Chapter-12
Parallel patterns: graph search

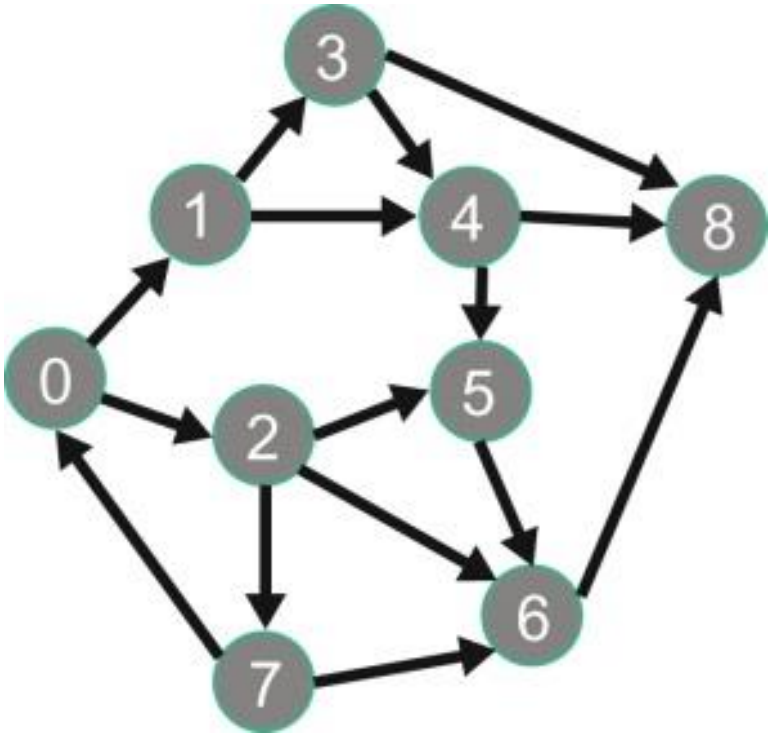
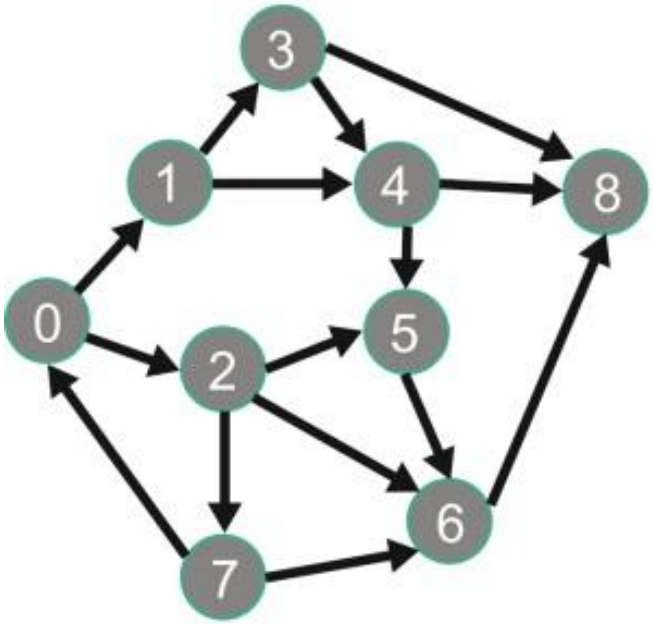


FIGURE 12.1: A simple graph with directional edges.



	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

FIGURE 12.2: Adjacency matrix representation of the simple graph example.

