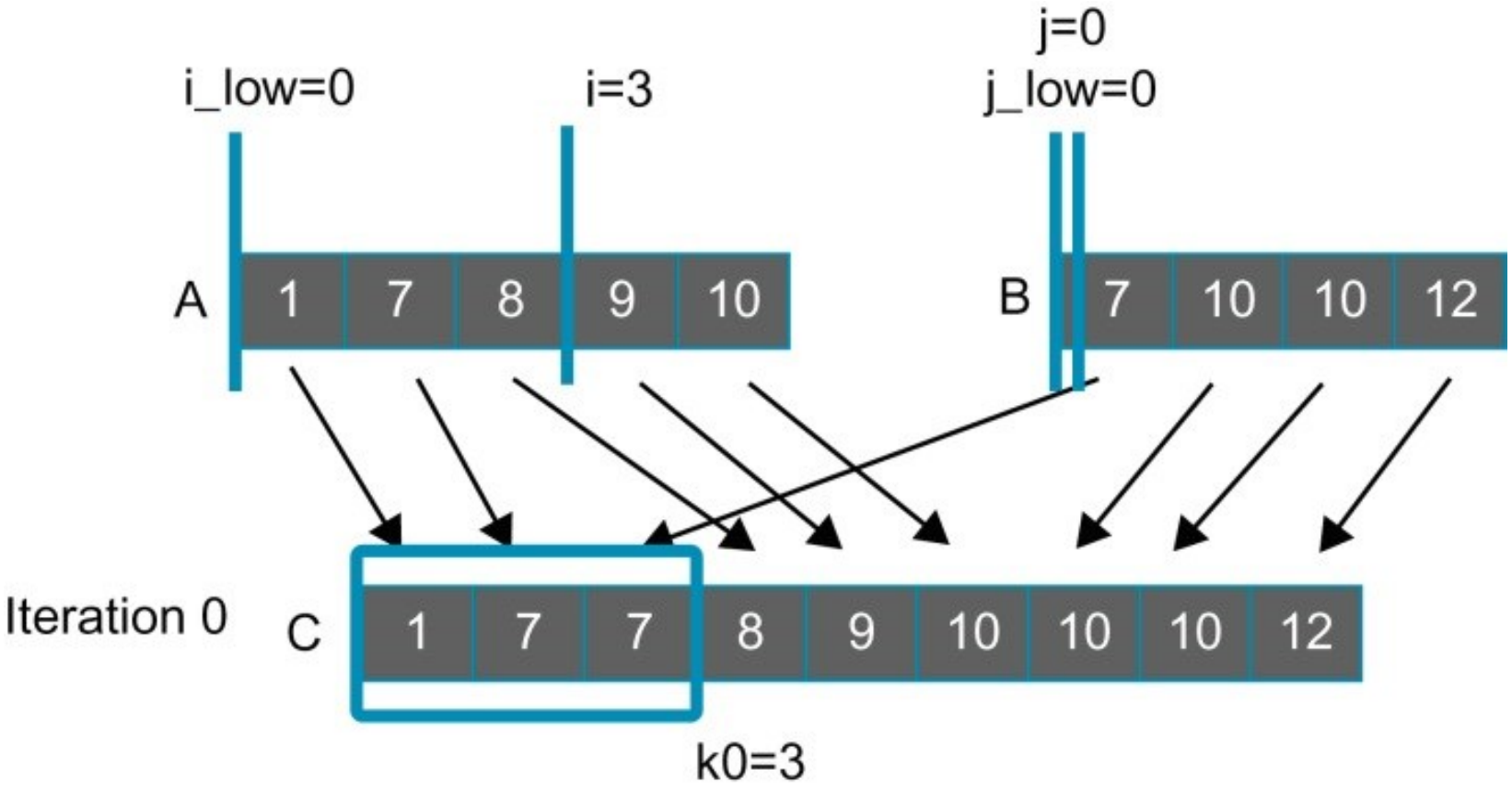

FIGURE 11.7: Iteration 0 of the co-rank function operation example for thread 0.

