PhD course
Big Data Analytics

LINKÖPING UNIVERSITY

Programming Models for Big-Graph Analytics

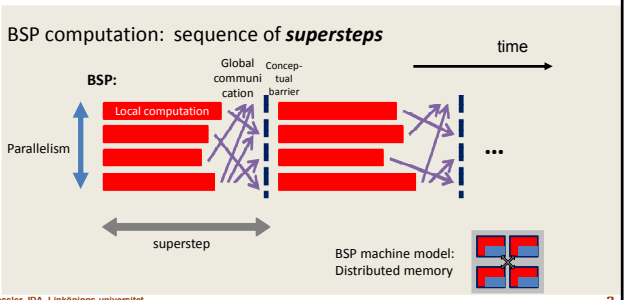
Bulk-Synchronous Parallel Computing, Pregel, Giraph, Hama

Christoph Kessler
IDA, Linköping University

March 2017

Bulk-Synchronous Parallel (BSP) Computing

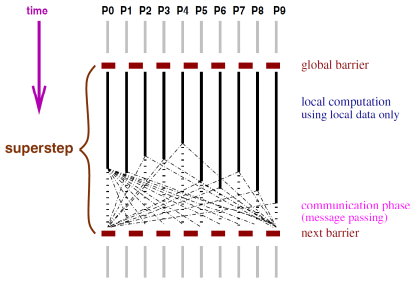
- Bulk-Synchronous Parallel (BSP) Model [Valiant'90]
  - A *Bridging Model* between abstract PRAM model and real-world parallel computers, to support algorithm development



BSP-Model

Bulk-synchronous parallel programming [Valiant'90] [McColl'93]

BSP computer = abstract message passing architecture (p, L, g, s)



MIMD

SPMD

h-relation models communication pattern / volume

h_i [words] = comm. fan-in, fan-out of P_i

$h = \max_{1 \le i \le p} h_i$

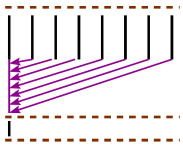
$t_{step} = w + hg + L$

BSP program = sequence of supersteps, separated by (logical) barriers

BSP Example:
Global Maximum (NB: non-optimal algorithm)

Compute maximum of n numbers $A[0,...,n-1]$ on BSP(p, L, g, s):

```
// A[0..n-1] distributed block-wise across p processors
step
  // local computation phase:
  m ← −∞;
  for all A[i] in my local partition of A {
    m ← max (m, A[i]);
  // communication phase:
  if myPID ≠ 0
    send ( m, 0 );
  else    // on P0:
    for each i ∈ {1,...,p−1}
      recv ( m_i, i );
step
  if myPID = 0
    for each i ∈ {1,...,p−1}
      m ← max(m, m_i);
```



Local work:
$\Theta(n/p)$

Communication:
$h = p - 1$
(P_0 is bottleneck)

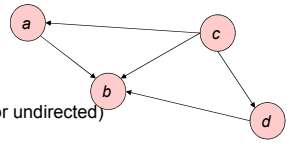
$t_{step} = w + hg + L$
$= \Theta\left(\frac{n}{p} + pg + L\right)$

BSP Remarks

- Local variables of BSP processes persist to next superstep
- In superstep *s*, contents of messages sent (and received) in step *s*-1 are accessible
- Two-sided and/or one-sided message passing possible

- BSP implementations in the 1990s include
  - *BSPlib* – library for C atop MPI [Hill et al. '98]
    - One-sided communication (put, get)
  - *PUB* – library for C atop MPI [Bonorden et al.'03]
    - One-sided communication
  - *NestStep* [K. 2000, 2004]
    - Partitioned global address space (PGAS) language extension of C

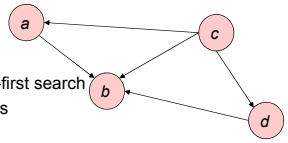
Graphs



- G=(V,E)
  - V: set of nodes
  - E: set of edges (directed or undirected)
    - Storage:
      - Adjacency matrix – random access but too inefficient for sparse graphs
      - Adjacency list – each node holds a list of its neighbors

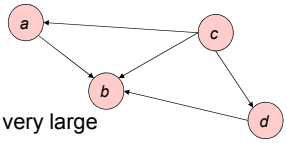
- Graph examples
  - WWW
  - Social network graphs
  - Transportation networks
  - Similarity of documents
  - Citation relationships
  - …