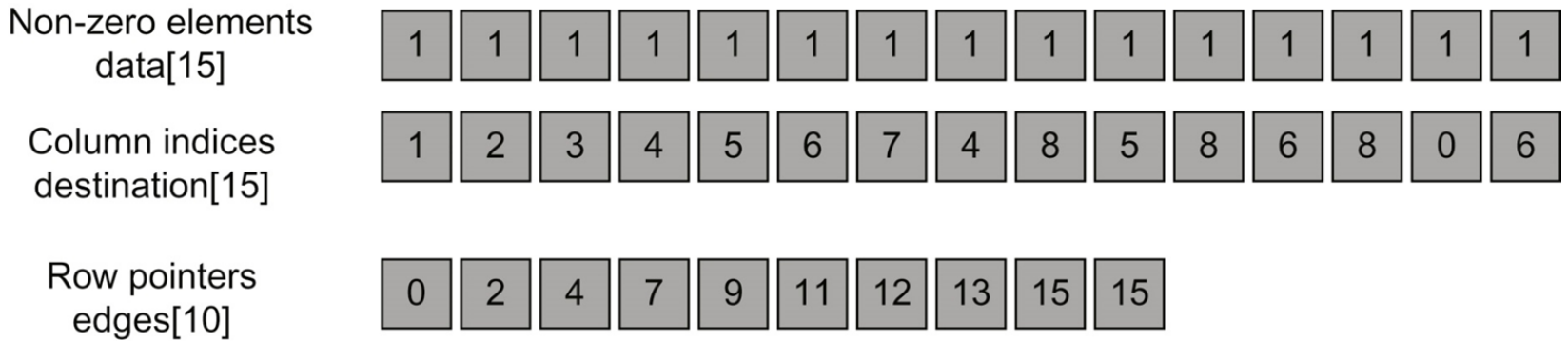


FIGURE 12.3: Sparse matrix (CSR) representation of adjacency matrix.

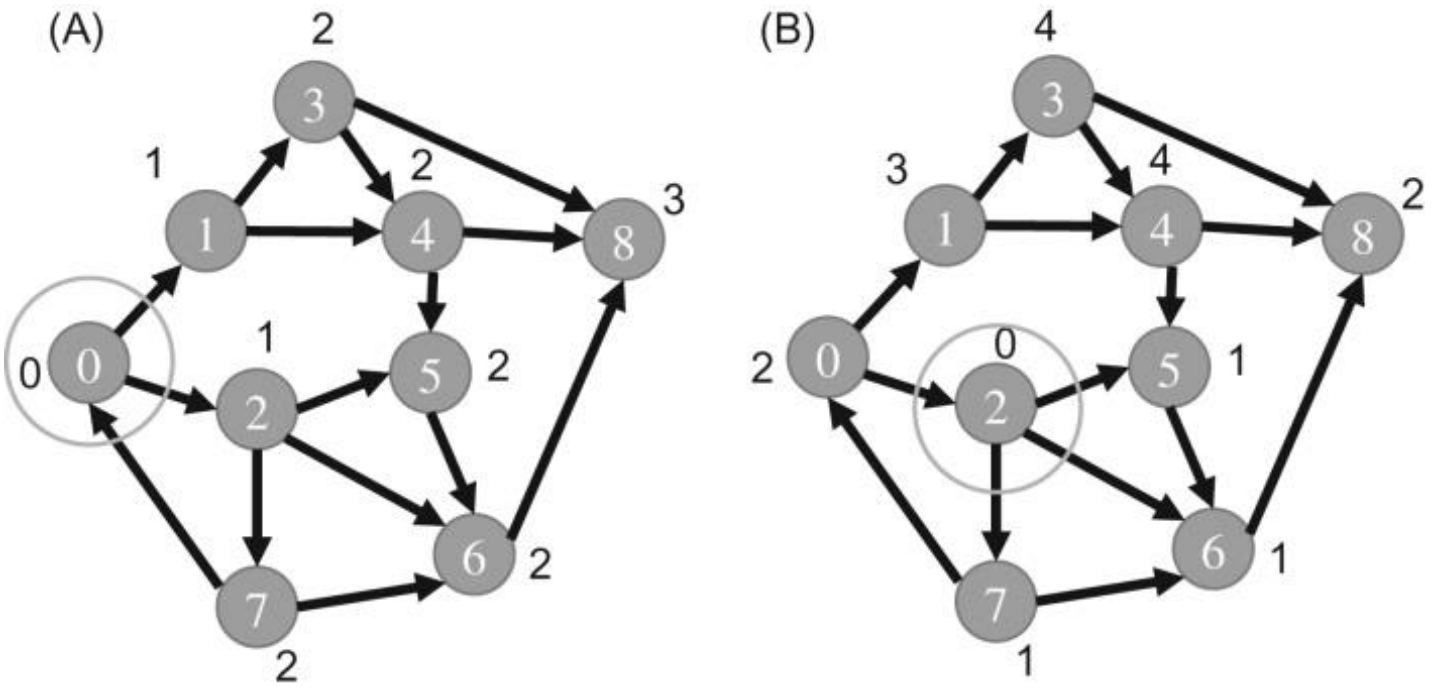


FIGURE 12.4: Breadth-first search results. (A) Vertex 0 is source, (B) vertex 2 is source.