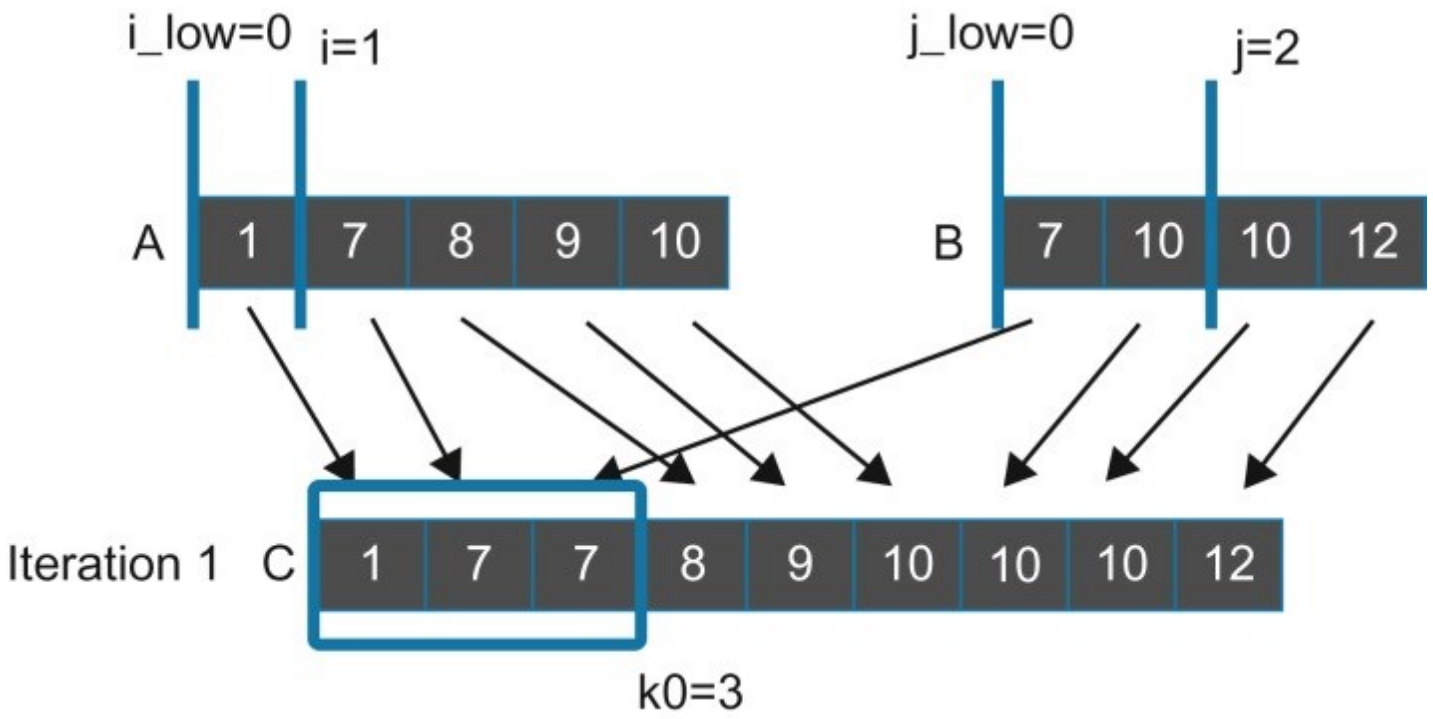

FIGURE 11.8: Iteration 1 of the co-rank function operation example for thread 0.