Graphs



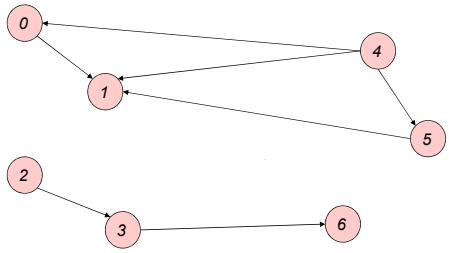
- **Graph algorithms**
  - Depth-first search, Breadth-first search
  - Single-source shortest paths
  - All-pairs shortest paths
  - Clustering
  - Page rank
  - Minimum cut / Maximum flow
  - Connected components
  - Strongly connected components
  - Hamiltonian, Euler Tour, Traveling Salesman Problem
  - …
- Typical properties
  - Poor locality of memory accesses
  - Little work per vertex
  - Changing degree of parallelism

Graphs

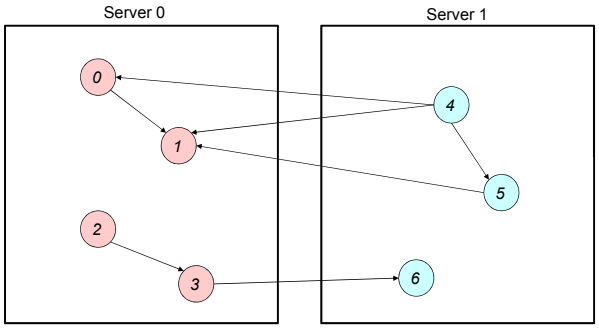


- Today, many graphs are very large
- How to distribute a graph?
  - Usually, partition and distribute the array of vertices (each vertex with a local *value* and its adjacency list of outgoing edges)
  - If locality matters in partitioning?
    - Replace default partitioning with a user-defined partitioning
    - Graph partitioner
      e.g. METIS (used in HPC e.g. for FEM meshes)

Example



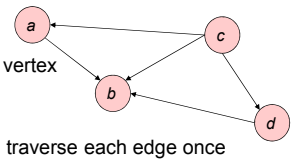
Example



Hash vertices over servers.

Ex.: S servers, N vertices, *hashfn* = *id*, B = ceil(N/S):
- *owner*(i) = i div B
- *local index*(i) = i mod B

Graph Traversal

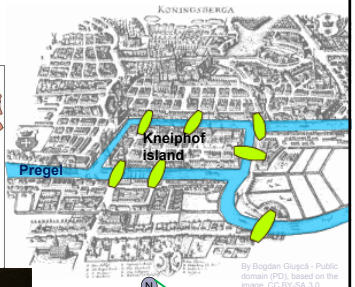


- Exploration starts at some vertex
- For each vertex,
  consider all its neighbors
- Visit each vertex once and traverse each edge once
- Sequential algorithms:
  Depth first search, Breadth first search
  Special case: Tree traversals
- Much (dynamic) parallelism
  (unless graph has very special structure)
- Little/no data locality
  (unless graph has very special structure)
- History of exploration (i.e., recursive call stack, visited vertices) must be kept
  → dependences / communication for visiting a remote vertex when calculating on a distributed system

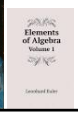
Graph Analytics

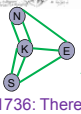
- Global analysis of a graph
- Consider each vertex and each edge
- 2 flavors:
  - Compute one value over all vertices / edges
    - E.g. sum-up/maximize/… attribute values over all vertices
  - Compute one value for each vertex / each edge
    - E.g. page rank for each vertex (web page)
- Often, iterative
  - multiple sweeps over the graph
- Suitable for massively parallel and distributed computation
- Requires a global view (random access) of the graph
- How to address huge graphs?

Pregel



Leonhard Euler (1707-1783), Swiss mathematician





A multigraph

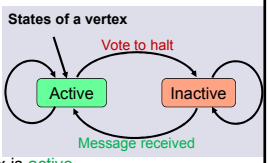
1736: There is no *Eulertour* (path traversing each bridge exactly once) for Königsberg nor any other topology with >2 nodes having odd degree

Pregel

[Malevicz *et al.* 2010]

- A framework to process/query large distributed graphs
- Proprietary (by Google)
- Pregel is the ”MapReduce” for graphs
  - Iterative computations
    - Sequence of BSP supersteps
  - Each BSP superstep is basically a composition of the MapReduce phases (Map, Combine, Sort, Reduce)
- Attempts to utilize all servers available by partitioning and distributing the graph
  - Good for computations touching all vertices / edges
  - Bad for computations touching only few vertices / edges

Pregel
Programming Model



- **Graph Vertices**
  - Each with unique **identifier** (String)
  - Each with a user-defined **value**
  - Each with a **state** in { Active, Inactive }
  - Initially (before superstep 1), every vertex is active
- **Graph Edges**
  - Each edge can have a user-defined **value** (e.g., weight)
- **One (conceptual) BSP process assigned to each vertex**
  - Not to the edges, by the way…
  - Iteratively executing supersteps while active
  - Two-sided communication with **send**() and **receive**() calls
- Graph can be **dynamic**
  - Vertices and/or edges can be added or removed in each superstep
- **Algorithm termination**
  - When all vertices are simultaneously inactive and there are no messages in transit
  - Otherwise, go for another superstep