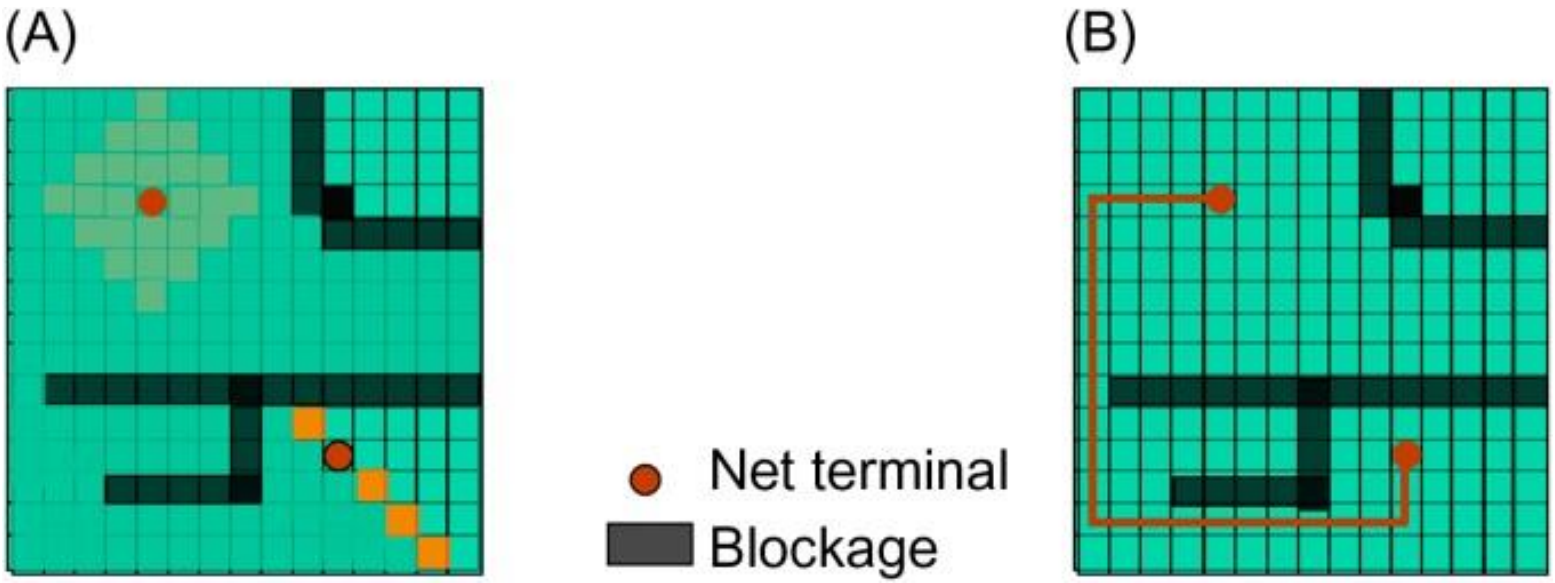


FIGURE 12.5: Maze routing in integrated circuits—an application for breadth-first search. (A) Breadth-first search, (B) identifying a routing path.

```
void BFS_sequential(int source, int *edges, int *dest, int *label)
{
    int frontier[2][MAX_FRONTIER_SIZE];
    int *c_frontier = &frontier[0];
    int c_frontier_tail = 0;
    int *p_frontier = &frontier[1];
    int p_frontier_tail = 0;

    insert_frontier(source, p_frontier, &p_frontier_tail);
    label[source] = 0;

    while (p_frontier_tail > 0) {
       for (int f=0; f < p_frontier_tail; f++) {  // visit all previous frontier vertices
           c_vertex = p_frontier[f];             // Pick up one of the previous frontier vertex
           for  (int i = edges[c_vertex]; i < edges[c_vertex+1]; i++) { //for all its edges
            if (label[dest[i]] == -1) {          // The dest vertex has not been visited
               insert_frontier(dest[i], c_frontier, &c_frontier_tail); // overflow check omitted for brevity
               label[dest[i]] = label[c_vertex]+1;
           }
        }
        int temp = c_frontier;  c_frontier = p_frontier; p_frontier = temp; //swap previous and current
        p_frontier_tail = c_frontier_tail;    c_frontier_tail = 0; //
     }
 }
```