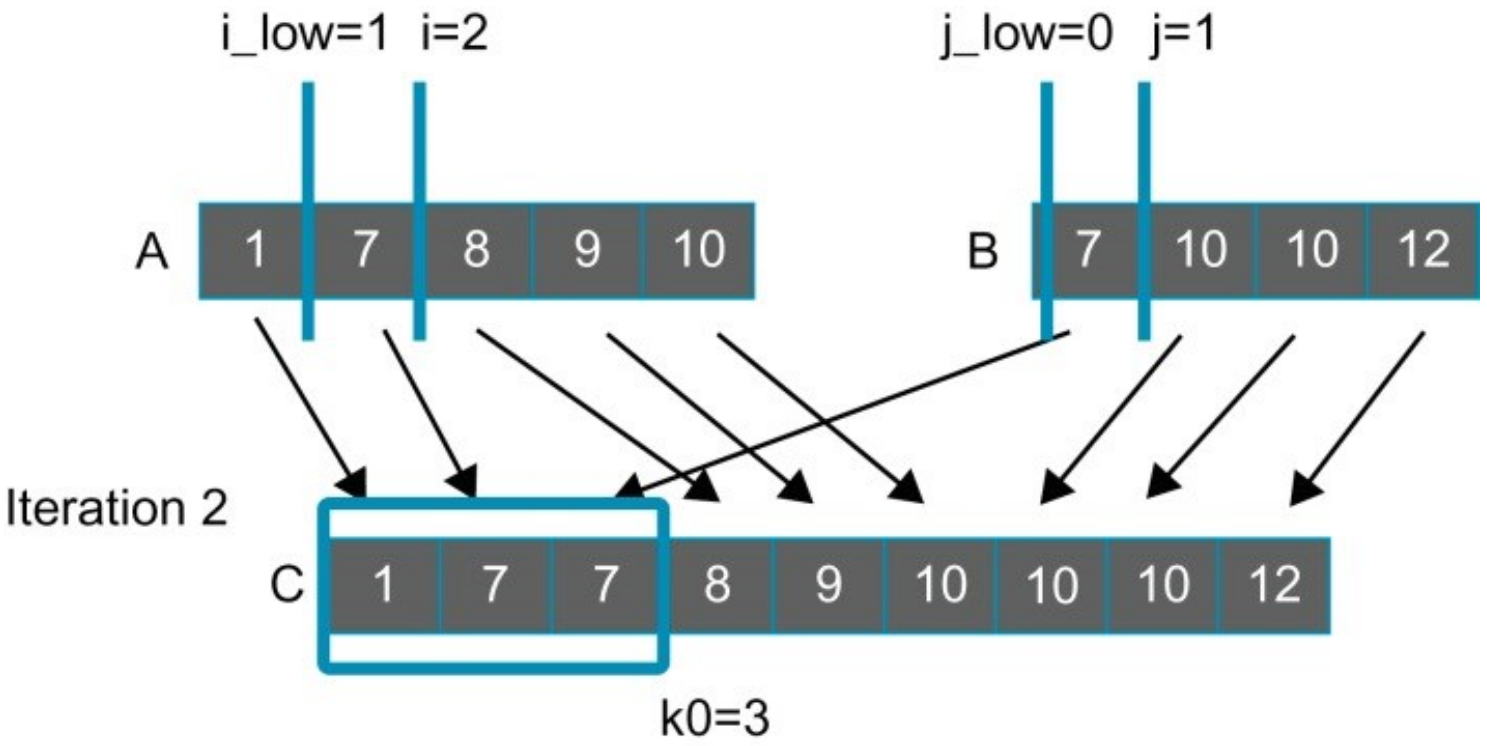

FIGURE 11.9: Iteration 2 of the co-rank function operation example for thread 0.

```
__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.

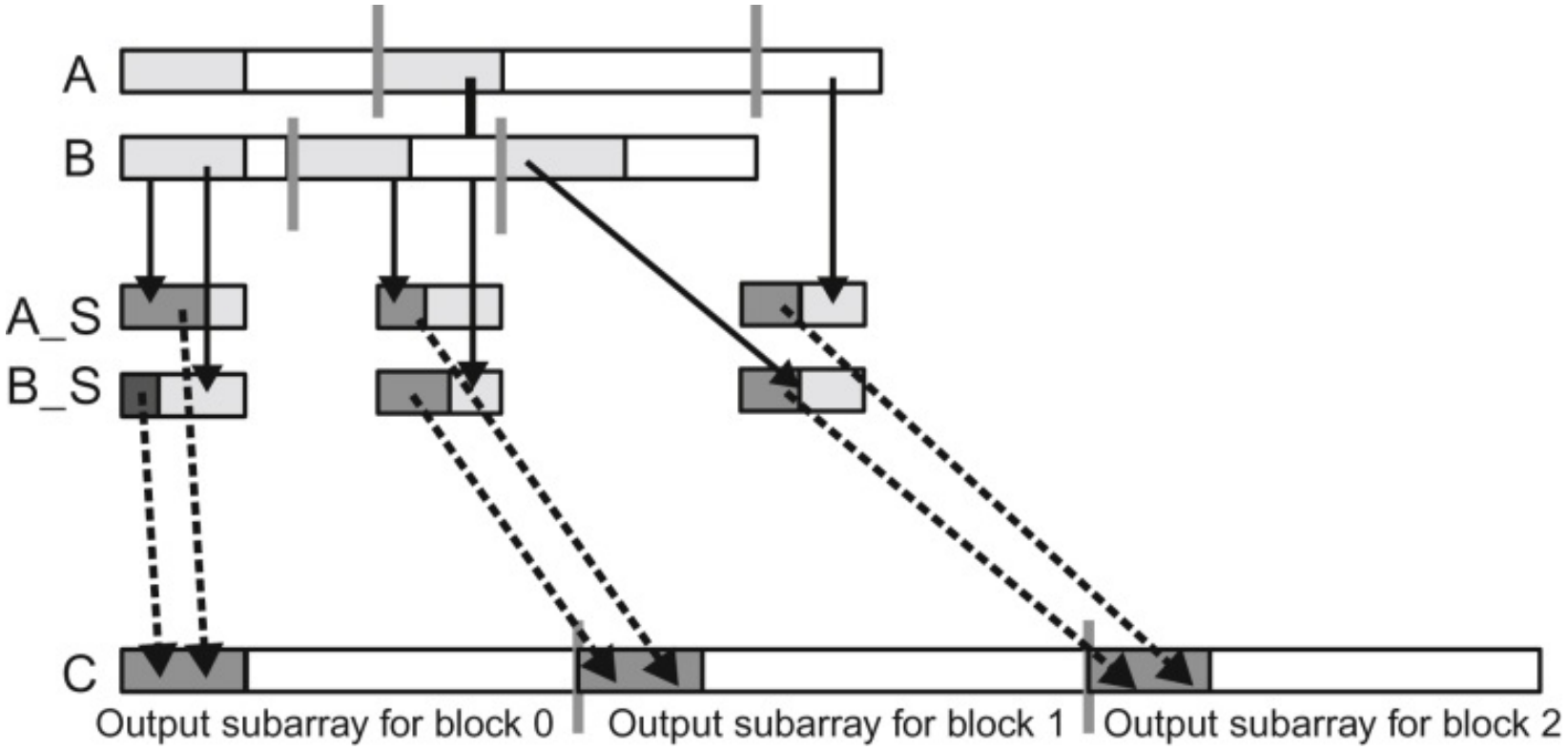

FIGURE 11.11: Design of a tiled merge kernel