Example: Maximum vertex value

For now, assume 1 BSP processing node per vertex

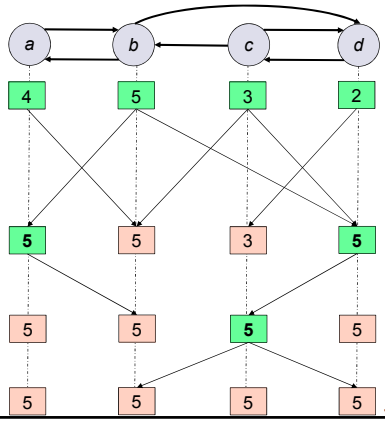
Initial states / values

Superstep 0:
send my value to all neighbors;
receive from all neighbors

Superstep 1:
maximize over all received values;
if larger than my value,
  update it and
  send new value to all neighbors;
  receive from neighbors;
else vote for halt;

Superstep 2:
…

Superstep 3:
…



Example: Maximum vertex value

Now: multiple BSP processes per server node
→ message aggregation,
  local combining

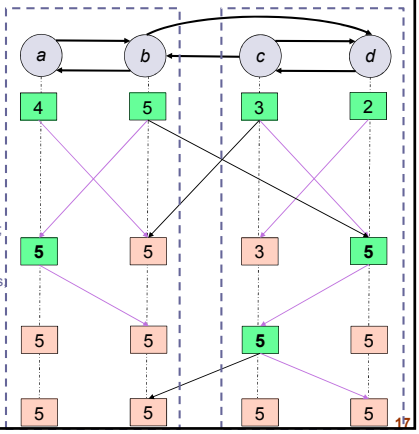
Initial states / values

Superstep 0:
send my value to all neighbors;
receive from all neighbors

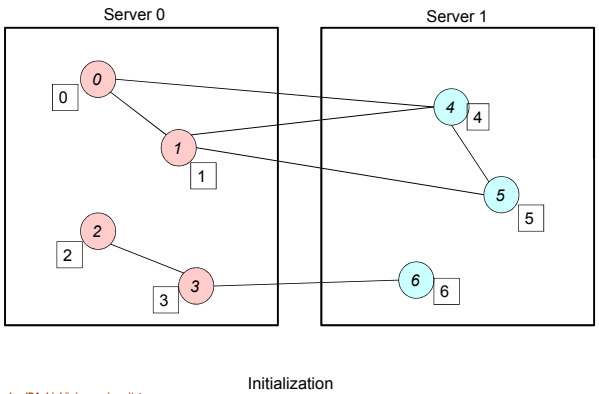
Superstep 1:
maximize over all received values;
if larger than my value,
  update it and
  send new value to all neighbors;
  receive from neighbors;
else vote for halt;

Superstep 2:
…

Superstep 3:
…

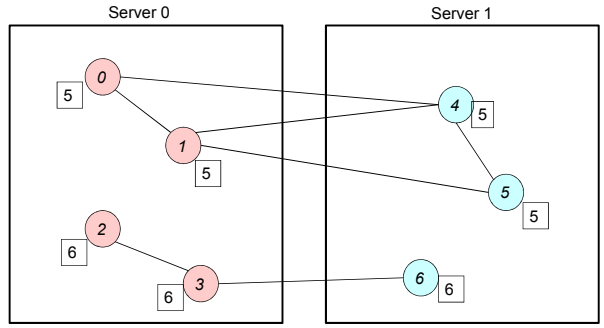


Exercise: Connected Components
based on Maximum (with local accumulation)



Initialization

Exercise: Connected Components
based on Maximum (with local accumulation)



Final values → CC ID's

Pregel C++ API (1)

- **Vertex**<VertexValue,EdgeValue,MessageValue> class
  - User overrides virtual *Compute*() method for superstep behavior
  - *vertex_id*() returns vertex identifier (string)
  - VertexValue: user-specified datatype for value
  - *Get_value*() reads, *Mutable_value*() sets the value
  - *GetOutEdgeIterator*() get an OutEdgeIterator
- int64 *superstep*() queries the superstep number.
  - Vertex and edge values are the only values that persist to the next superstep
- All **messages** sent to vertex in previous superstep are available
  - Each message contains a value of type MessageValue
  - void *SendMessageTo*(dest_vertex_id, msg_value)
  - void *VoteToHalt*()