FIGURE 12.6: A sequential breadth-first search function.

```
void BFS_host(unsigned int source, unsigned int *edges, unsigned int *dest, unsigned int *label)
{
    // allocate edges_d, dest_d, label_d, and visited_d in device global memory
    // copy edges, dest, and label to device global memory
    // allocate frontier_d, c_frontier_tail_d, p_frontier_tail_d in device global memory

    unsigned int *c_frontier_d = &frontier_d[0];
    unsigned int *p_frontier_d = &frontier_d[MAX_FRONTIER_SIZE];

    // launch a simple kernel to initialize the following in the device global memory
    // initialize all visited_d elements to 0 except source to 1
    // *c_frontier_tail_d = 0;
    // p_frontier_d[0] = source;
    // *p_frontier_tail_d = 1;
    // label[source] = 0;

    p_frontier_tail = 1;

    while (p_frontier_tail > 0) {
        int num_blocks = ceil(p_frontier_tail/float(BLOCK_SIZE));

        BFS_Bqueue_kernel<<<num_blocks, BLOCK_SIZE>>>(p_frontier_d, p_frontier_tail_d,
                        c_frontier_d, c_frontier_tail_d, edges_d, dest_d, label_d, visited_d);

        // use cudaMemcpy to read the *c_frontier_tail value back to host and assign
        // it to p_frontier_tail for the while-loop condition test

        int* temp = c_frontier_d; c_frontier_d = p_frontier_d; p_frontier_d = temp; //swap the roles
        // launch a simple kernel to set *p_frontier_tail_d = *c_frontier_tail_d; *c_frontier_tail_d = 0;
    }
}
```

FIGURE 12.7: A sketch of the BFS host code function.

```
__global__ void BFS_Bqueue_kernel(unsigned int *p_frontier, unsigned int *p_frontier_tail, unsigned int *c_frontier,
unsigned int *c_frontier_tail, unsigned int *edges, unsigned int *dest, unsigned int *label, unsigned int*
visited) {

    __shared__ unsigned int c_frontier_s[BLOCK_QUEUE_SIZE];
    __shared__ unsigned int c_frontier_tail_s, our_c_frontier_tail;

    if(threadIdx.x == 0) c_frontier_tail_s = 0;
    __syncthreads();

    const unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
    if(tid < *p_frontier_tail) {
        const unsigned int my_vertex = p_frontier[tid];
        for(unsigned int i = edges[my_vertex]; i < edges[my_vertex + 1]; ++i) {
            const unsigned int was_visited = atomicExch(&(visited[dest[i]]), 1);
            if(!was_visited) {
                label[dest[i]] = label[my_vertex] + 1;
                const unsigned int my_tail = atomicAdd(&c_frontier_tail_s, 1);
                if(my_tail < BLOCK_QUEUE_SIZE) {
                    c_frontier_s[my_tail] = dest[i];
                } else { // If full, add it to the global queue directly
                    c_frontier_tail_s = BLOCK_QUEUE_SIZE;
                    const unsigned int my_global_tail = atomicAdd(c_frontier_tail, 1);
                    c_frontier[my_global_tail] = dest[i];
                }
            }
        }
    }
    __syncthreads();

    if(threadIdx.x == 0) {
        our_c_frontier_tail = atomicAdd(c_frontier_tail, c_frontier_tail_s);
    }
    __syncthreads();

    for(unsigned int i = threadIdx.x; i < c_frontier_tail_s; i += blockDim.x) {
        c_frontier[our_c_frontier_tail + i] = c_frontier_s[i];
    }

}
```

FIGURE 12.8: A parallel BFS kernel based on block-level privatized queues.