```
_global_ void merge_tiled_kernel(int* A, int m, int* B, int n, int* C, int tile_size)



{
  /* shared memory allocation */
  extern _shared_ int shareAB[];
  int * A_S = &shareAB[0];                  //shareA is first half of shareAB
  int * B_S = &shareAB[tile_size];          //ShareB is second half of ShareAB
  int C_curr  = blockIdx.x * ceil((m+n)/gridDim.x) ;          // starting point of the C subarray for current block
  int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), (m+n));     // starting point for next block

  if (threadIdx.x ==0)
  {
    A_S[0] = co_rank(C_curr, A, m, B, n);      // Make the block-level co-rank values visible to
    A_S[1] = co_rank(C_next, A, m, B, n);      // other threads in the block
  }
  _syncthreads();
  int A_curr  = A_S[0];
  int A_next  = A_S[1];
  int B_curr = C_curr – A_curr;
  int B_next   = C_next – A_next;
  _syncthreads();
```

FIGURE 11.12: Part 1—identifying block-level output and input subarrays.

```
int counter = 0;                                        //iteration counter
int C_length = C_next – C_curr;
int A_length = A_next – A_curr;
int B_length = B_next – B_curr;
int total_iteration = ceil((C_length)/tile_size);       //total iteration
int C_completed = 0;
int A_consumed = 0;

int B_consumed = 0;
while(counter < total_iteration)
{
   /* loading tile-size A and B elements into shared memory */
   for(int i=0; i<tile_size; i+=blockDim.x)
   {
      if( i + threadIdx.x < A_length – A_consumed)
      {
         A_S[i + threadIdx.x] = A[A_curr + A_consumed + i + threadIdx.x ];
      }
   }
   for(int i=0; i<tile_size; i+=blockDim.x)
   {
      if(i + threadIdx.x  < B_length – B_consumed)
      {
         B_S[i + threadIdx.x] = B[B_curr + B_consumed + i + threadIdx.x];
      }
   }
   __syncthreads();
```

FIGURE 11.13: Part 2—loading A and B elements into the shared memory.

