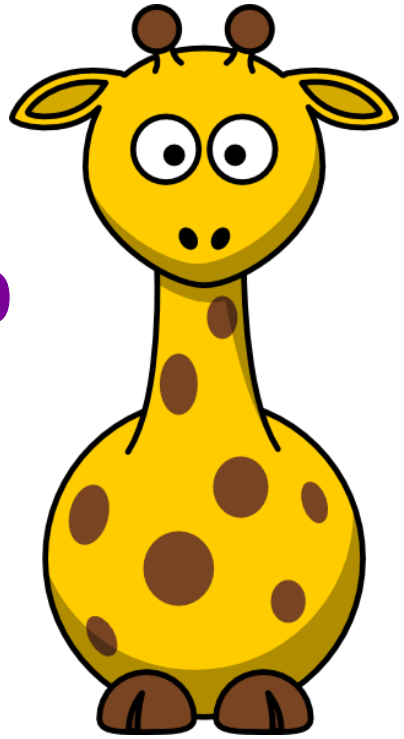
Giraph: Large-scale graph processing infrastructure on Hadoop

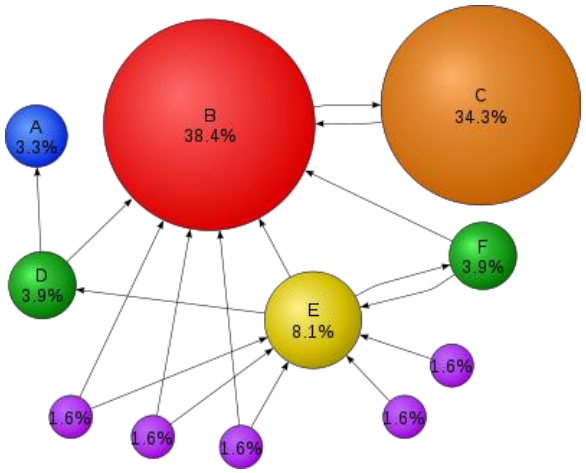
Qu Zhi



YAHOO!®

Why scalable graph processing?

- Web and social graphs are at immense scale and continuing to grow
  - In 2008, Google estimated the number of web pages at 1 trillion
  - At the 2011 F8, Facebook announced it has 800 million monthly active users
  - In September 2011, Twitter claimed to have over 100 million active monthly users
  - In March 2011, LinkedIn said it had over 120 million registered users
- Relevant and personalized information for users relies strongly on iterative graph ranking algorithms (search results, news, ads, etc.)
  - In web graphs, page rank and its variants



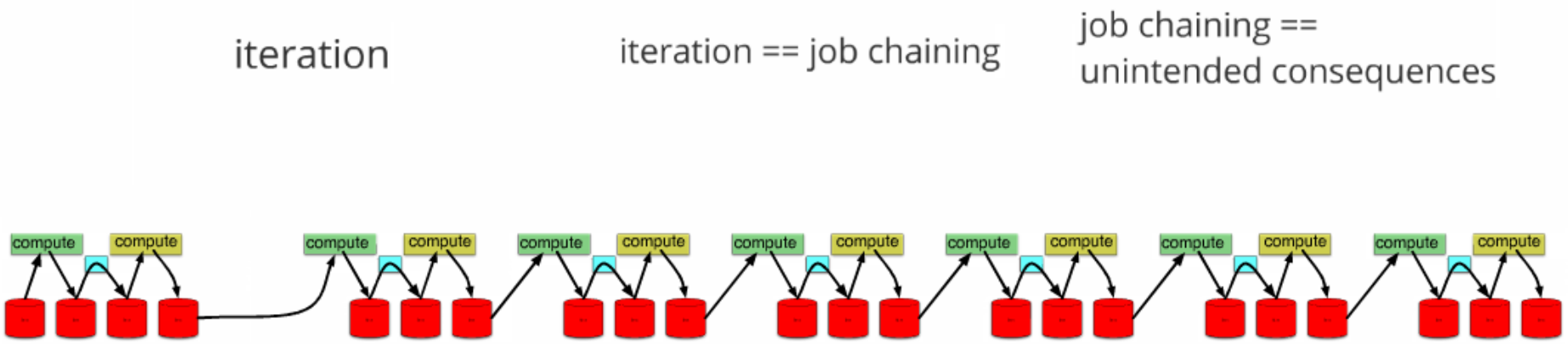
YAHOO!

