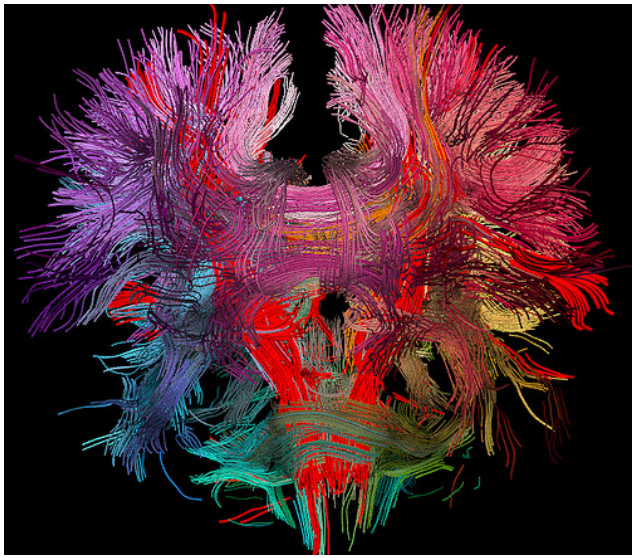
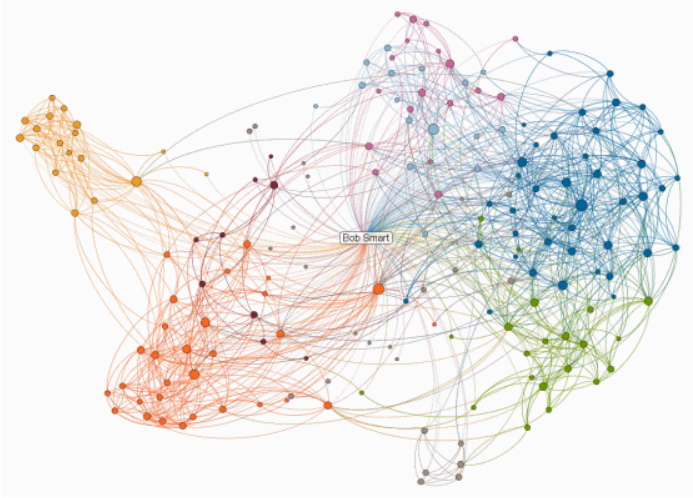
Distributed Graph Processing

Abhishek Verma

CS425

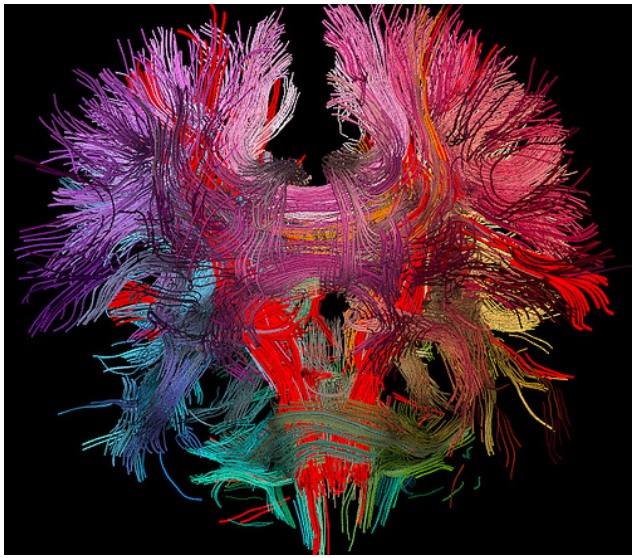
Guess the Graph?



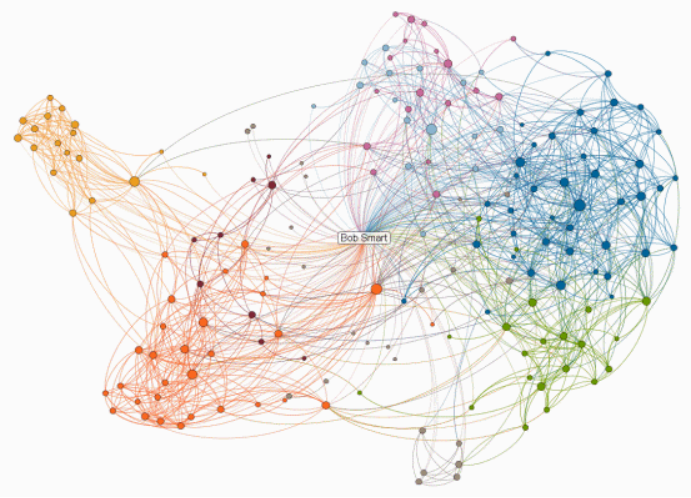


Guess the Graph?

The Internet



Human Brain



LinkedIn Social Network

Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Google Fiber
- Finding Max Flow
  - Scheduling
- Bipartite matching
  - Dating websites
- Identify special nodes and communities
  - Spread of diseases, terrorists

Outline

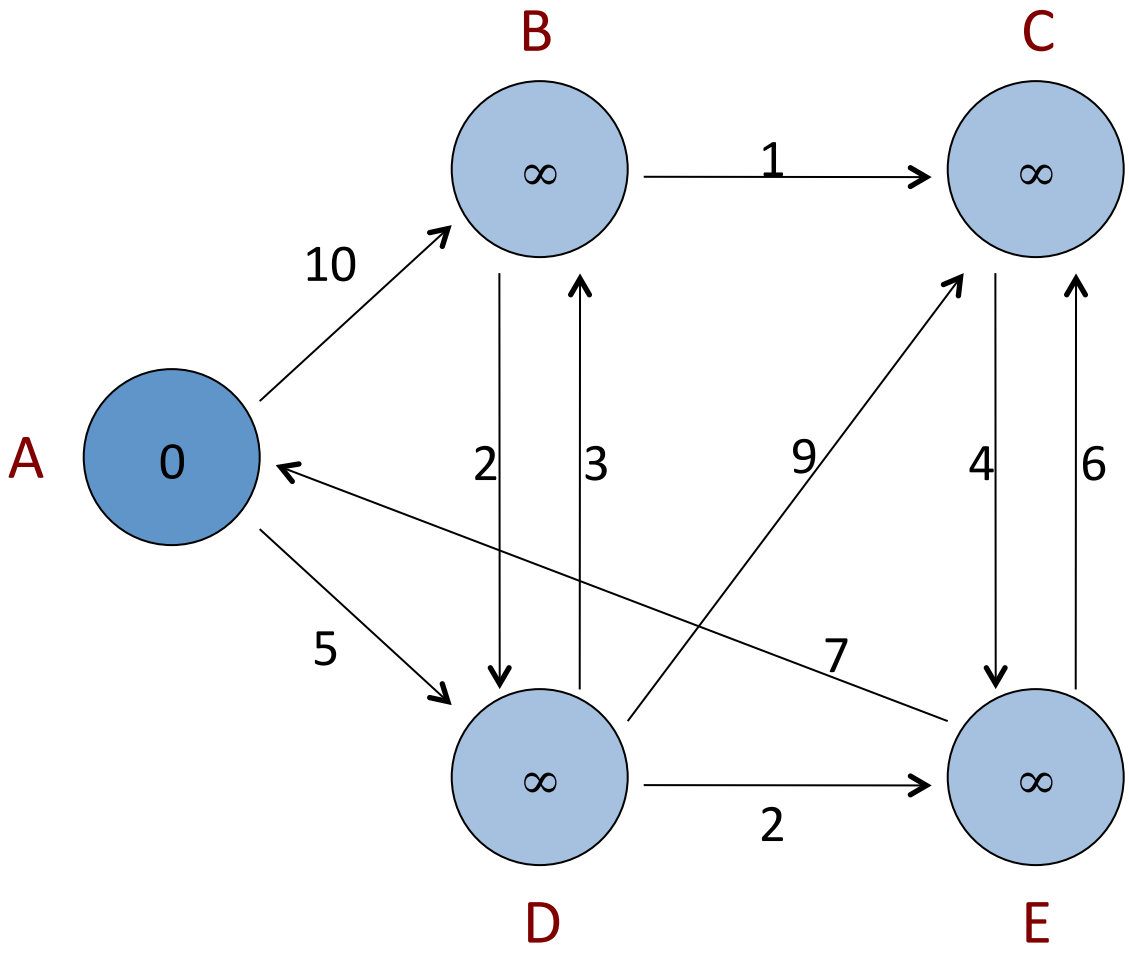
- Single Source Shortest Path (SSSP)
- SSSP MapReduce Implementation
- Pregel
  - Computation Model
  - SSSP Example
  - Writing a Pregel program
  - System Design
- MapReduce vs Pregel for Graph processing

Single Source Shortest Path (SSSP)

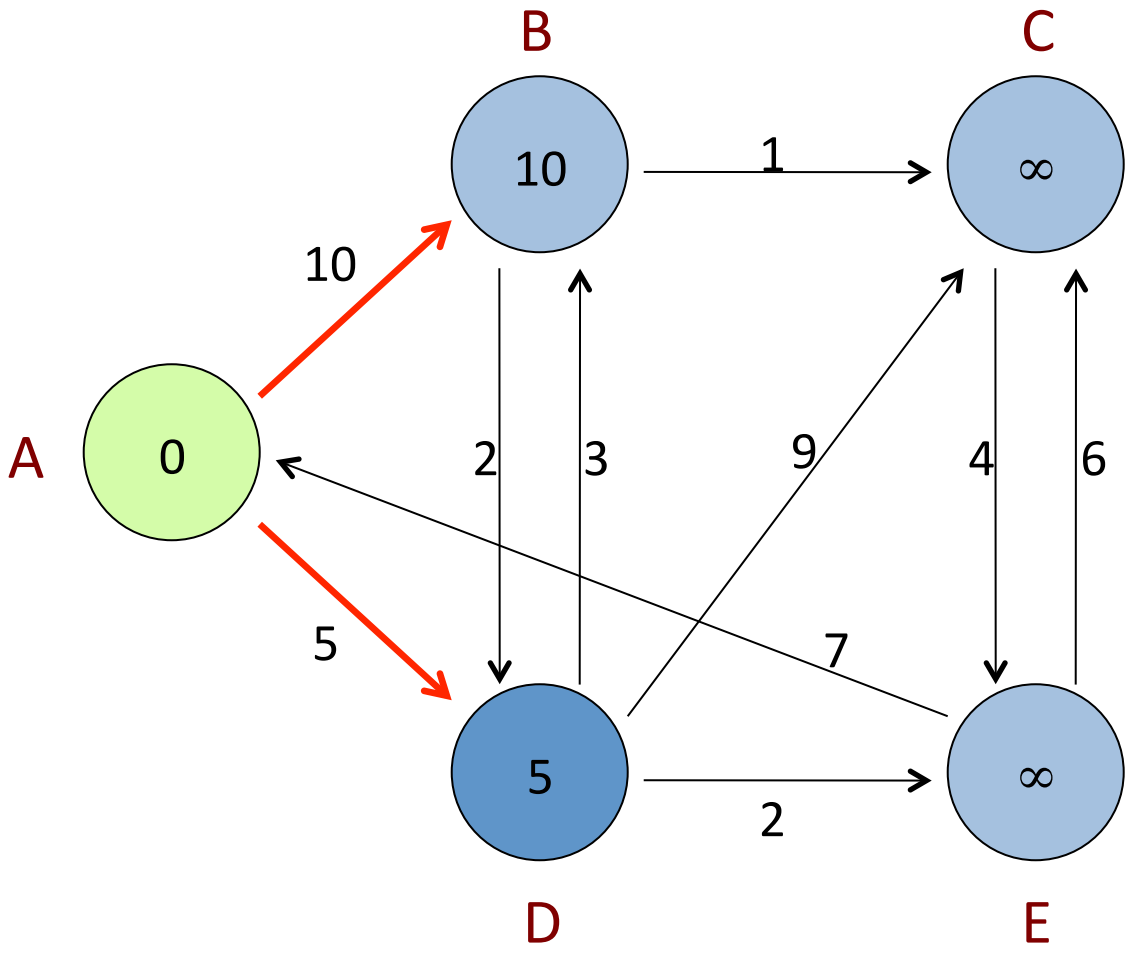
- Problem
  - Find shortest path from a source node to all target nodes