```
    int c_curr = threadIdx.x   *  (tile_size/blockDim.x);
    int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);
    c_curr  = (c_curr <= C_length – C_completed) ? c_curr : C_length – C_completed;
    c_next = (c_next <= C_length – C_completed) ? c_next : C_length – C_completed;
    /* find co-rank for c_curr and c_next */
    int a_curr = co_rank(c_curr, A_S, min(tile_size, A_length-A_consumed),
                                          B_S, min(tile_size, B_length-B_consumed));
    int b_curr = c_curr – a_curr;
    int a_next = co_rank(c_next, A_S, min(tile_size, A_length-A_consumed),
                                          B_S, min(tile_size, B_length-B_consumed));
    int b_next = c_next – a_next;

    /* All threads call the sequential merge function */
    merge_sequential (A_S+a_curr, a_next-a_curr, B_S+b_curr, b_next-b_curr,
                C+C_curr+C_completed+c_curr);
    /* Update the A and B elements that have been consumed thus far */
    counter ++;
    C_completed += tile_size;
    A_consumed += co_rank(tile_size,  A_S, tile_size, B_S, tile_size);
    B_consumed = C_completed – A_consumed;
    _syncthreads();
  }
}
```

FIGURE 11.14: Part 3—all threads merge their individual subarrays in parallel

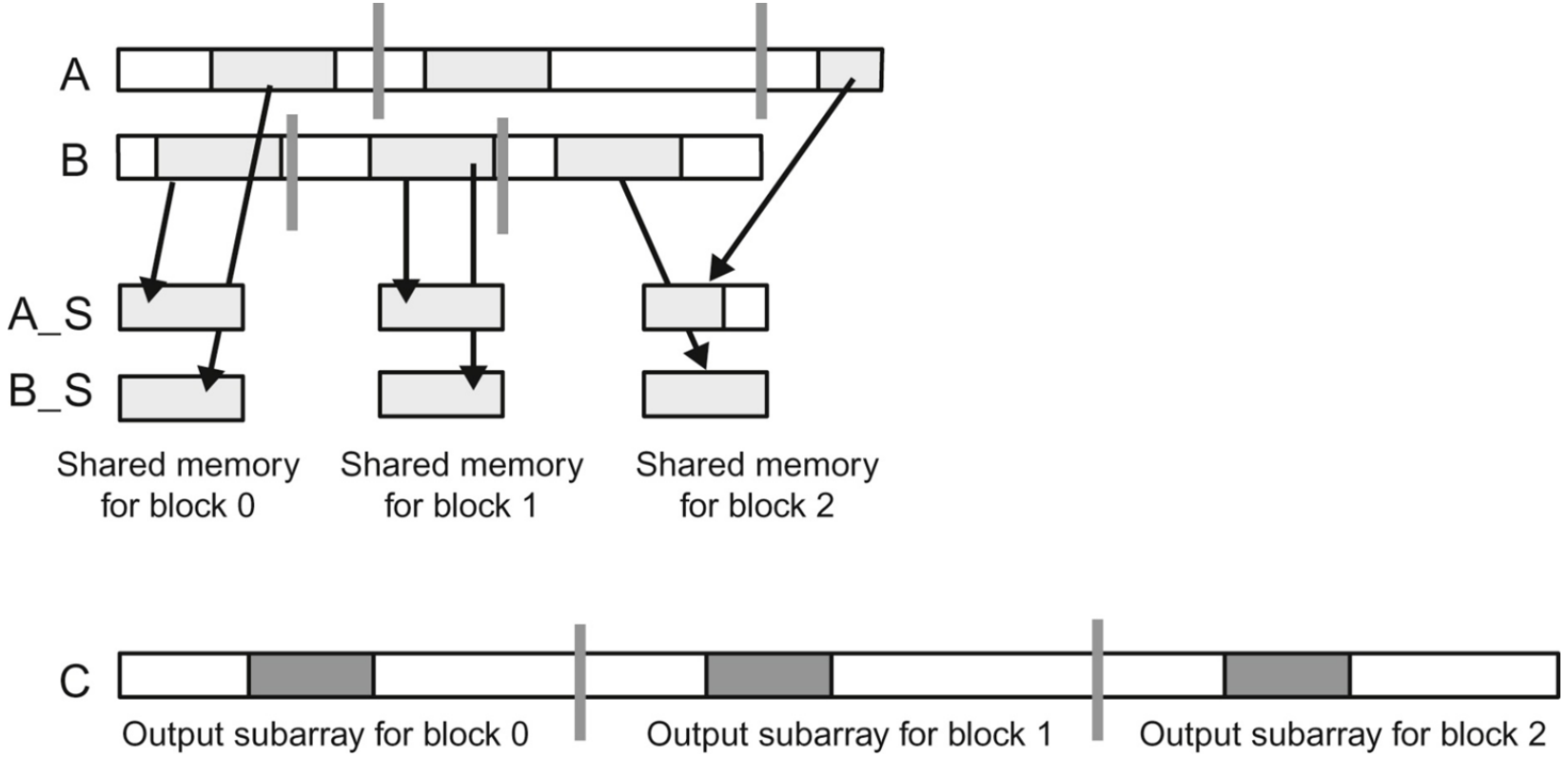

FIGURE 11.15: Iteration 1 of the running example.

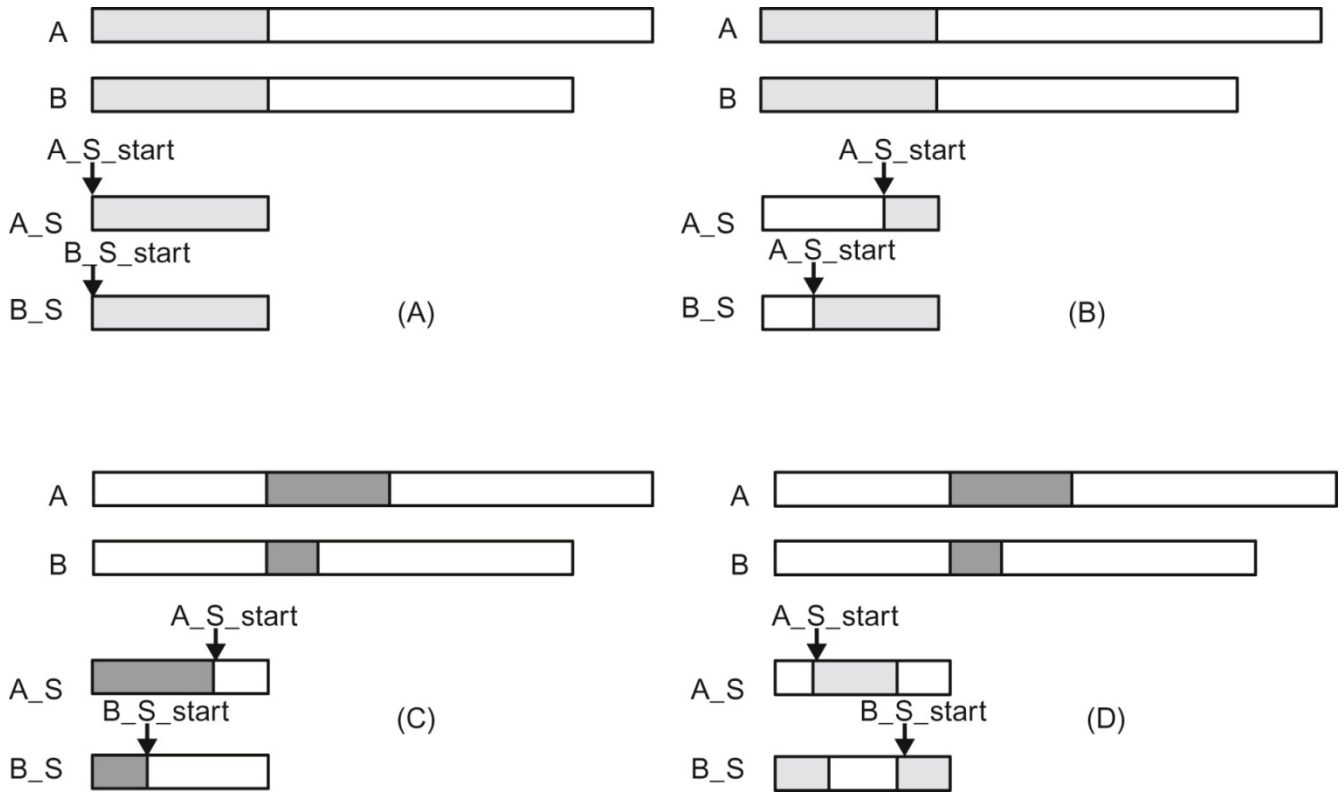

FIGURE 11.16: A circular buffer scheme for managing the shared memory tiles.

