In this talk we’ll discuss two parallel algorithms to compute shortest paths in unweighted directed graphs. The first one has some (but not great) improvement in time complexity and is efficient in terms of work, while the other has a much greater improvement in time complexity, but at the cost of inefficient work. In a future lecture, we’ll use both of these algorithms to make an algorithm that is efficient in both time and work.

0 Preliminaries

1. \( G = (V, E) \) is an unweighted directed graph, \(|V| = n, |E| = m\). Let \( s, t \in V \). The problem is to find the shortest path from \( s \) to \( t \).

2. Non-parallel BFS (breadth-first search):
   - Start with \( s \) at level 0.
   - For \( i = 0, \ldots, n - 1 \), collect all outgoing edges \((u, v)\) from level \( i \) and put \( v \) into level \( i + 1 \) if \( v \) has not already been visited.
   - The distance from \( s \) to \( t \) is the level of \( t \), or \( \infty \) if \( t \) was never visited (thus unreachable from \( s \)).
   - This takes \( O(m) \) time because each edge must be processed once in the worst case.

3. For parallel implementations:
   - We use the CRCW (concurrent-read-concurrent-write) model. On concurrent write of values \( x_1, \ldots, x_k \), a random choice from \( x_1, \ldots, x_k \) is written.
   - Results about parallel computing that we will use:
     - There is an \( O(\log^*(p)) \) algorithm to balance load among \( p \) processors with high probability.
     - Can accumulate processors’ outputs (sum, min, etc.) in anywhere from \( O(1) \) to \( O(\log p) \) time, depending on the model.
     - We will assume both of the above are \( O(1) \) to simplify analysis, effectively ignoring the effect of choice of model on our analysis.

1 Parallel BFS

Take the normal BFS algorithm and parallelize the processing of the outgoing edges in each level.

- Let \( m_i \) be the number of edges outgoing from level \( i \). Then each level takes \( O(1 + m_i/p) \) time and \( O(p + m_i) \) work.
• Therefore, in total this takes \( \sum_{i=1}^{n-1} O(1 + m_i/p) = O(n) + O(\sum_{i=1}^{n-1} m_i/p) = O(n + m/p) \) time since \( \sum_{i=1}^{n-1} m_i = m \) in the worst case. The work is then \( O(np + m) \).

• If we set \( p = m/n \), then the runtime is \( O(n) \) while the work is \( O(m) \). This is a minor speedup over the non-parallel algorithm, with only a constant factor additional cost in work.

2 Matrix multiplication

• Order the vertices in some order, and call them \( v_1, \ldots, v_n \).

• Warm-up: Let \( A \) be the adjacency matrix of \( G \). Then \( (A^\ell)_{i,j} \) counts the number of paths from \( v_i \) to \( v_j \) of length \( \ell \). (Why?)

Similarly, \( ((A + I)^n)_{i,j} > 0 \) if and only if \( v_j \) is reachable from \( v_i \). (Why?)

• We adapt the above observation slightly to find shortest paths instead of reachability. Define the \( \ell \)-step distance from \( v_i \) to \( v_j \) to be \( d(v_i, v_j) \) if there exists a path from \( v_i \) to \( v_j \) using at most \( \ell \) edges, or \( \infty \) otherwise. Let \( A \) be the 1-step distance matrix, that is,

\[
A_{i,j} := \begin{cases} 
0 & i = j \\
1 & (v_i, v_j) \in E \\
\infty & \text{otherwise.}
\end{cases}
\]

and compute \( A^* := \bigotimes_{\ell=1}^n A \) where \( A \ast B \) is defined by

\[
(A \ast B)_{i,j} := \min_{k=1,\ldots,n} (A_{i,k} + B_{k,j}).
\]

(This definition is just matrix multiplication where the matrix entries are viewed as elements of \((\mathbb{Z} \cup \{\infty\}, \min, +)\), the min-plus semiring, also known as the tropical semiring.)

• Example: \( G: \{v_1 \rightarrow v_2 \rightarrow v_3\} \), \( A = \begin{bmatrix} 0 & 1 & \infty \\ \infty & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix} \), \( A^2 = \begin{bmatrix} 0 & 1 & 2 \\ \infty & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix} \).

• Claim: For \( \ell \geq 1 \), \( A^\ell \) is precisely the \( \ell \)-step distance matrix.

Proof: Induction on the fact that the \( \ell \)-step distance from \( v_i \) to \( v_j \) is the minimum over all \( k \) of the 1-step distance from \( v_i \) to \( v_k \) plus the \((\ell - 1)\)-step distance from \( v_k \) to \( v_j \).

• The algorithm: Compute \( A^* \) and output \( (A^*)_{s,t} \).

Question: How to compute \( A^* \) efficiently in parallel? (Hint: We can multiply two matrices in \( O(1) \) time using \( n^3 \) processors.)

Final complexity bounds: \( O(\log n) \) time, \( O(n^3 \log n) \) work. Good time complexity, inefficient work complexity. However, note that this algorithm works on weighted graphs too, and actually computes all-pairs shortest paths.