- Solution for single processor machine
  - Dijkstra's algorithm

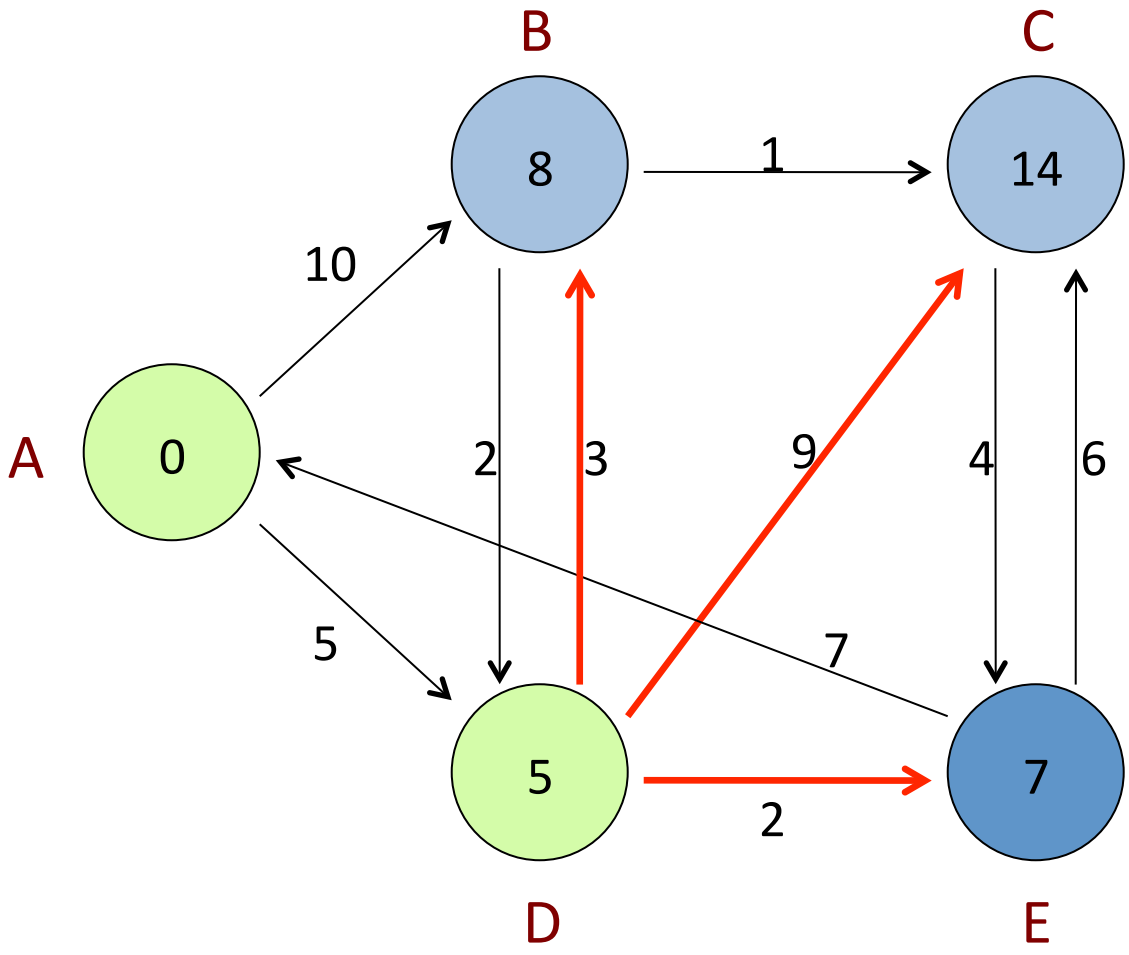
Example: SSSP – Dijkstra's Algorithm



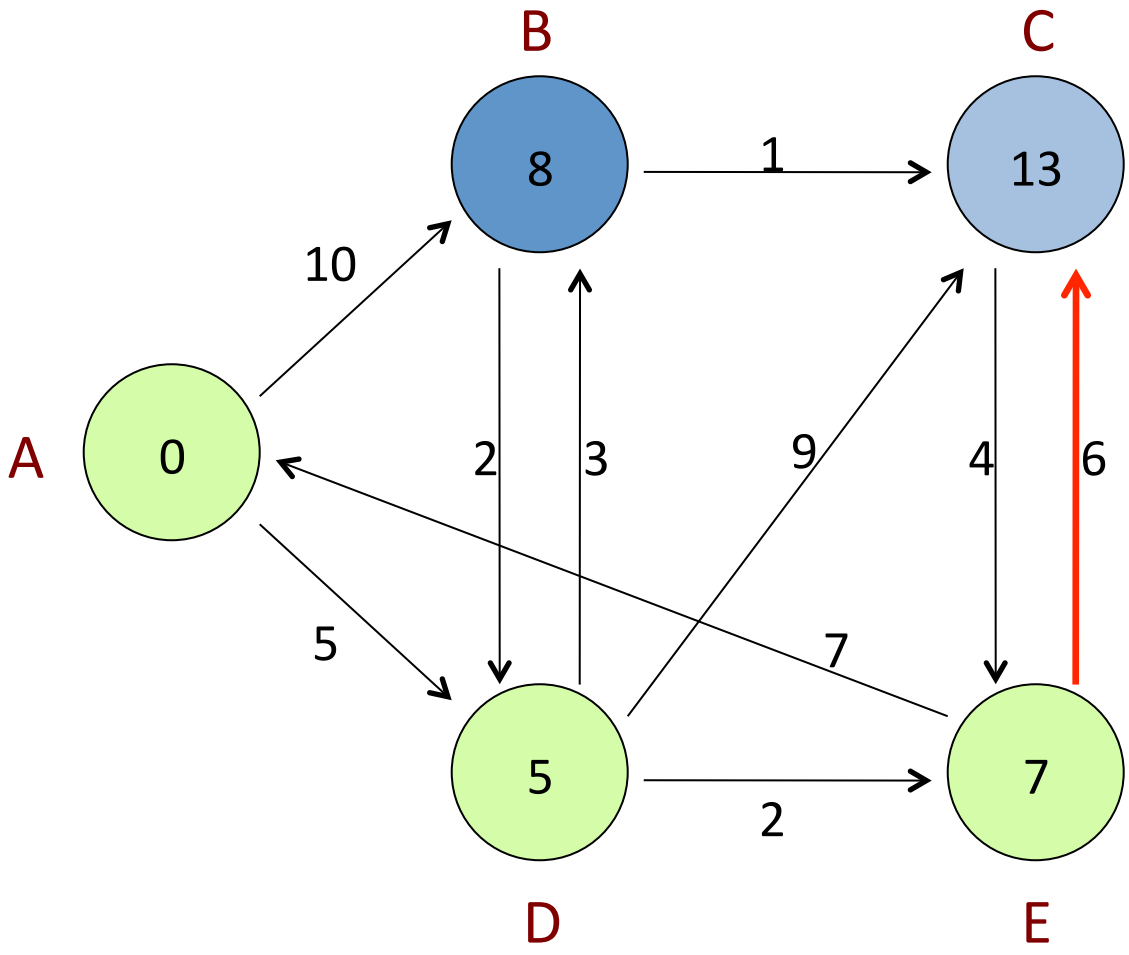
Example: SSSP – Dijkstra's Algorithm



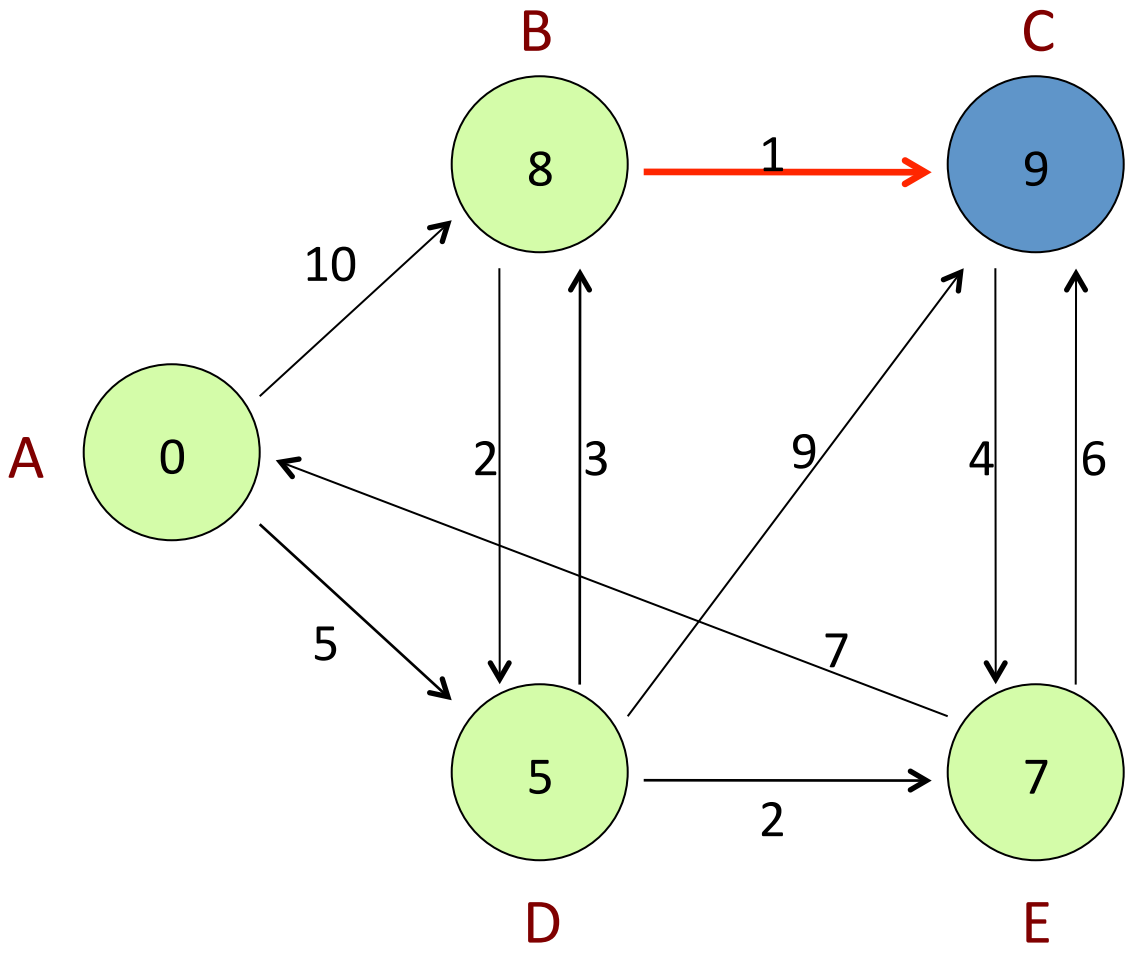
Example: SSSP – Dijkstra's Algorithm



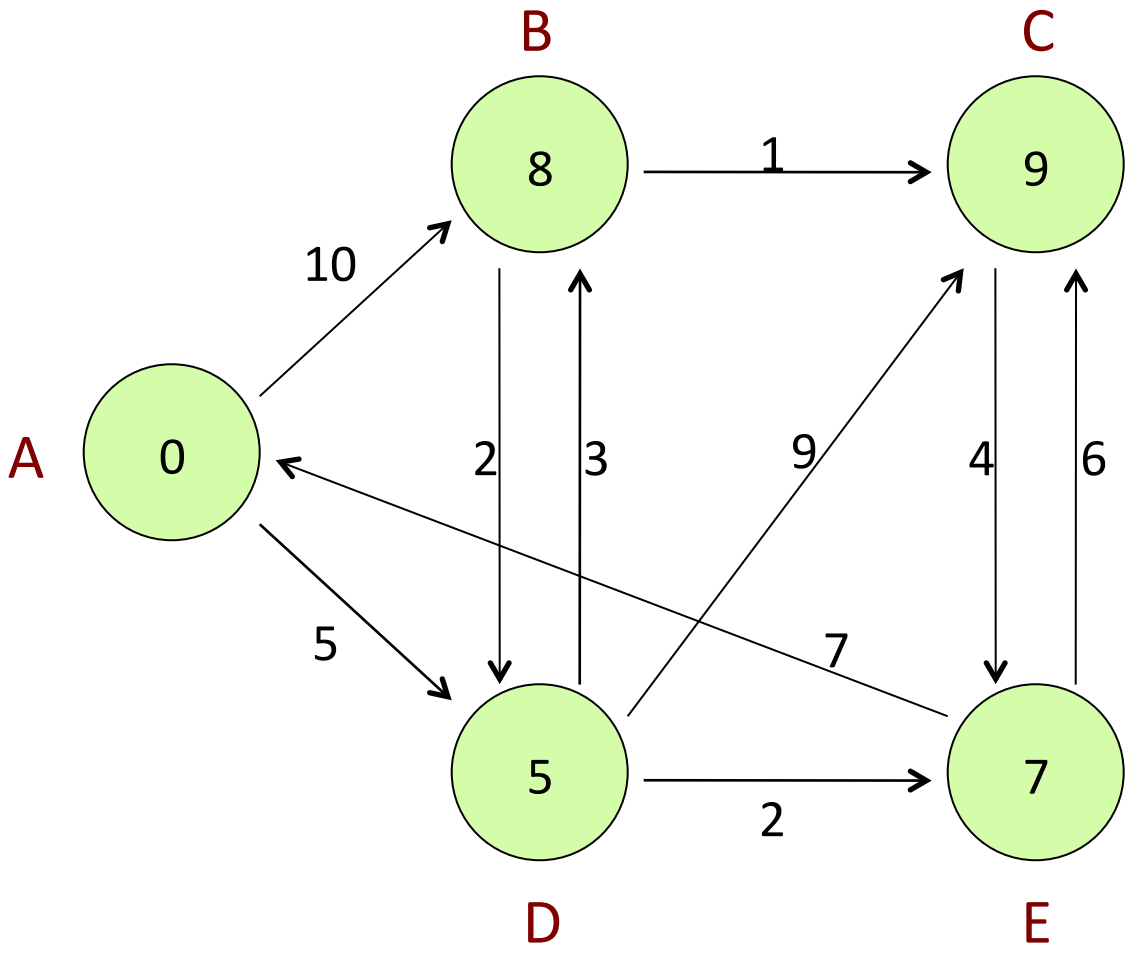
Example: SSSP – Dijkstra's Algorithm



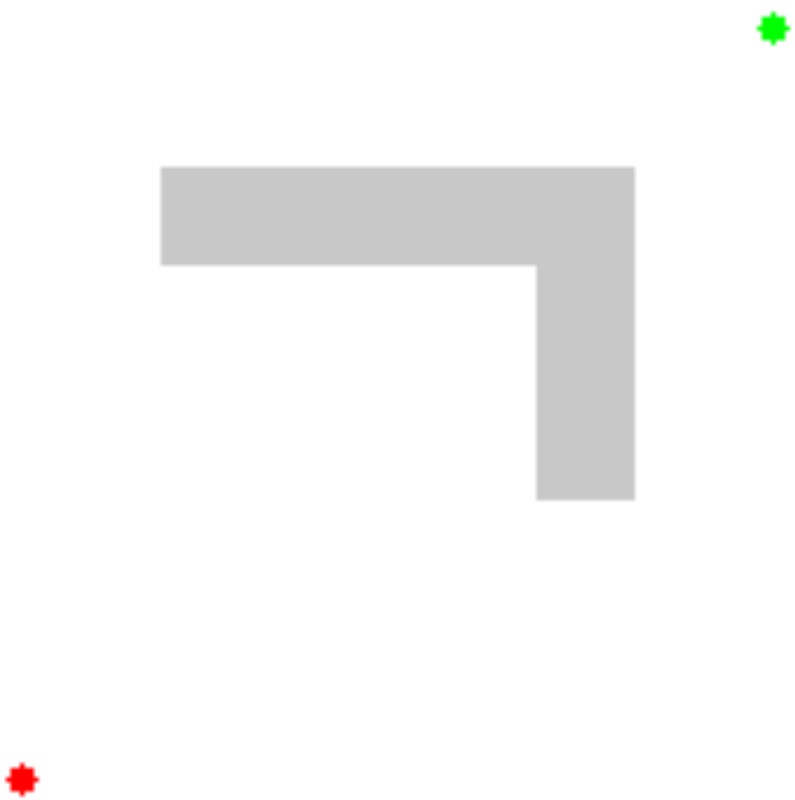
Example: SSSP – Dijkstra's Algorithm



Example: SSSP – Dijkstra's Algorithm



Example 2

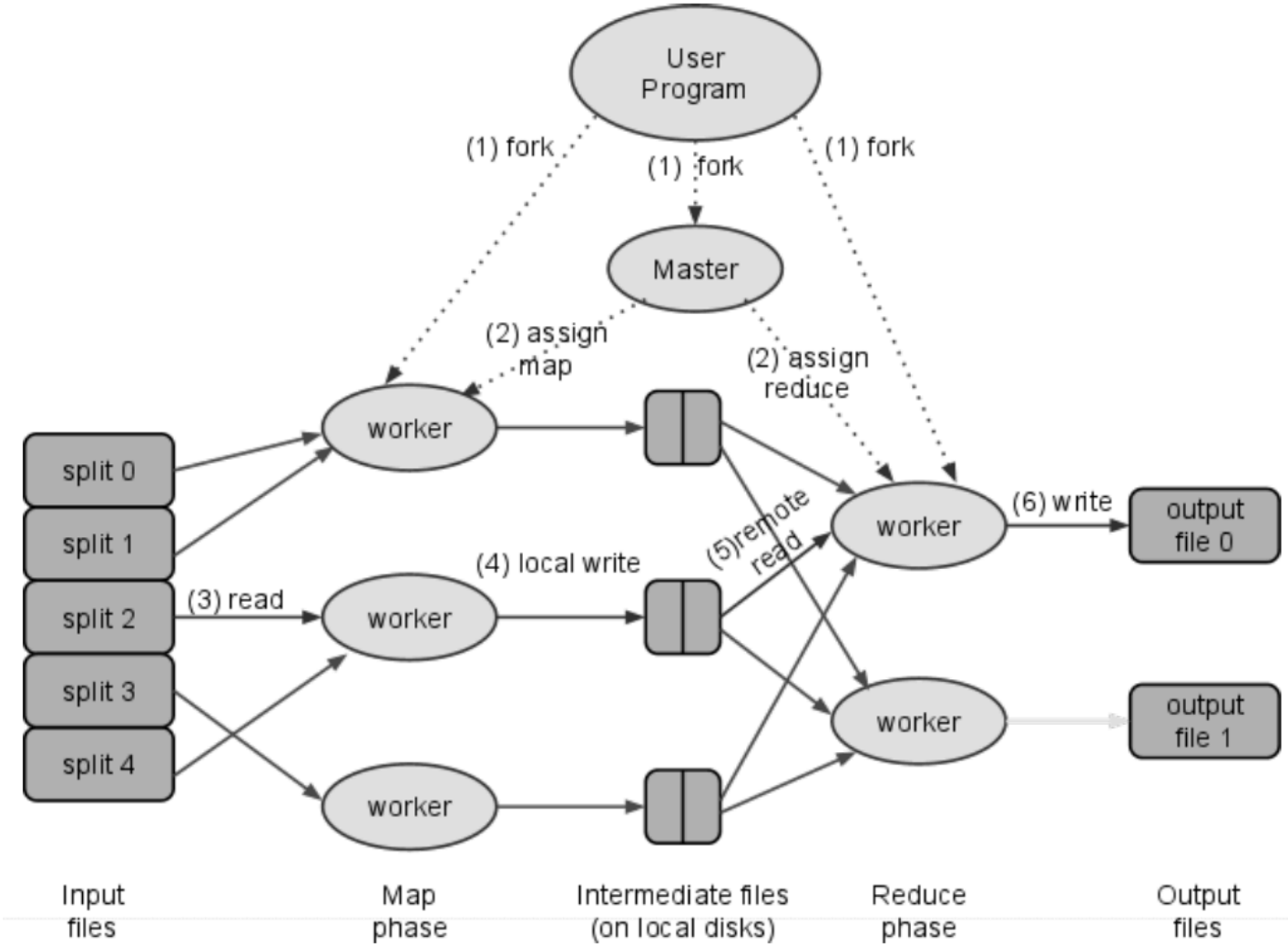


Single Source Shortest Path (SSSP)