```
int A_S_start = 0;
int B_S_start = 0;
int A_S_consumed = tile_size;  //in the first iteration, fill the tile_size
int B_S_consumed = tile_size;  //in the first iteration, fill the tile_size
while(counter < total_iteration)
{
        /* loading A_S_consumed elements into A_S */
        for(int i=0; i<A_S_consumed; i+=blockDim.x)
        {
                if( i + threadIdx.x < A_length – A_consumed && i + threadIdx.x < A_S_consumed)
                {
                        A_S[(A_S_start + i + threadIdx.x)%tile_size] =
                                A[A_curr + A_consumed + i + threadIdx.x ];
                }
        }
        /* loading B_S_consumed elements into B_S */
        for(int i=0; i<B_S_consumed; i+=blockDim.x)
        {
                if(i + threadIdx.x  < B_length – B_consumed && i + threadIdx.x < B_S_consumed)
                {
                        B_S[(B_S_start + i + threadIdx.x)%tile_size] =
                                B[B_curr + B_consumed + i + threadIdx.x];

                }
        }
```

FIGURE 11.17: Part 2 of a circular-buffer merge kernel.

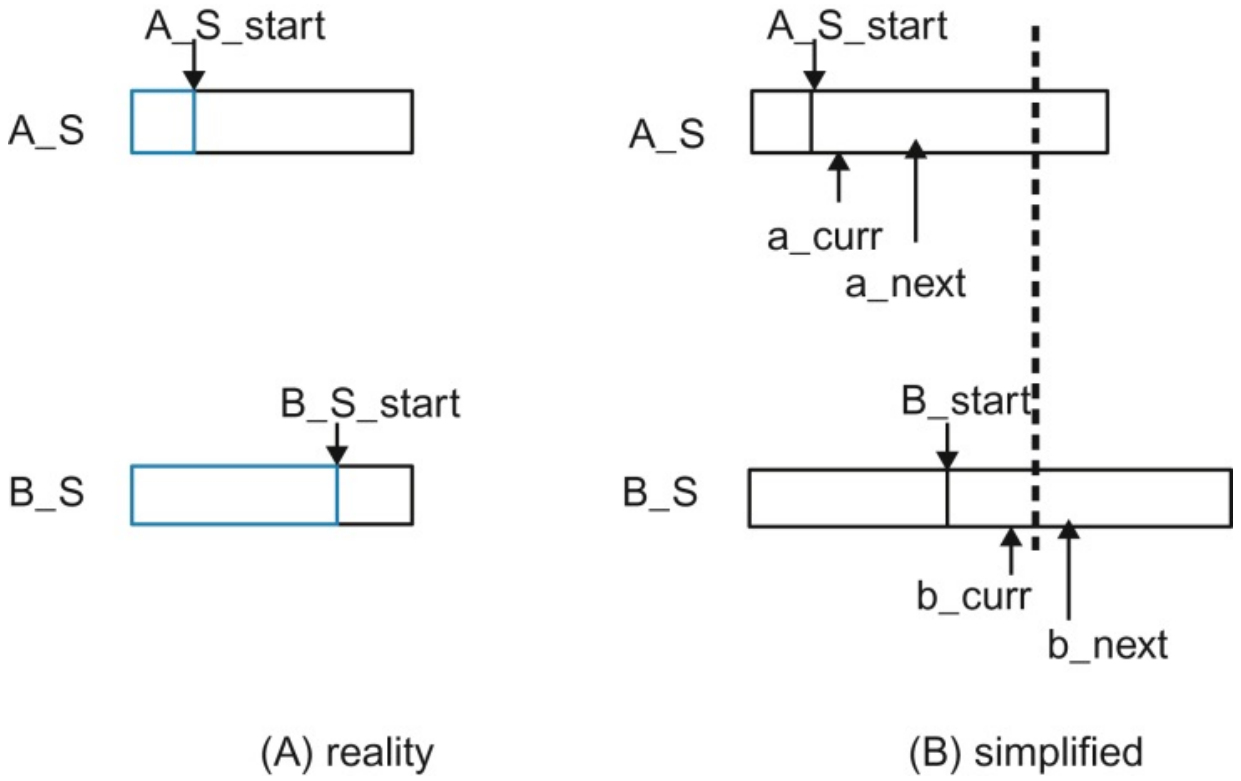

FIGURE 11.18: A simplified model for the co-rank values when using a circular buffer.