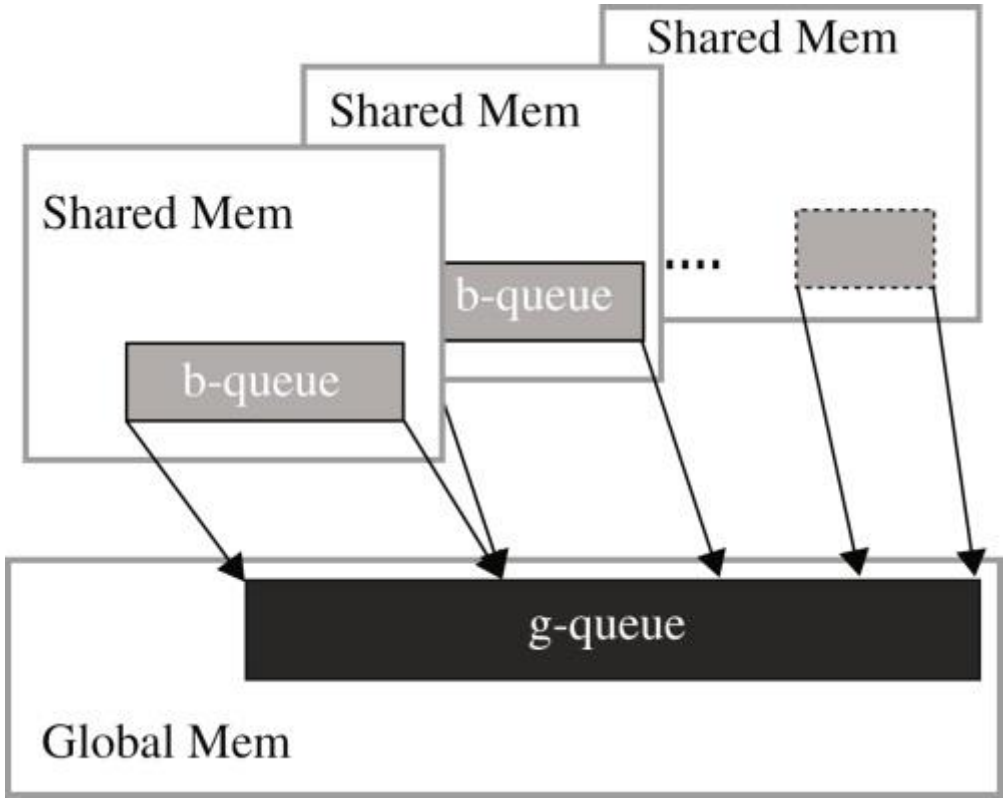


FIGURE 12.9: Block-level queue (b-queue) contents are copied into the global queue (g-queue) at the end of the kernel in a coalesced manner.