- Problem
  - Find shortest path from a source node to all target nodes

- Solution on single processor machine
  - Dijkstra's algorithm
- **Distributed** solution: parallel breadth-first search
  - MapReduce
  - Pregel

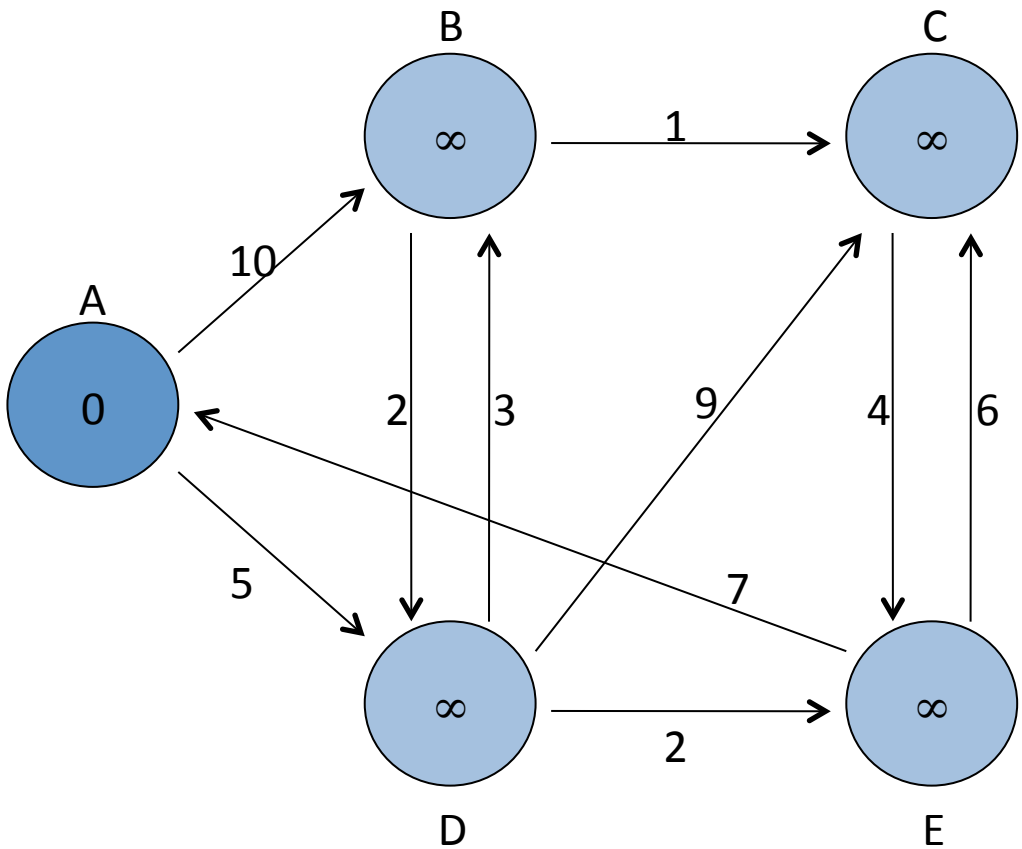
MapReduce Execution Overview



Example: SSSP – Parallel BFS in MapReduce

- Adjacency matrix

	A	B	C	D	E
A		10		5	
B			1	2	
C					4
D		3	9		2
E	7		6		



- Adjacency List

A: (B, 10), (D, 5)

B: (C, 1), (D, 2)

C: (E, 4)

D: (B, 3), (C, 9), (E, 2)

E: (A, 7), (C, 6)

Example: SSSP – Parallel BFS in MapReduce

- Map input: <node ID, <dist, adj list>>

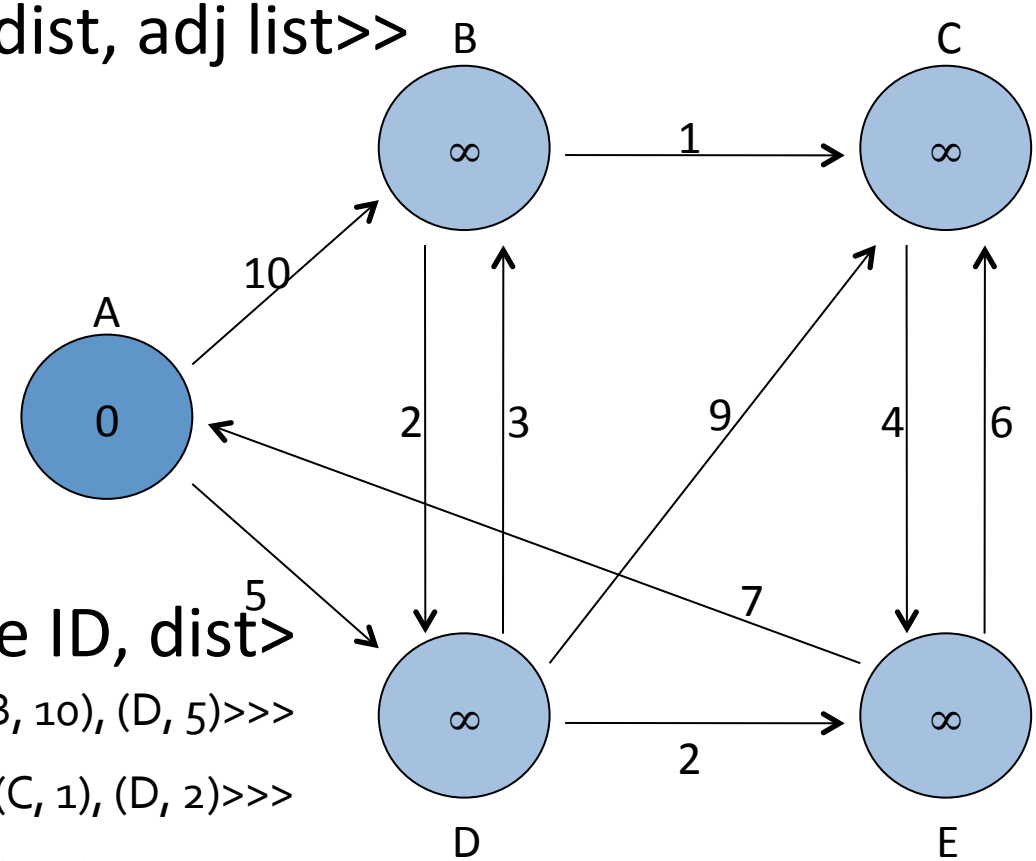
<A, <0, <(B, 10), (D, 5)>>>

<B, <inf, <(C, 1), (D, 2)>>>

<C, <inf, <(E, 4)>>>

<D, <inf, <(B, 3), (C, 9), (E, 2)>>>

<E, <inf, <(A, 7), (C, 6)>>>



- Map output: <dest node ID, dist>

<B, 10> <D, 5>

<C, inf> <D, inf>

<E, inf>

<B, inf> <C, inf> <E, inf>

<A, inf> <C, inf>

<A, <0, <(B, 10), (D, 5)>>>

<B, <inf, <(C, 1), (D, 2)>>>

<C, <inf, <(E, 4)>>>

<D, <inf, <(B, 3), (C, 9), (E, 2)>>>

<E, <inf, <(A, 7), (C, 6)>>>

Flushed to local disk!!

Example: SSSP – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<A, <0, <(B, 10), (D, 5)>>>

<A, inf>

<B, <inf, <(C, 1), (D, 2)>>>

<B, 10> <B, inf>

<C, <inf, <(E, 4)>>>

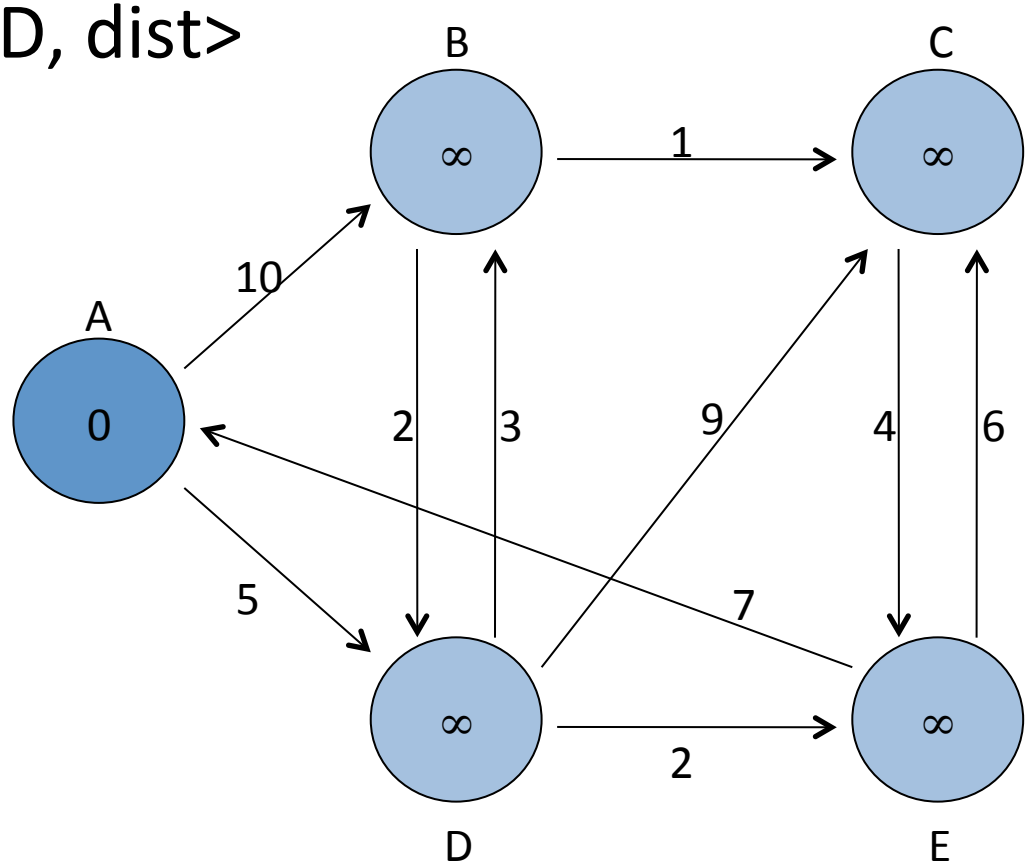
<C, inf> <C, inf> <C, inf>

<D, <inf, <(B, 3), (C, 9), (E, 2)>>>

<D, 5> <D, inf>

<E, <inf, <(A, 7), (C, 6)>>>

<E, inf> <E, inf>



Example: SSSP – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<A, <0, <(B, 10), (D, 5)>>>