```
int c_curr  = threadIdx.x   *  (tile_size/blockDim.x);
int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);

c_curr  = (c_curr <= C_length-C_completed) ? c_curr : C_length-C_completed;
c_next = (c_next <= C_length-C_completed) ? c_next : C_length-C_completed;
          /* find co-rank for c_curr and c_next */
int a_curr = co_rank_circular(c_curr,
              A_S, min(tile_size, A_length-A_consumed),
              B_S, min(tile_size, B_length-B_consumed),
              A_S_start, B_S_start, tile_size);
int b_curr = c_curr -a_curr;
int a_next = co_rank_circular(c_next,
              A_S, min(tile_size, A_length-A_consumed),
              B_S, min(tile_size, B_length-B_consumed),
              A_S_start, B_S_start, tile_size);

int b_next = c_next – a_next;
          /* All threads call the circular-buffer version of the sequential merge function */
merge_sequetial_circular( A_S, a_next-a_curr,
              B_S, b_next-b_curr, C+C_curr+C_completed+c_curr,
                    A_S_start+a_curr, B_S_start+b_curr, tile_size);

          /* Figure out the work has been done */
 counter ++;
 A_S_consumed = co_rank_circular(min(tile_size,C_length-C_completed),
              A_S, min(tile_size, A_length-A_consumed),
              B_S, min(tile_size, B_length-B_consumed),
              A_S_start, B_S_start, tile_size);
  B_S_consumed = min(tile_size, C_length-C_completed) -A_S_consumed;
  A_consumed+= A_S_consumed;
  C_completed += min(tile_size, C_length-C_completed);
  B_consumed = C_completed -A_consumed;

  A_S_start = A_S_start + A_S_consumed;
 if (A_S_start >= tile_size) A_S_start = A_S_start -tile_size;
 B_S_start = B_S_start + B_S_consumed;
 if (B_S_start >= tile_size) B_S_start = B_S_start -tile_size;

__syncthreads();
}
}
```

FIGURE 11.19: Part 3 of a circular-buffer merge kernel.

```
int co_rank_circular(int k, int* A, intm, int* B, int n,
int A_S_start, int B_S_start, inttile_size)
{
   inti= 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)
   {
      int i_cir = (A_S_start+i> = tile_size) ?
              A_S_start+i-tile_size : A_S_start+i;

      int i_m_1_cir = (A_S_start+i-1 > = tile_size)?
              A_S_start+i-1-tile_size: A_S_start+i-1;

      int j_cir = (B_S_start+j> = tile_size) ?
              B_S_start+j-tile_size : B_S_start+j;

      int j_m_1_cir = (B_S_start+i-1 >= tile_size)?
              B_S_start+j-1-tile_size: B_S_start+j-1;

      if (i > 0 && j < n && A[i_m_1_cir] > B[j_cir]) {
         delta = ((i-i_low +1) >> 1) ; // ceil(i-i_low)/2)
         j_low = j;
         i = i - delta;
         j = j + delta;
       } else if (j > 0 && i < m && B[j_m_1_cir] >= A[i_cir]) {
         delta = ((j - j_low +1) >> 1) ;
         i_low = i;
         i = i + delta;
         j = j - delta;
       } else {
         active = false;
       }
      }
      return i;
}
```

FIGURE 11.20: A co_rank_circular function that operates on circular buffers.

```
void merge_sequential_circular(int*A, intm,
                int*B, intn, int*C, intA_S_start,
                intB_S_start, inttile_size)
{
   int i = 0;  //virtual index into A
   int j = 0;  //virtual index into B
   int k = 0; //virtual index into C

   while ((i < m) && (j < n)) {
     int i_cir= (A_S_start+ i>= tile_size)?
                A_S_start+i-tile_size;  A_S_start+i;
     int j_cir= (B_S_start+ j>= tile_size)?
                B_S_start+j-tile_size;  B_S_start+j;
     if (A[i_cir] <= B[j_cir]) {
        C[k++] = A[i_cir]; i++;
     } else {
        C[k++] = B[j_cir]; j++;
     }
   }
   if (i == m) { //done with A[] handle remaining B[]
     for (; j < n; j++) {
        int j_cir = (B_S_start + j>= tile_size)?
                B_S_start+j-tile_size;  B_S_start+j;
       C[k++] = B[j_cir];
     }
   } else { //done with B[], handle remaining A[]
     for (; i <m; i++) {
        int i_cir = (A_S_start + i>= tile_size)?
                A_S_start+i-tile_size;  A_S_start+i;
        C[k++] = A[i_cir];
     }
   }
}
```

FIGURE 11.21: Implementation of the merge_sequential_circular function.