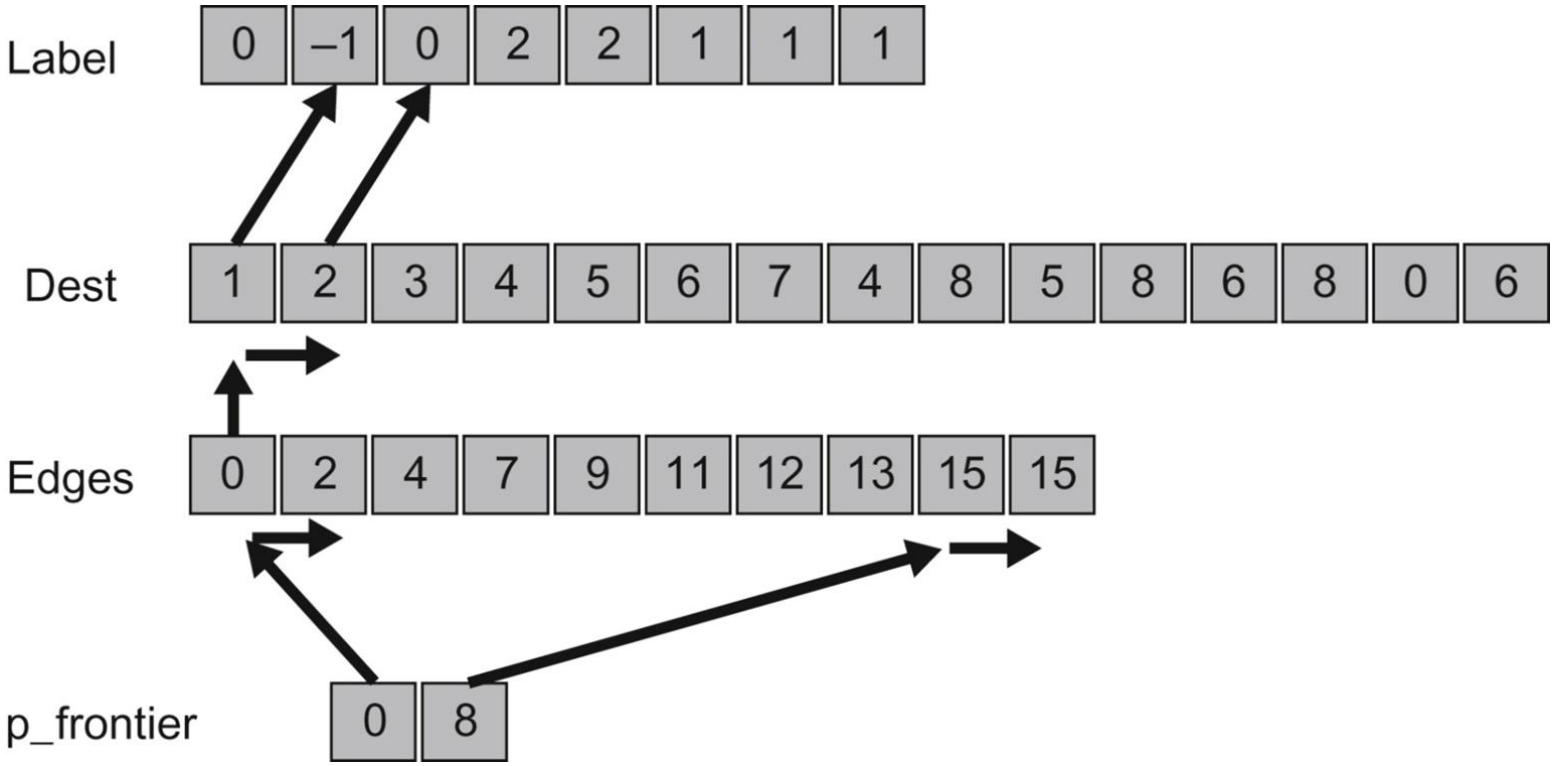


FIGURE 12.10: Memory access pattern for processing the level-2 frontier in Fig. 12.5.

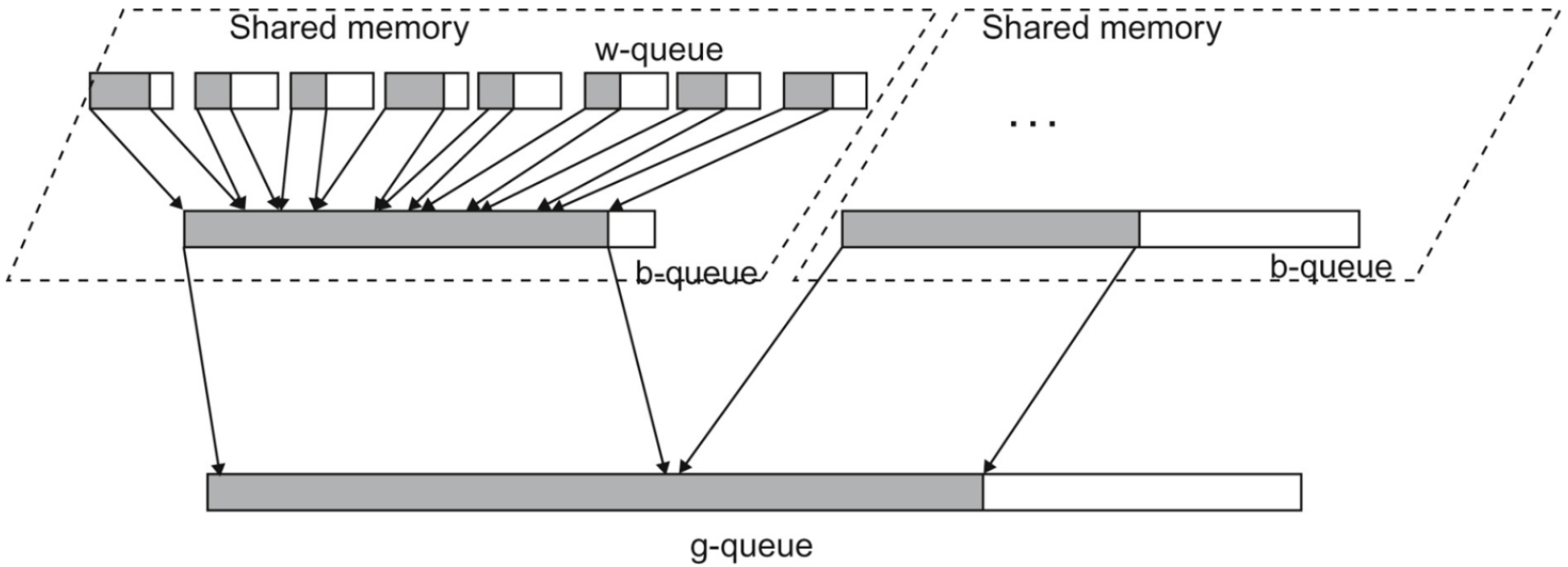


FIGURE 12.11: The design and consolidation process of w-queue, b-queue, and g-queue.