~~<A, inf>~~

<B, <~~inf~~, <(C, 1), (D, 2)>>>

<B, 10> ~~<B, inf>~~

<C, <inf, <(E, 4)>>>

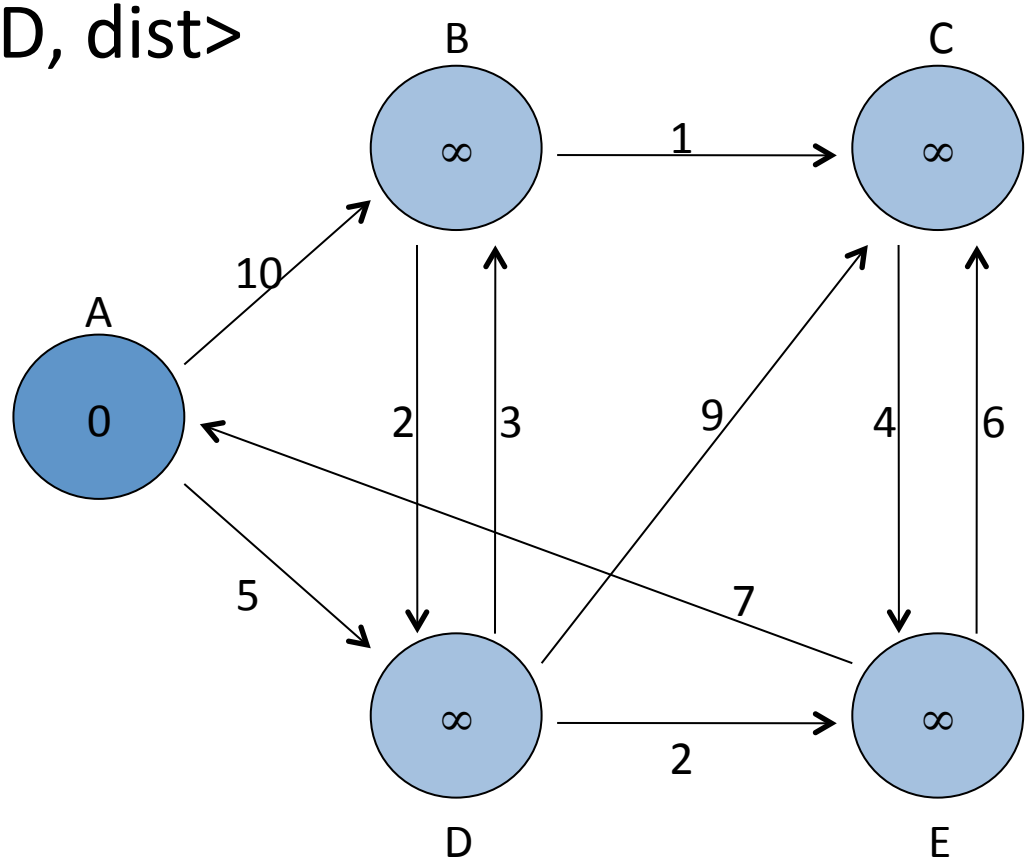
~~<C, inf> <C, inf> <C, inf>~~

<D, <~~inf~~, <(B, 3), (C, 9), (E, 2)>>>

<D, 5> ~~<D, inf>~~

<E, <inf, <(A, 7), (C, 6)>>>

~~<E, inf> <E, inf>~~



Example: SSSP – Parallel BFS in MapReduce

- Reduce output: <node ID, <dist, adj list>>
  = Map input for next iteration

<A, <0, <(B, 10), (D, 5)>>>

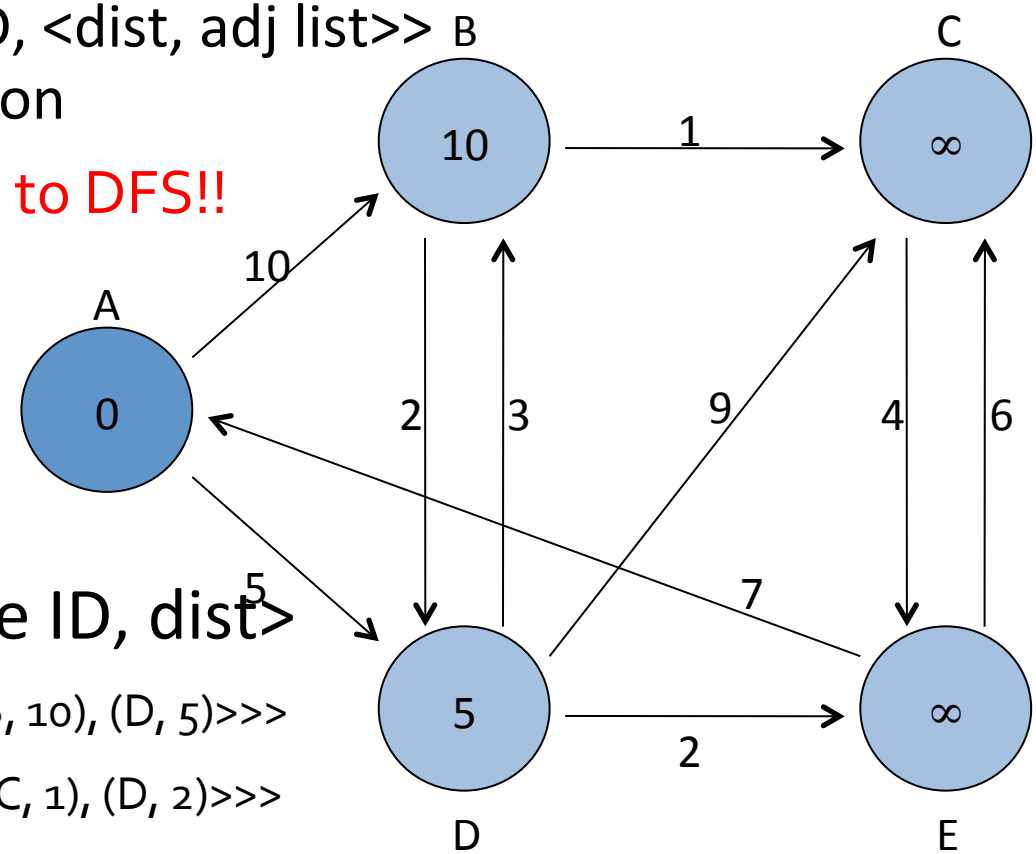
<B, <10, <(C, 1), (D, 2)>>>

<C, <inf, <(E, 4)>>>

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

<E, <inf, <(A, 7), (C, 6)>>>

Flushed to DFS!!



- Map output: <dest node ID, dist>

<B, 10> <D, 5>

<C, 11> <D, 12>

<E, inf>

<B, 8> <C, 14> <E, 7>

<A, inf> <C, inf>

<A, <0, <(B, 10), (D, 5)>>>

<B, <10, <(C, 1), (D, 2)>>>

<C, <inf, <(E, 4)>>>

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

<E, <inf, <(A, 7), (C, 6)>>>

Flushed to local disk!!

Example: SSSP – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<A, <0, <(B, 10), (D, 5)>>>

<A, inf>

<B, <10, <(C, 1), (D, 2)>>>

<B, 10> <B, 8>

<C, <inf, <(E, 4)>>>

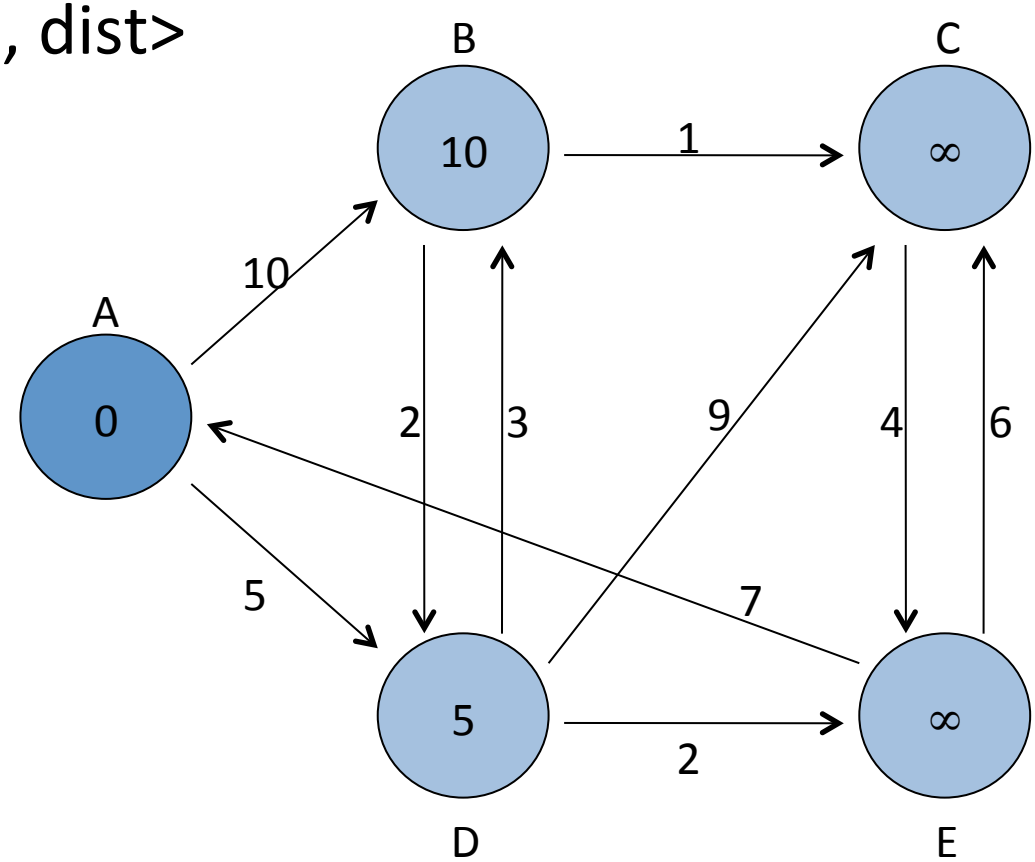
<C, 11> <C, 14> <C, inf>

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

<D, 5> <D, 12>

<E, <inf, <(A, 7), (C, 6)>>>

<E, inf> <E, 7>



Example: SSSP – Parallel BFS in MapReduce

- Reduce input: <node ID, dist>

<A, <0, <(B, 10), (D, 5)>>>

~~<A, inf>~~

<B, <~~10~~, <(C, 1), (D, 2)>>>

~~<B, 10>~~ <B, 8>

<C, <~~inf~~, <(E, 4)>>>

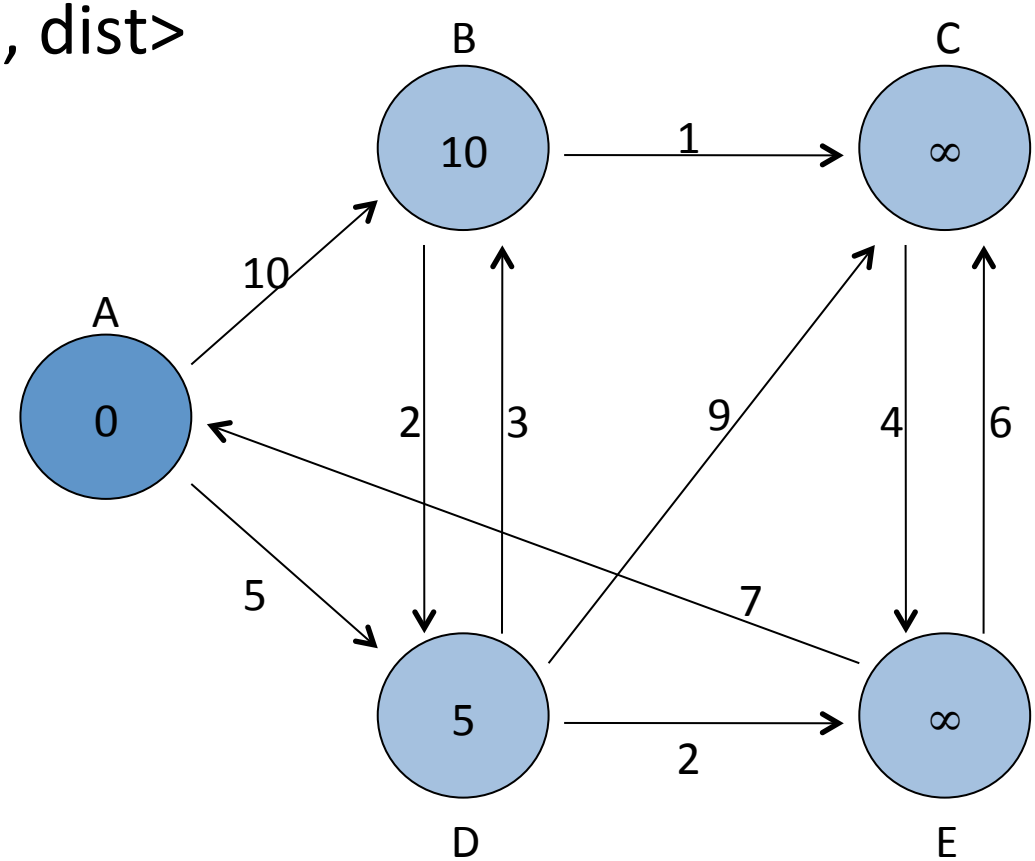
<C, 11> ~~<C, 14> <C, inf>~~

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

~~<D, 5> <D, 12>~~

<E, <~~inf~~, <(A, 7), (C, 6)>>>

~~<E, inf>~~ <E, 7>



Example: SSSP – Parallel BFS in MapReduce

- Reduce output: <node ID, <dist, adj list>>
  = Map input for next iteration

<A, <0, <(B, 10), (D, 5)>>>

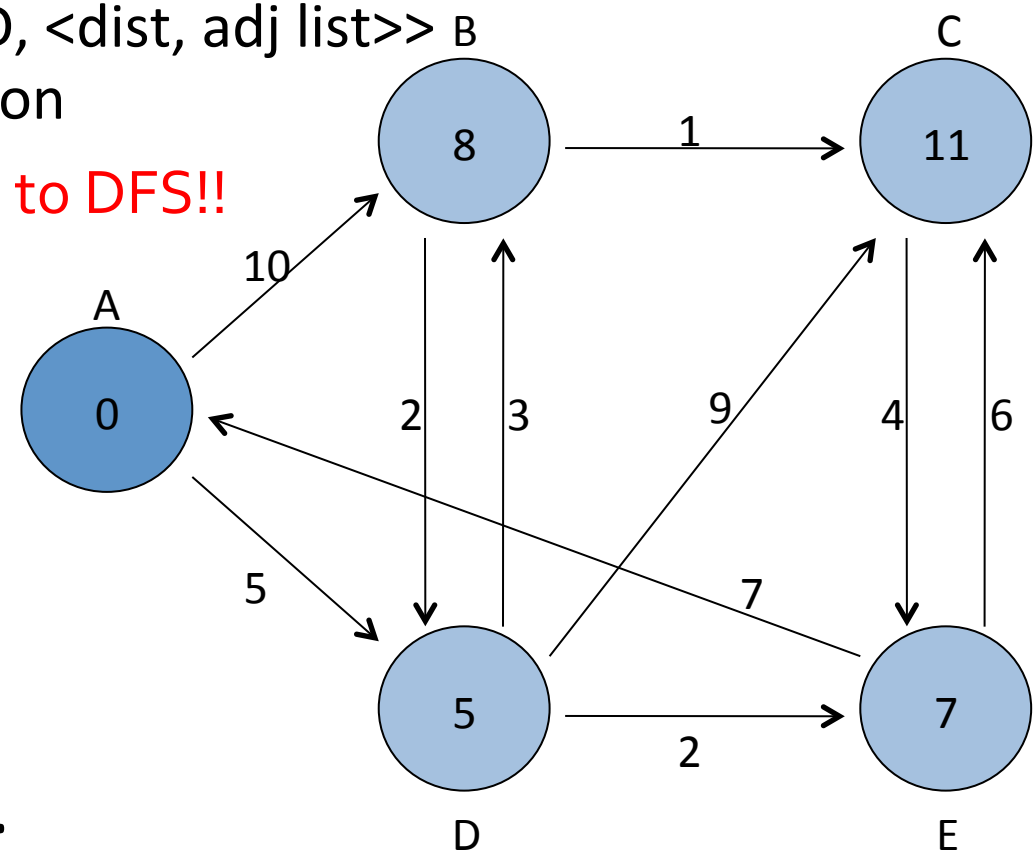
<B, <8, <(C, 1), (D, 2)>>>

<C, <11, <(E, 4)>>>

<D, <5, <(B, 3), (C, 9), (E, 2)>>>

<E, <7, <(A, 7), (C, 6)>>>

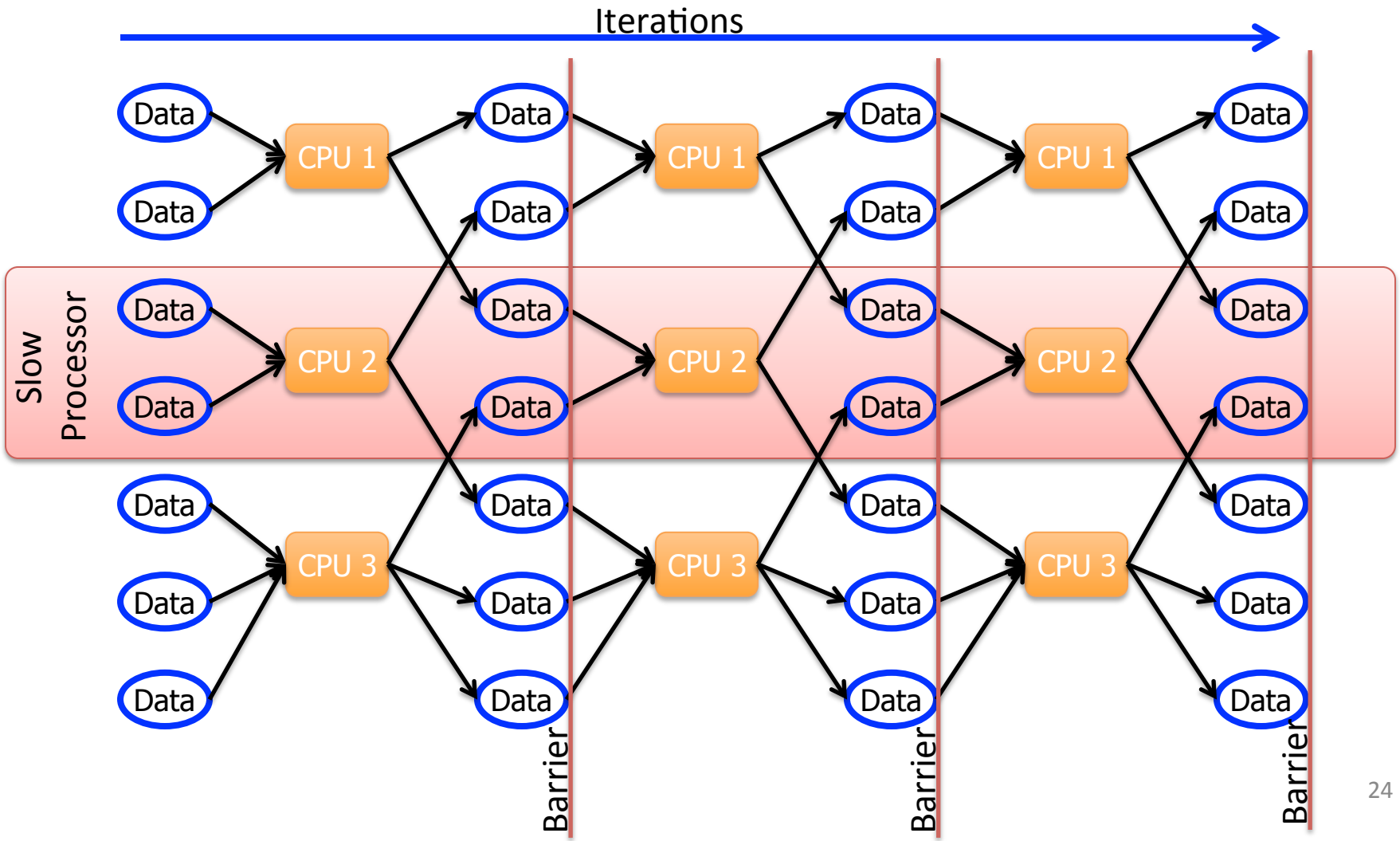
Flushed to DFS!!