Example: PageRank

```
class PageRankVertex : public Vertex<double, void, double>
{
  public:
    virtual void Compute ( MessageIterator* msgs )
    {
        if (superstep() >= 1) {
            double sum = 0;
            for ( ; ! msgs->Done(); msgs->Next() )
                sum += msgs->Value();
            *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors( GetValue() / n );
        }
        else
            VoteToHalt();
    }
};
```


Pregel C++ API (2)

- **Combiners**
  - For reductions (e.g. maximization, sum, …) of values going to same receiver
  - Local accumulation before sending off accumulated value to destination residing on remote server
  - Fewer inter-node messages, fast node-local combining
- **Aggregators**
  - are global All-Reductions (every vertex gets the global sum/maximum/… over all vertices in the graph)
  - Could be generalized to histogramming
    - For MPI programmers known from MPI_Allreduce
  - Needs only 1 superstep instead of possibly many

Pregel C++ API (3)

Graph topology modifications

- Superstep execution can add or remove vertices or edges
- Could lead to conflicts across parallel BSP processes, e.g.
  - Multiple processes try to create a vertex with same name in same superstep
  - .. Or: with same name but different initial values
  - One wants to add an edge from/to a vertex that another wants to remove
  - …
- Conflict resolution policy:
  - Edge removals always done first, then vertex removals
  - Then do vertex additions, then edge additions
  - Then do *Compute*() for the vertex
- Purely vertex-local modifications (e.g. adding/removing own outgoing edges) are conflict-free

Pregel Implementation

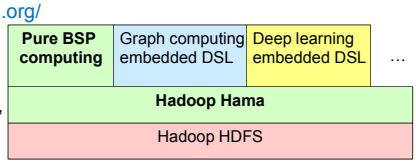
- Proprietary (Google)
- Atop Google cluster architecture [Barroso et al. 2003]
- Persistent data stored as files on a distributed file system e.g. in Google File System GFS
  - with a name service for file location lookup
- Graph partitioning: vertices hashed over the *p* cluster nodes: *owner*(v) = *hashfunction*(v) mod *p*
  - Default distribution is not transparent to the programmer
- Master process (e.g. BSP process 0)
  - coordinates workers and decides about termination
- Worker processes
  - Number can be controlled by the user
  - Call *Compute*() for each vertex in local partition
  - (Combine and) Aggregate messages to vertices on each other node
  - Tell the master how many local vertices remain active for next superstep
- Fault tolerance through checkpointing, master reassigns lost partitions

Giraph

- Pregel-like, Java-based API atop Hadoop
- Open-source
  - Giraph.apache.org

Hama

- BSP computing over Hadoop
- https://hama.apache.org/
- Includes embedded DSLs for graph computing, deep learning…



- Java-based API

```
public abstract class Vertex<V extends Writable,
                             E extends Writable,
                             M extends Writable>
    implements VertexInterface<V, E, M>
{
    public void compute ( Iterator<M> messages ) throws IOException;
    ...
};
```


References

BSP:

- L. Valiant: A bridging model for parallel computation. *Comm. ACM* 33(8), 103-111, 1990
- L. Valiant: A bridging model for multi-core computing. *J. Comp. and Syst. Sciences* 77(1), 154-166, 2011.
- D.B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl: Questions and Answers about BSP. *Scientific Programming*, vol. 6, no. 3, pp. 249-274, 1997. doi:10.1155/1997/532130

Pregel:

- Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski: Pregel: A System for Large-Scale Graph Processing. Proc. SIGMOD'10, June 6–11, 2010, pp. 135-145, ACM.

BSP on Hadoop / Cloud:

- Apache HAMA: https://hama.apache.org/
- K. Siddique et al.: Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications. IEEE Access 4:8879 - 8887, Nov. 2016. http://ieeexplore.ieee.org/document/7752866/
- Redekopp, M., Simmhan, Y., Prasanna, V.K.: Optimizations and analysis of BSP graph processing models on public clouds. IPDPS 2013

Questions for Reflection

- Consider the simple (not time-optimal) BSP maximization example. Design an asymptotically time-optimal BSP solution. How many supersteps are required? (assuming that no Aggregator construct is available)
- Why does (parallel) Depth-first-search not fit so well for distributed graph computing with Pregel?
- Run the Connected Components algorithm example (with 7 nodes and 2 server nodes) as shown in the image. Show where local accumulation by a combiner is applied.
- For the Pregel vertex value maximization algorithm, how many supersteps would you need in the best / worst / average case (assuming no Aggregator construct were available)? What *is* the best / worst case here?
- How could Aggregators be implemented in a Pregel cluster?
- How could removals of edges speed up the vertex value maximization algorithm for large distributed graphs?
- Find further graph algorithms that are similar in structure to distributed connected components or PageRank.