Background

• Adjacency matrix representation
  – Each vertex labeled with unique number
  – 1 if there is an edge – 0 otherwise

• Graph of n vertices fully connected – each vertex has n-1 outgoing edges
  – n(n-1) total edges

• Sparsely connected graphs – sparse matrix
Data array – unnecessary – all 1’s
Breadth-First Search

- Often used to discover shortest path to do something
- Several forms of BFS
  - Simple form – given a source vertex – label each vertex with the smallest number of edges need to traverse from source to this vertex
Fig. 12.4(A) shows the desired BFS result with vertex 0 as the source. Through one edge, we can get to vertices 1 and 2. Thus, we mark these vertices as level 1. By

**FIGURE 12.4**

Breadth-first search results. (A) Vertex 0 is source, (B) vertex 2 is source.
Breadth-First Search

- Do a BFS to get the level numbers
- To find path: start at destination and work back to start
  - Each step – at a vertex v with level number i
    - Find a neighbor of v with level number i – 1
- Example – design of an integrated circuit chip
  - Components need to be connected to complete design
Breadth-First Search

**FIGURE 12.5**

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

- Chip is represented as grid of wiring blocks where each block can potentially serve as a piece of wire
- Wire can be formed by extending in either horizontal or vertical direction
- Already used wiring blocks form a barrier
- **Maze problem!!**
  - Perform BFS on maze & then find shortest 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++) {
            if (label[dest[i]] == -1) {
                insert_frontier (dest[i], c_frontier, &c_frontier_tail);
                label[dest[i]] = label[c_frontex] + 1;
            }
        }
        int temp = c_frontier;
        c_frontier = p_frontier;
        p_frontier = temp;
        p_frontier_tail = c_frontier_tail;
        c_frontier_tail = 0;
    }
}
Parallel BFS Search

- Harish & Narayanan: each thread – assigned a vertex
  - Each iteration – all vertices visited
  - If any source of incoming vertex was visited first last iteration – destination visited this iteration
  - Large number of levels – can be very inefficient

- Luo et al.: parallelize each iteration of loop using multiple threads to collaboratively process previous frontier array & assemble current frontier array
void BFS_host (unsigned int source, unsigned int* edges, unsigned int* dest, unsigned int* label) {
    unsigned int *c_frontier_d = &frontier_d[0];
    unsigned int * p_frontier_d = &frontier_d[MAX_FRONTIER_SIZE];

    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, fest_d label_d, visited_d);
        // use cudaMemcpy to read the *c_frontier_tail value back to the host and assign
        // it top_frontier_tail for the while loop condition test
        int* temp = c_frontier_d;
        c_frontier_d = p_frontier_d;
        p_frontier_d = temp;
    }
__global__ void BFS_Bqueue_kernel (unsigned int *p_frontier, unsigned int *p_frontier_tail, unsigned int *c_frontier, unsigned int *cfrontier_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_x = 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 = atomExch (&(visited[dest[i]]), 1);
            if (!was_visited) {
                Label[dest[i]] = label[my_vertex] + 1;
                const unsigned int my_tail = atomicAll (&c_frontier_tail_s, 1);
                if (my_tail < BLOCK_QUEUE_SIZE) {
                    c_frontier_s[my_tail] = dest[i];
                } else {
                    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_front_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];
}
Parallel BFS Search

- Parallel each iteration of loop – assign a section of previous frontier array to each thread block
- Writing vertices to c_frontier array
  - Need to handle mutual exclusion
    - atomic operations
  - Need to handle thread cache coherence for global memory
    - Terminate & relaunch kernel each loop trip
  - Need to handle writes to global memory
    - Each thread buffer (block_level_queue) writes to global memory
Optimizations

- Memory Bandwidth
  - Accesses to arrays in global memory – adjacent threads are not accessing adjacent global memory locations

- Memory contention
  - Block level queues – goal to reduce memory contention from a global queue
    - Partially successful – still have some contention

- Kernel launch overhead
  - Initialization required for kernel launch – could negate gain from parallelism if kernel execution is small

- Load balance
  - Amount of work done by a thread – depends upon number of neighbors