… the rest omitted …

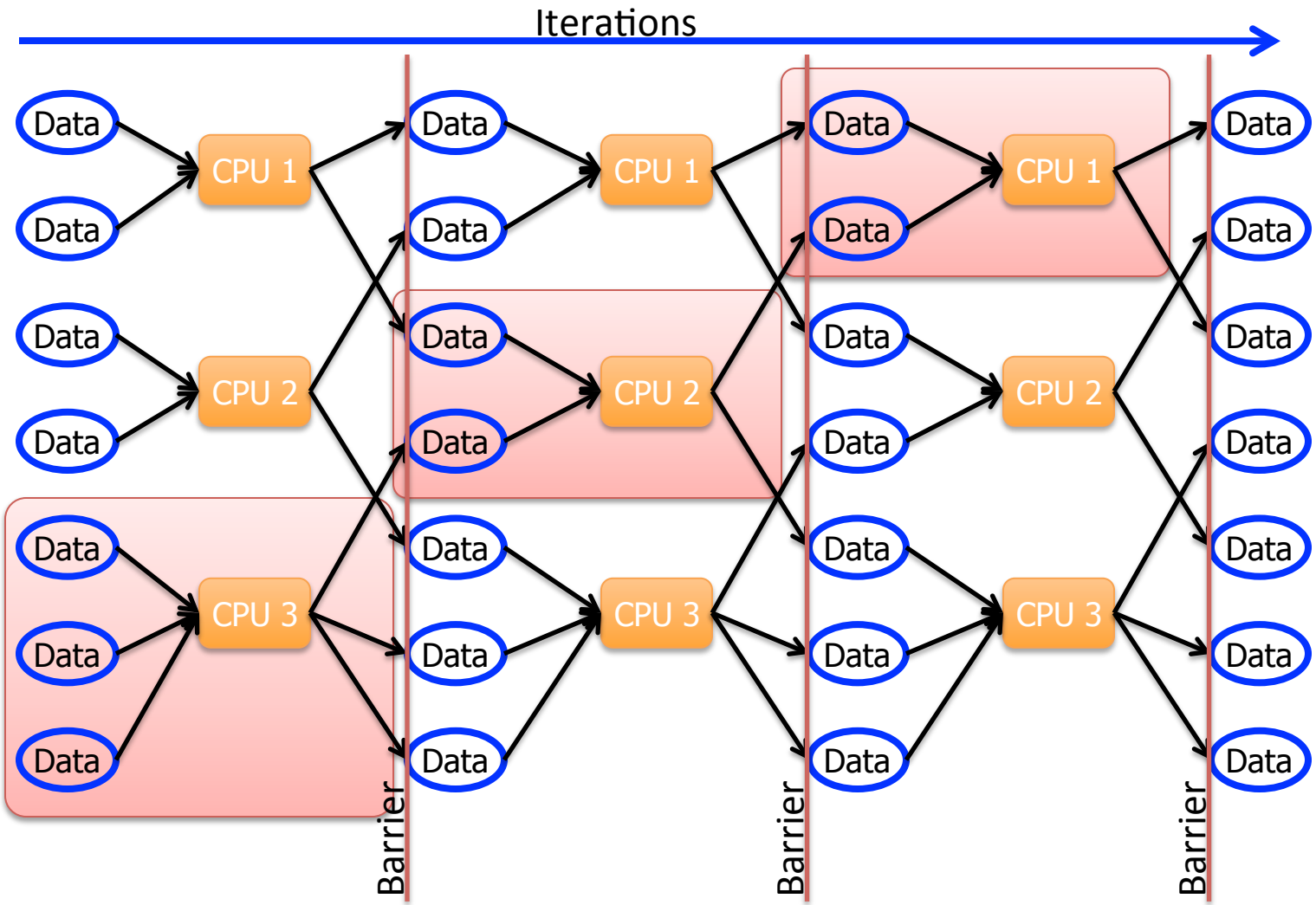
Problems with MapReduce

- Map-Reduce cannot express iterative algorithms efficiently :



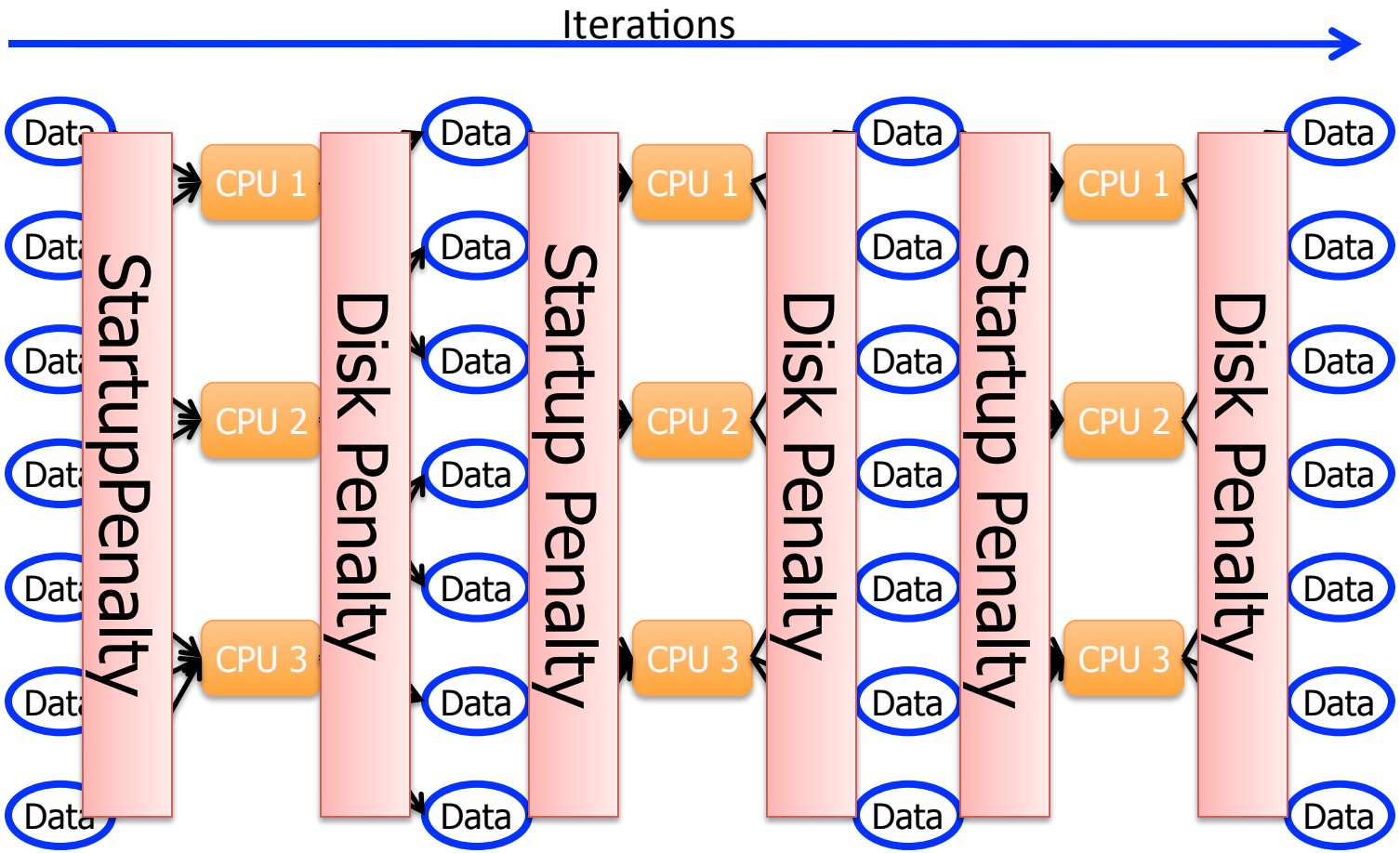
MapAbuse: Iterative MapReduce

- Only a subset of data needs computation:



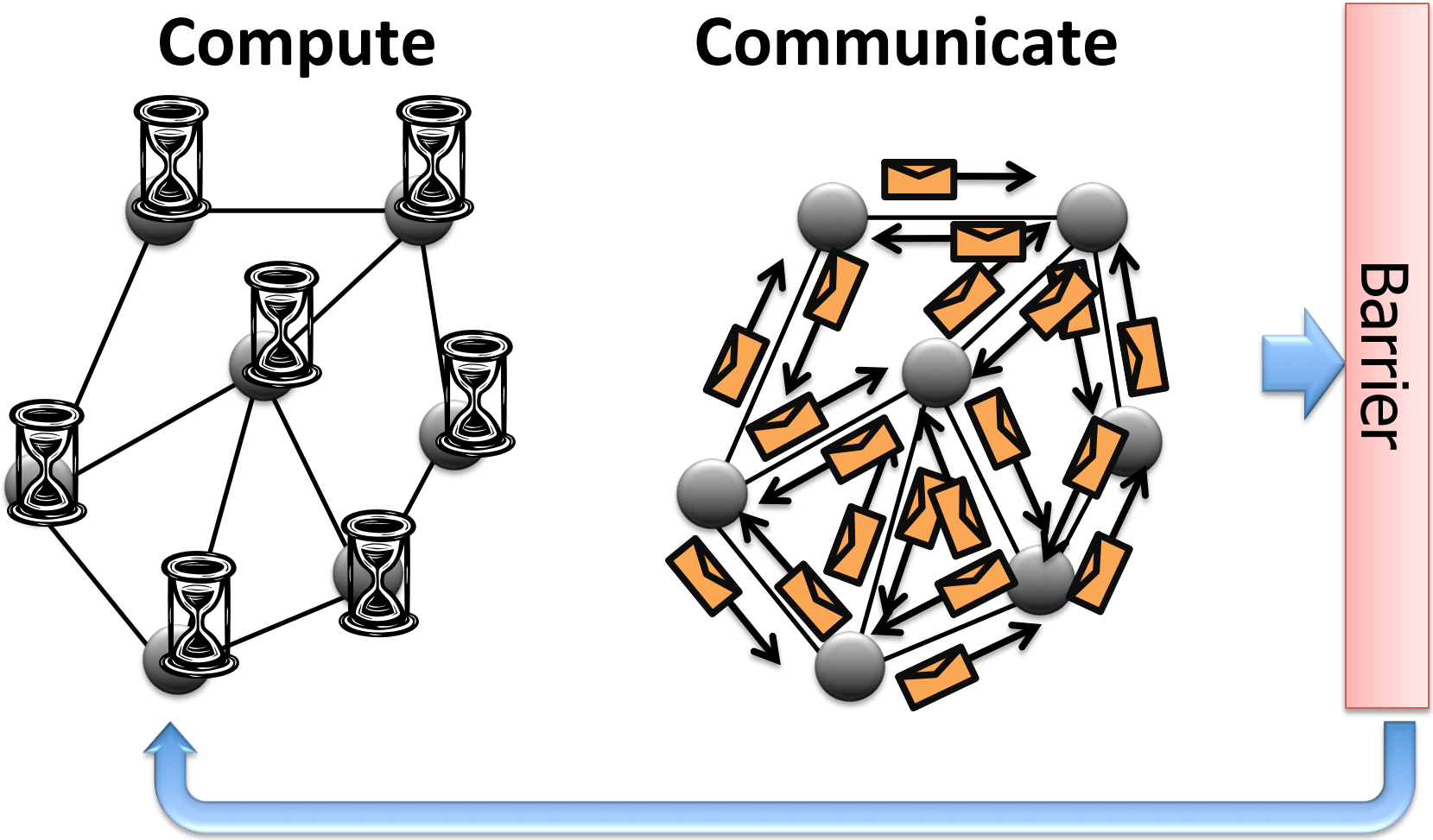
MapAbuse: Iterative MapReduce

- System is not optimized for iteration:

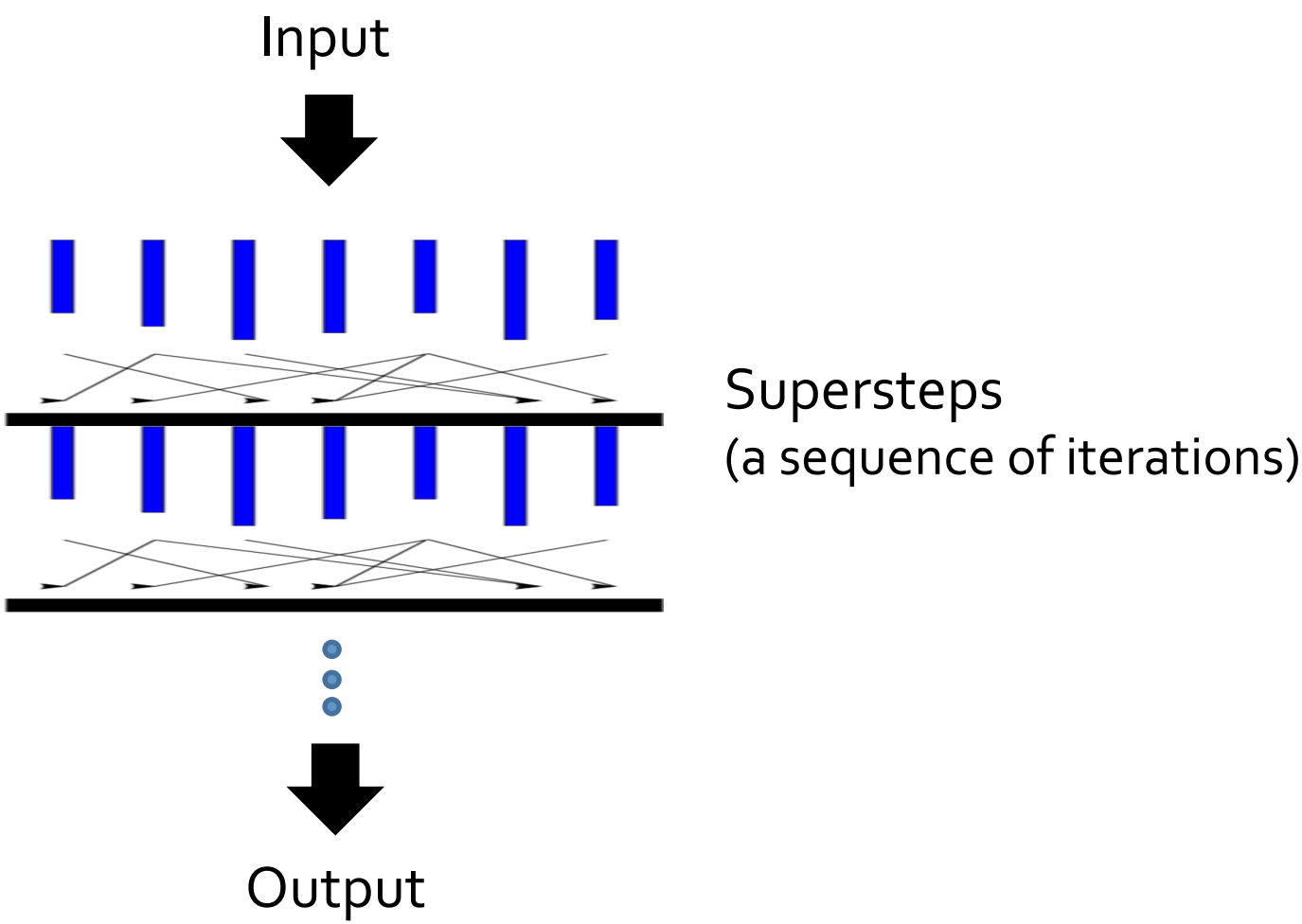


Pregel Model

- Bulk Synchronous Parallel Model:

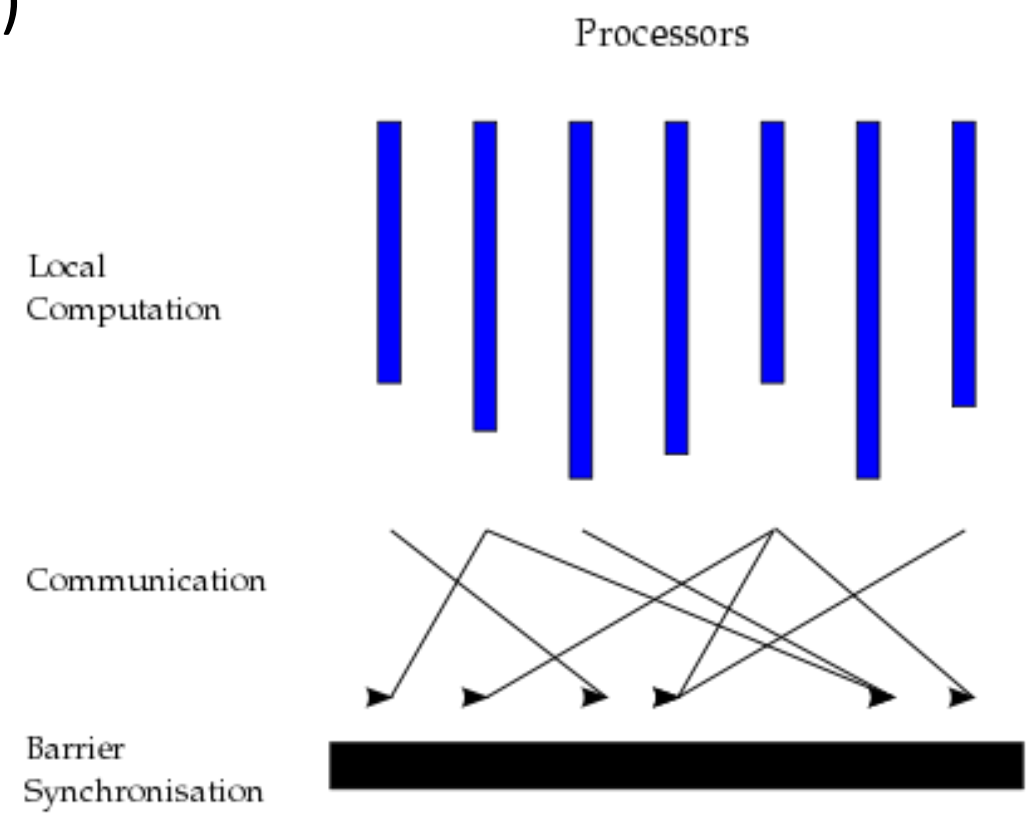


Computation Model



Computation Model

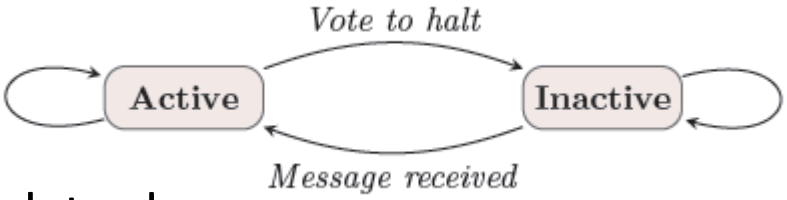
- "Think like a vertex"
- Inspired by Valiant's Bulk Synchronous Parallel model (1990)



Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Computation Model

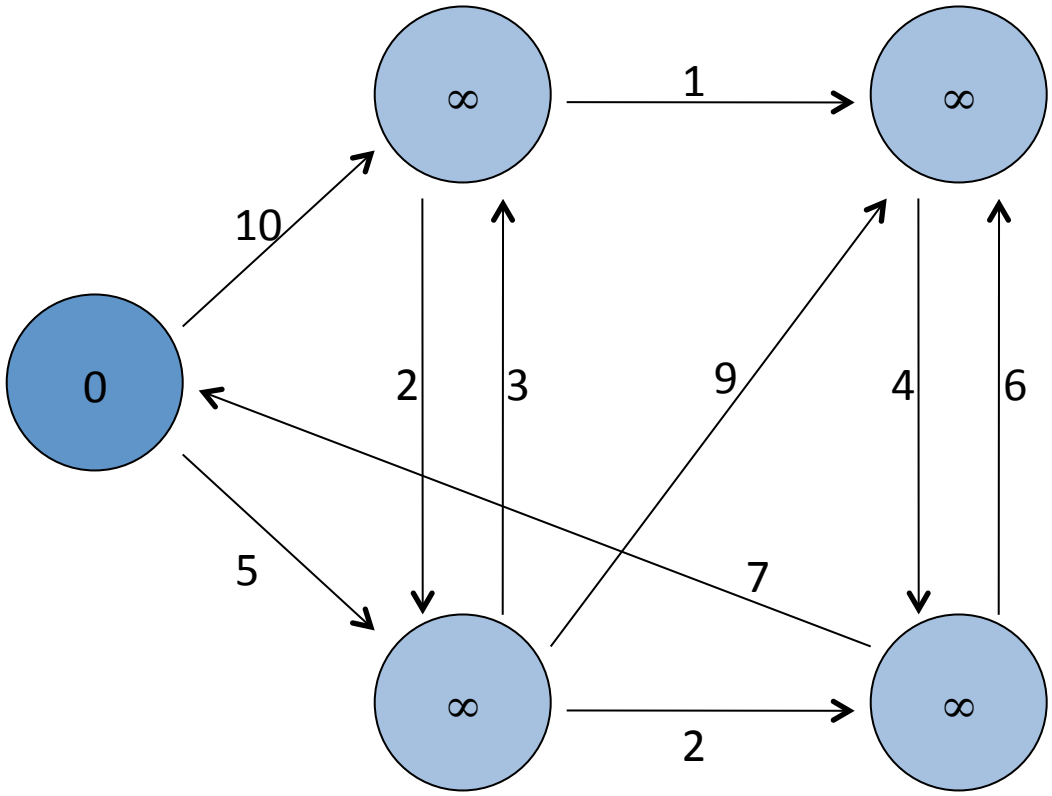
- Superstep: the vertices compute in parallel
  - Each vertex
    - Receives messages sent in the previous superstep
    - Executes the same user-defined function
    - Modifies its value or that of its outgoing edges
    - Sends messages to other vertices (to be received in the next superstep)
    - Mutates the topology of the graph
    - Votes to halt if it has no further work to do



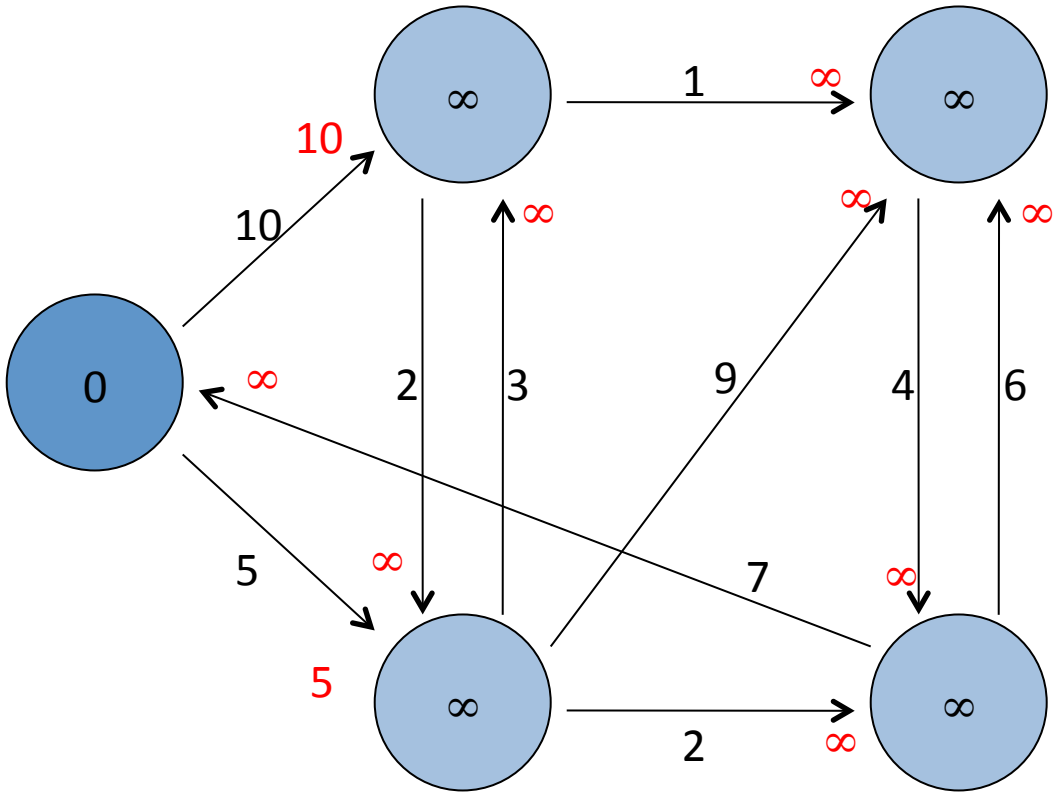
  - Termination condition
    - All vertices are simultaneously inactive
    - There are no messages in transit

Intermission

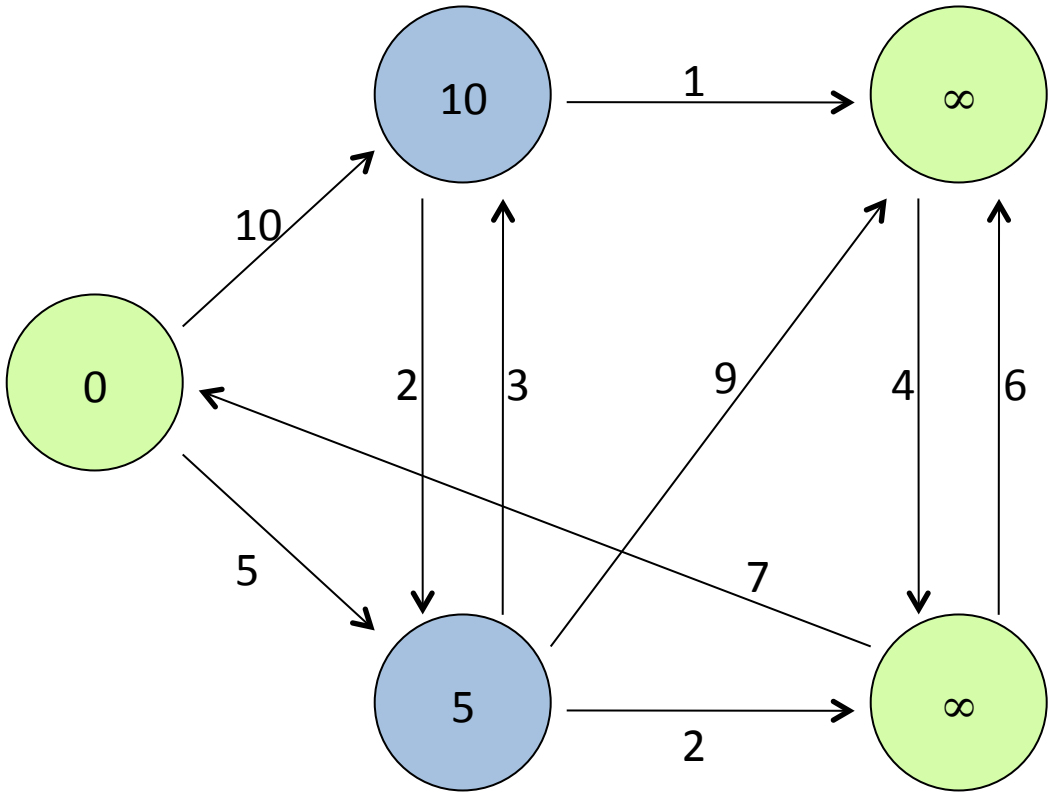
Example: SSSP – Parallel BFS in Pregel



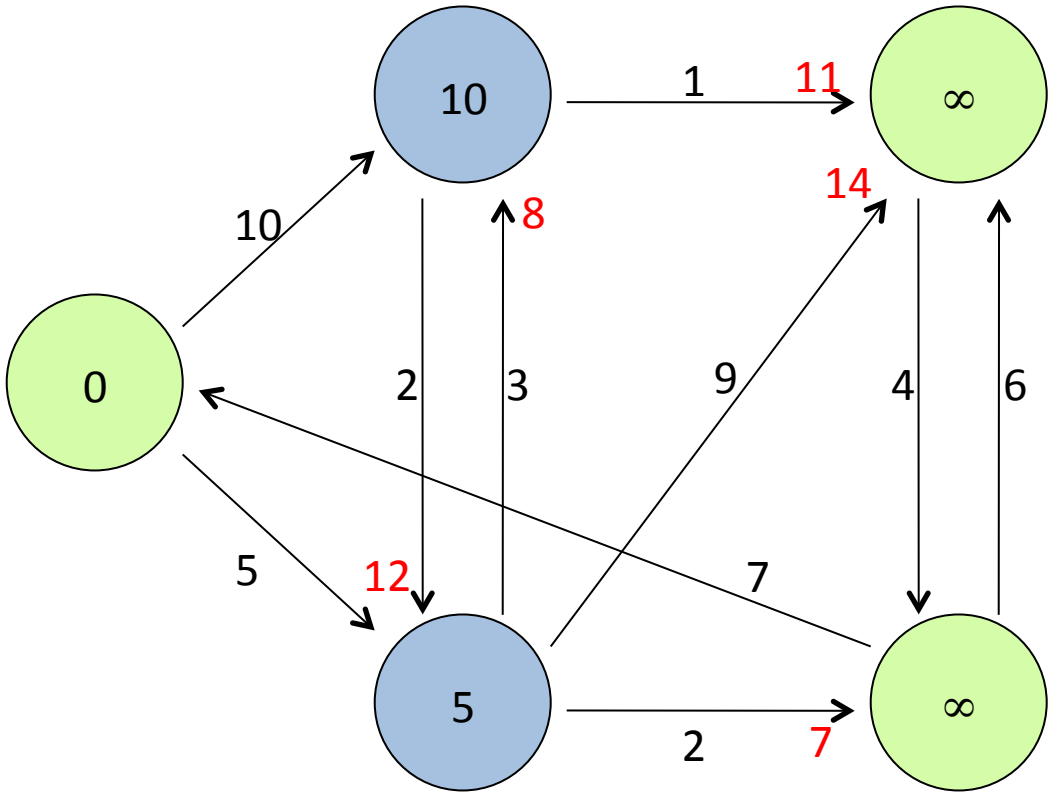
Example: SSSP – Parallel BFS in Pregel



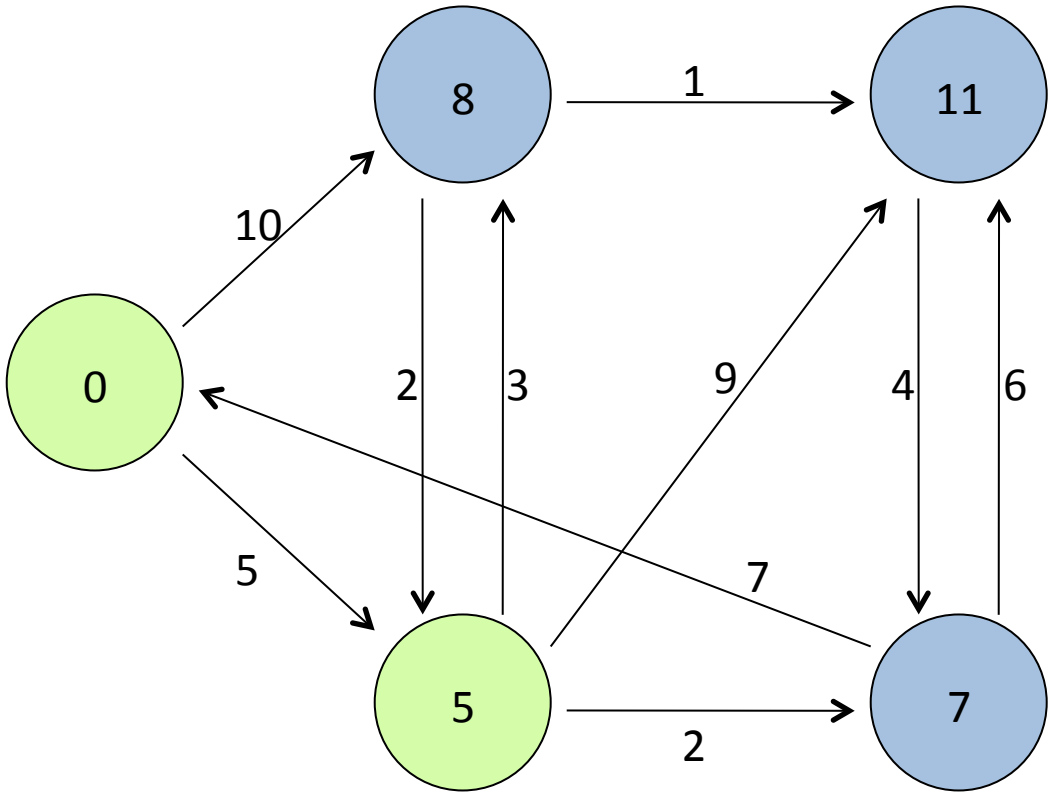
Example: SSSP – Parallel BFS in Pregel



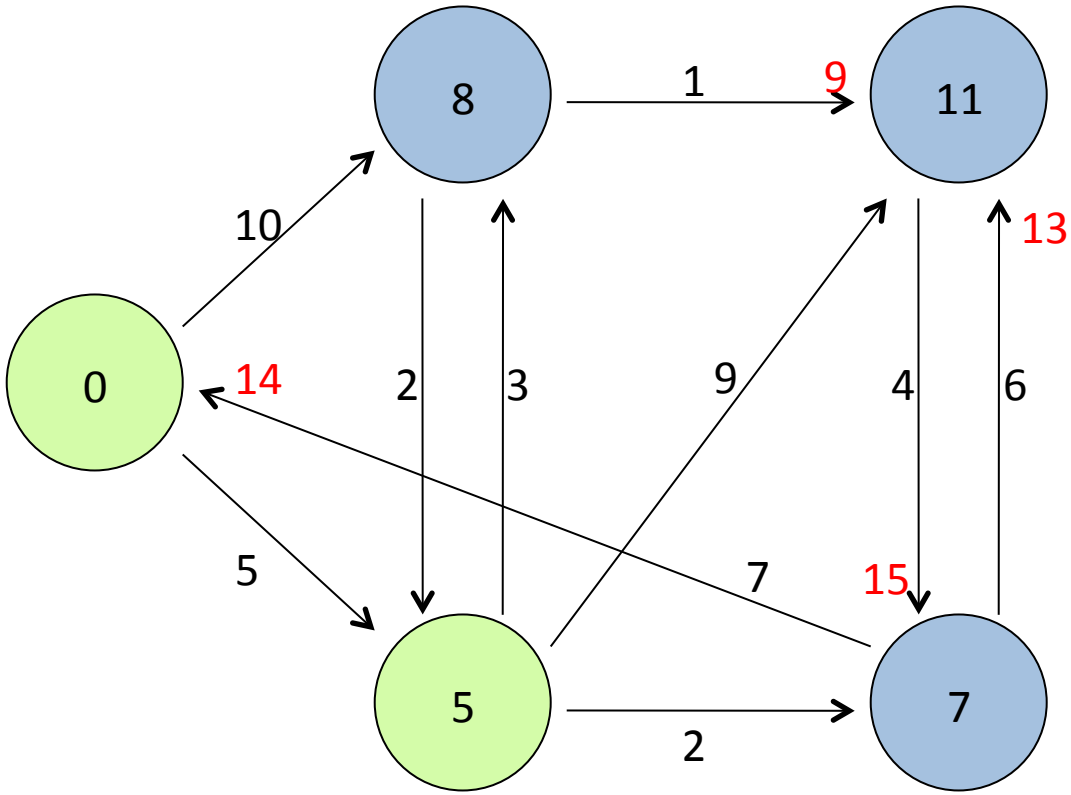
Example: SSSP – Parallel BFS in Pregel



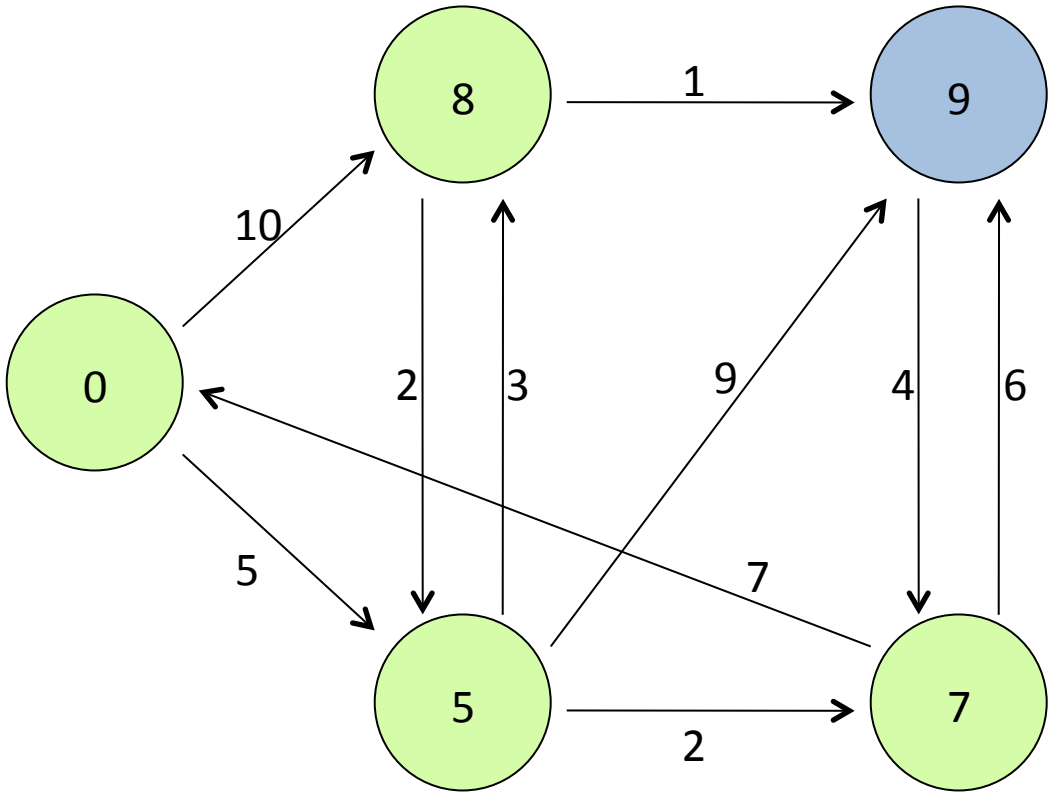
Example: SSSP – Parallel BFS in Pregel



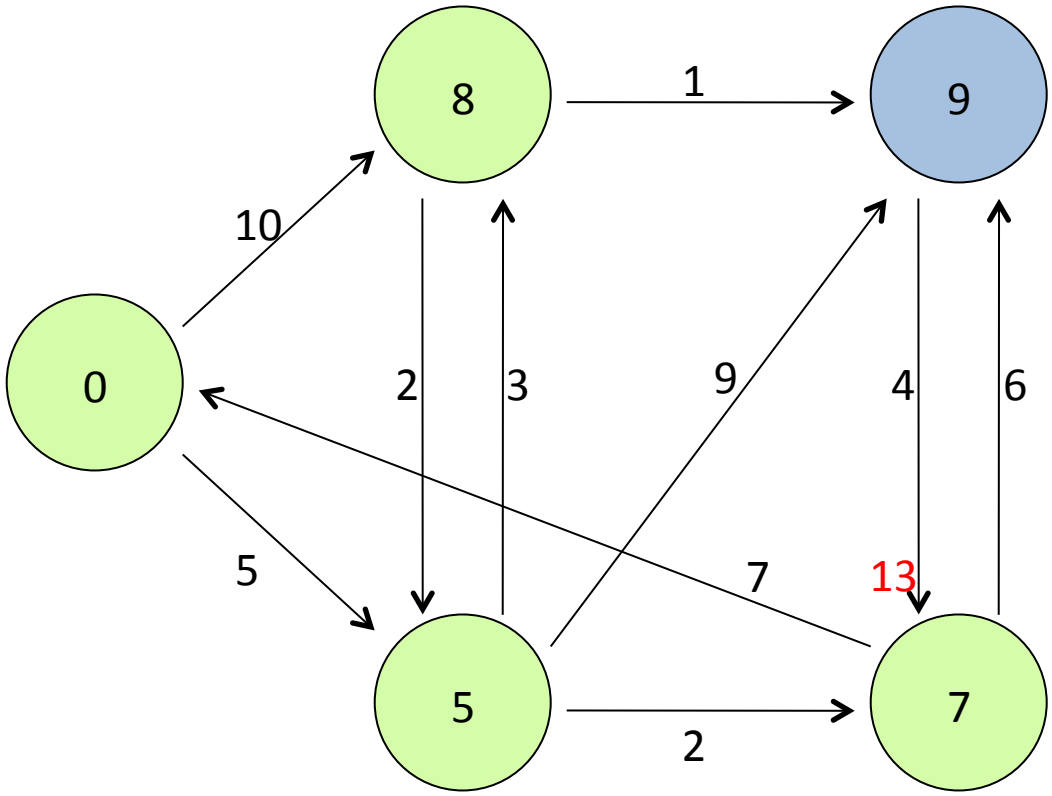
Example: SSSP – Parallel BFS in Pregel



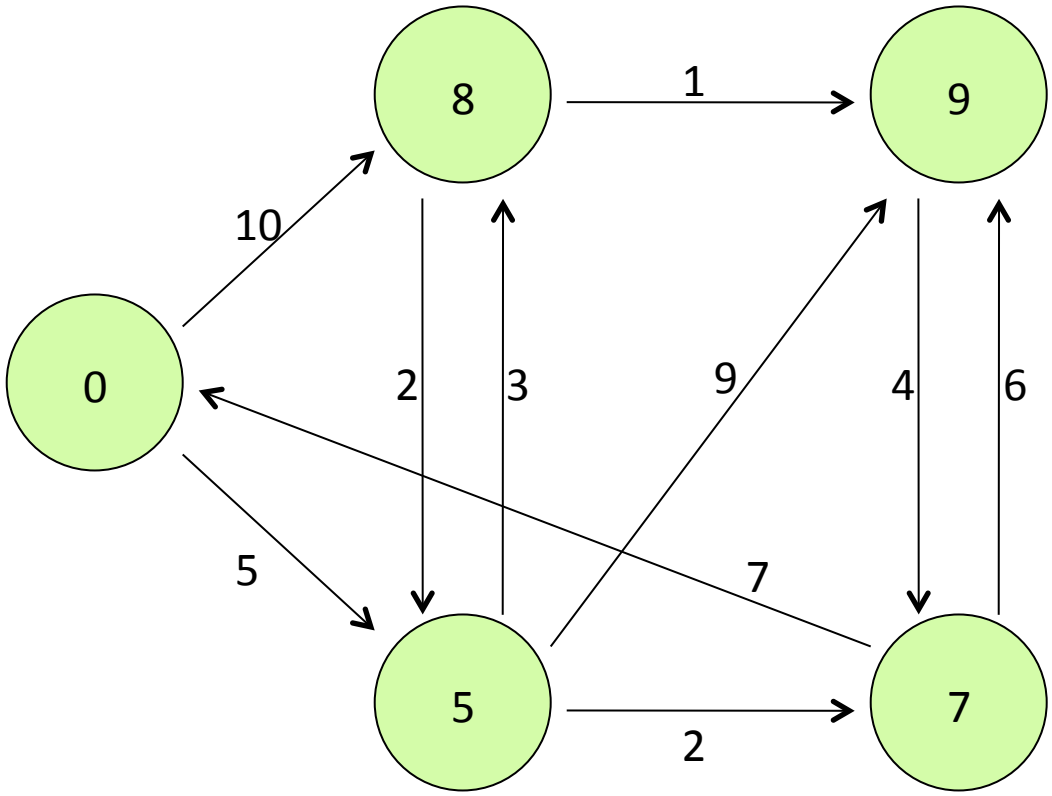
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



C++ API

- Writing a Pregel program
  - Subclassing the predefined **Vertex** class

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

Override this!

in msgs

out msg

Example: Vertex Class for SSSP

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

System Design

- Pregel system also uses the master/worker model
  - Master
    - Maintains worker
    - Recovers faults of workers
    - Provides Web-UI monitoring tool of job progress
  - Worker
    - Processes its task
    - Communicates with the other workers
- Persistent data is stored as files on a distributed storage system (such as GFS or BigTable)
- Temporary data is stored on local disk

Execution of a Pregel Program

1. Many copies of the program begin executing on a cluster
2. The master assigns a partition of input to each worker
   - Each worker loads the vertices and marks them as active
3. The master instructs each worker to perform a superstep
   - Each worker loops through its active vertices & computes for each vertex
   - Messages are sent asynchronously, but are delivered before the end of the superstep
   - This step is repeated as long as any vertices are active, or any messages are in transit
4. After the computation halts, the master may instruct each worker to save its portion of the graph

Fault Tolerance

- Checkpointing
  - The master periodically instructs the workers to save the state of their partitions to persistent storage
    - e.g., Vertex values, edge values, incoming messages
- Failure detection
  - Using regular "ping" messages
- Recovery
  - The master reassigns graph partitions to the currently available workers
  - The workers all reload their partition state from most recent available checkpoint

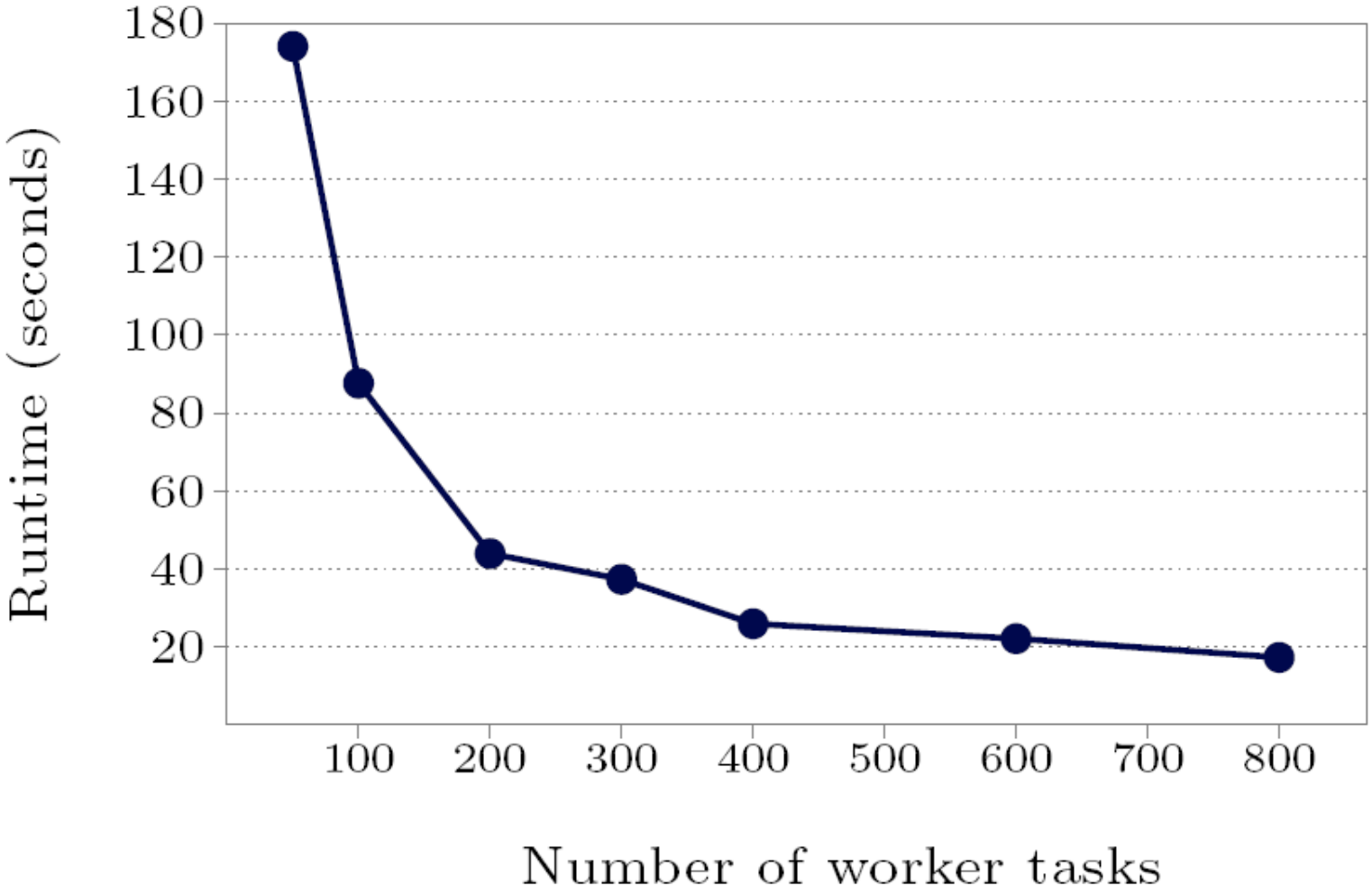
Experiments

- Environment
  - H/W: A cluster of 300 multicore commodity PCs
  - Data: binary trees, log-normal random graphs (general graphs)

- Naïve SSSP implementation
  - The weight of all edges = 1
  - No checkpointing

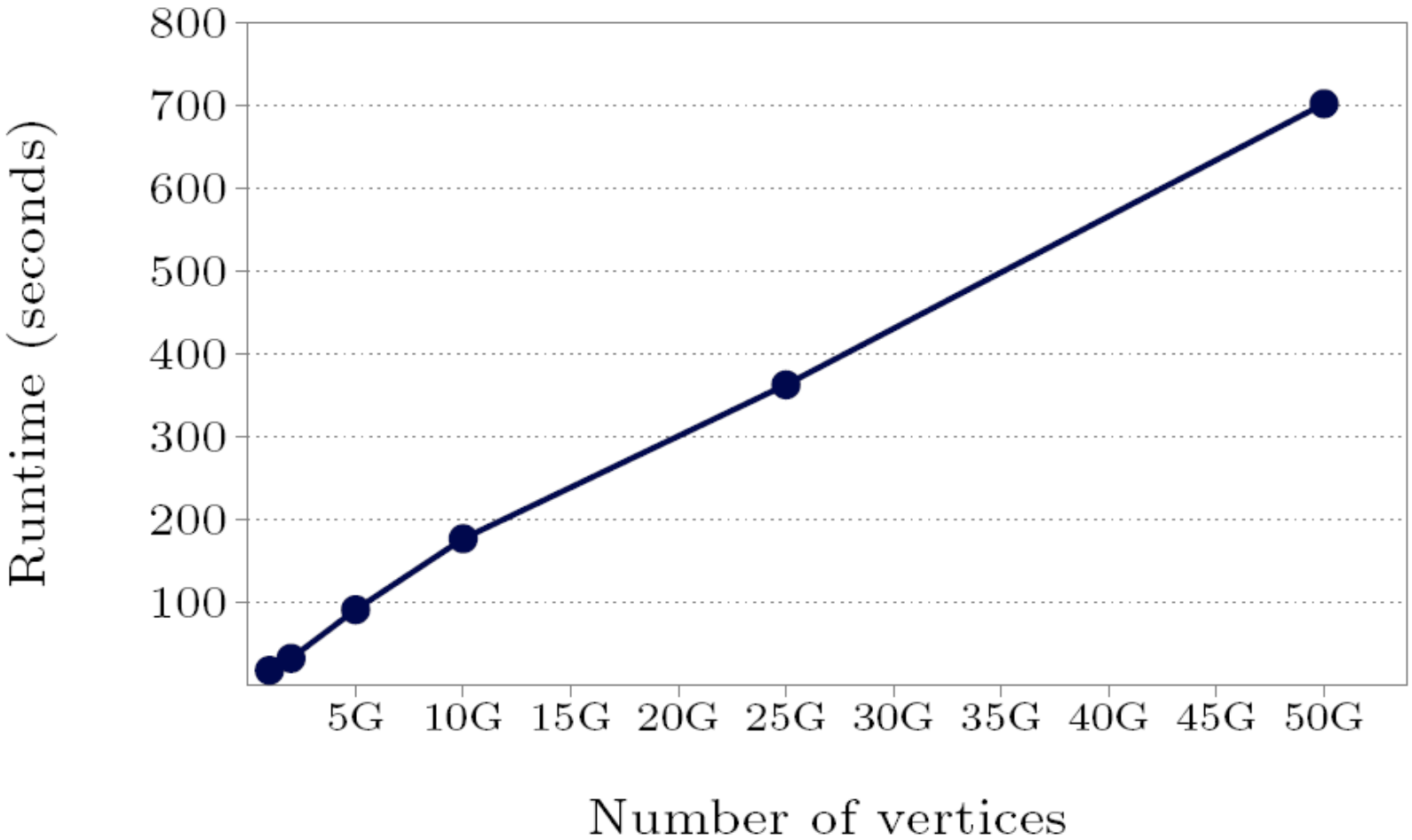
Experiments

- SSSP – 1 billion vertex binary tree: varying # of worker tasks



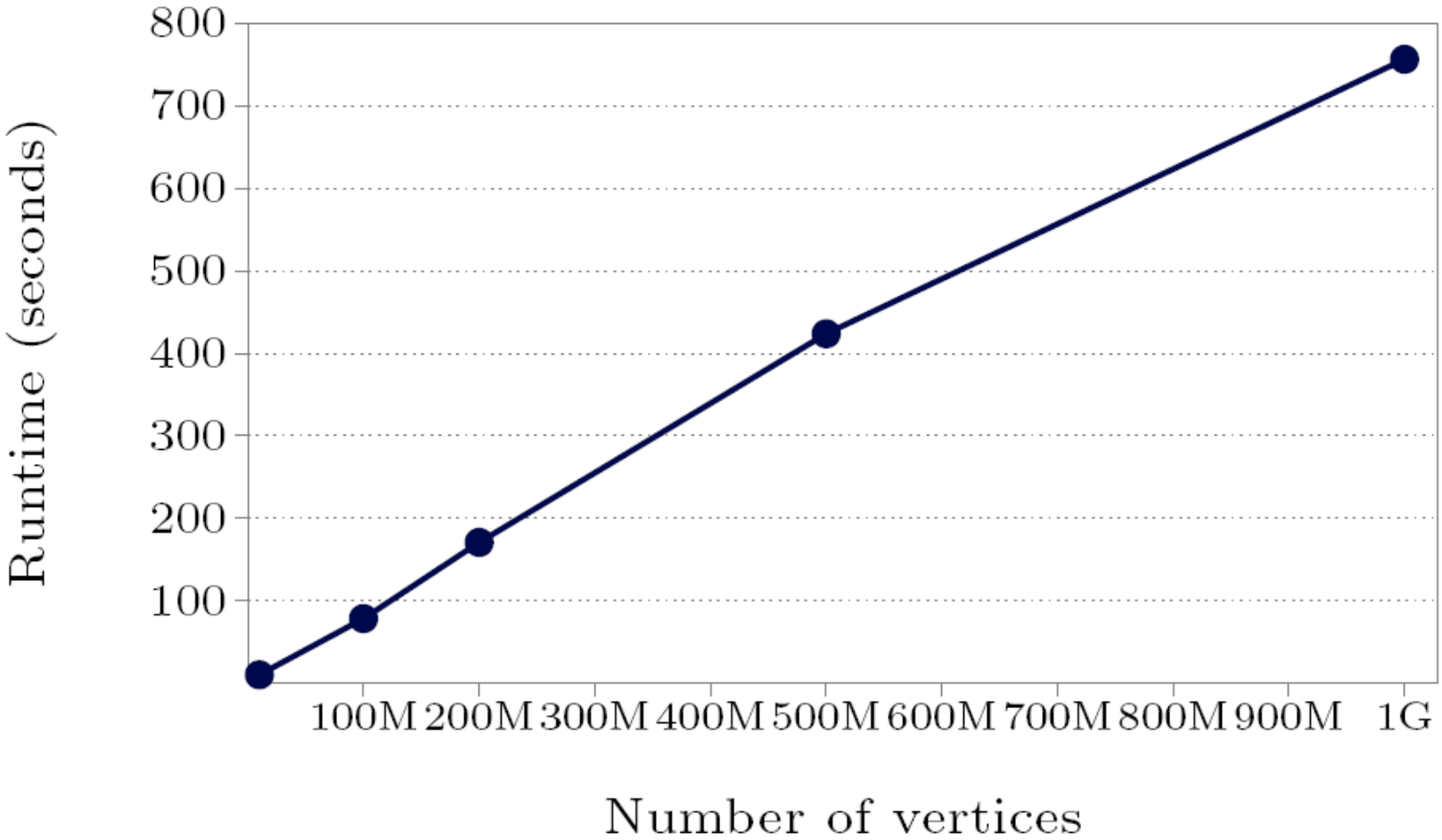
Experiments

- SSSP – binary trees: varying graph sizes on 800 worker tasks



Experiments

- SSSP – Random graphs: varying graph sizes on 800 worker tasks



Differences from MapReduce

- Graph algorithms can be written as a series of chained MapReduce invocation
- Pregel
  - Keeps vertices & edges on the machine that performs computation
  - Uses network transfers only for messages
- MapReduce
  - Passes the entire state of the graph from one stage to the next
  - Needs to coordinate the steps of a chained MapReduce

MapReduce vs Graph Processing

- When to use MapReduce vs Graph Processing Frameworks?
  - Two models can be reduced to each other
  - MR on Pregel: Every vertex to every other vertex
  - Pregel on MR: Similar to PageRank on MR

- Use graph fw for sparse graphs and matrixes
  - Comm. limited to small neighborhood (partitioning)
  - Faster than the shuffle/sort/reduce of MR
  - Local state kept between iterations (side-effects)

MapReduce vs Graph Processing (2)

- Use MR for all-to-all problems
  - Distributed joins
  - When little or no state between tasks (otherwise have to written to disk)
- Graph processing is middleground between message passing (MPI) and data-intensive computing

Backup Slides

Pregel vs GraphLab

- Distributed system models
  - Asynchronous model and Synchronous model
  - Well known tradeoff between two models
    - Synchronous: concurrency-control/failures easy, poor perf.
    - Asynchronous: concurrency-control/failures hard, good perf.
- Pregel is a synchronous system
  - No concurrency control, no worry of consistency
  - Fault-tolerance, check point at each barrier
- GraphLab is asynchronous system
  - Consistency of updates harder (sequential, vertex)
  - Fault-tolerance harder (need a snapshot with consistency)

Pregel vs GraphLab (2)

- Synchronous model requires waiting for every node
  - Good for problems that require it
  - Bad when waiting for stragglers or load-imbalance
- Asynchronous model can make faster progress
  - Some problems work in presence of drift
  - Fast, but harder to reason about computation
  - Can load balance in scheduling to deal with load skew