Example social graph applications

- Popularity rank (page rank)
  - Can be personalized for a user or “type” of user
  - Determining popular users, news, jobs, etc.
- Shortest paths
  - Many variants single-source, s-t shortest paths, all-to-all shortest (memory/storage prohibitive)
  - How are users, groups connected?
- Clustering, semi-clustering
  - Max clique, triangle closure, label propagation algorithms
  - Finding related people, groups, interests

Existing solutions

- Sequence of map-reduce jobs in Hadoop
  - Classic map-reduce overheads (job startup/shutdown, reloading data from HDFS, shuffling)
  - Map-reduce programming model not a good fit for graph algorithms
  - Disk IO and job scheduling quickly dominate the algorithm



YAHOO!

Existing solutions

- Message passing interface (MPI)
  - Not fault-tolerant
  - Too generic
- Google's Pregel
  - Requires its own computing infrastructure
  - Not available (unless you work at Google)
  - Master is a SPOF

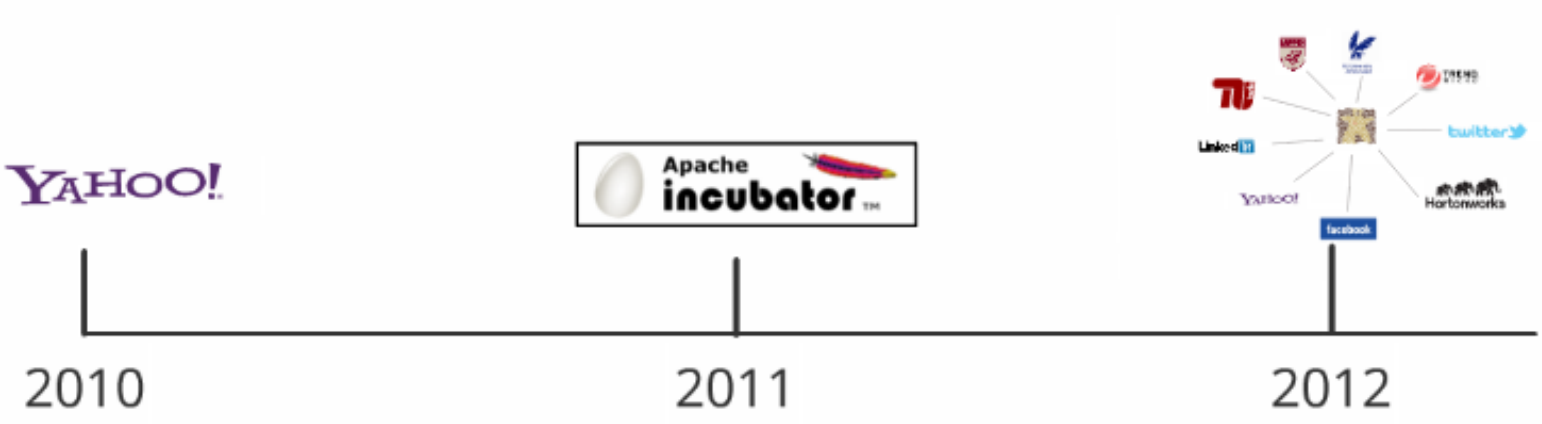
YAHOO!

Giraph goals

- Easy deployment on existing big data processing infrastructure
  - Maintaining a separate graph processing cluster is a lot of overhead for operations
- Dynamic resource management
  - Handle failures gracefully
  - Integrate new resources when available
- Graph-oriented API
  - Make graph processing code as simple as possible
- Open
  - Leverage the community

From Yahoo! to

- Yahoo! Research developed original codebase
- Entered Apache Incubator in July 2011
- New Apache team quickly formed



YAHOO!

Giraph design

Easy deployment on existing big data processing infrastructure

- Leverage Hadoop installations around the world for iterative graph processing
  - Big data today is processed on Hadoop with the Map-Reduce computing model
  - Map-Reduce with Hadoop is widely deployed outside of Yahoo! as well (i.e. EC2, Cloudera, etc.)

Dynamic resource management

- Bulk synchronous parallel (BSP) computing model
- Fault-tolerant/dynamic graph processing infrastructure
  - Automatically adjust to available resources on the Hadoop grid
  - No single point of failure except Hadoop namenode and jobtracker
  - Relies on ZooKeeper as a fault-tolerant coordination service

Bulk synchronous parallel model

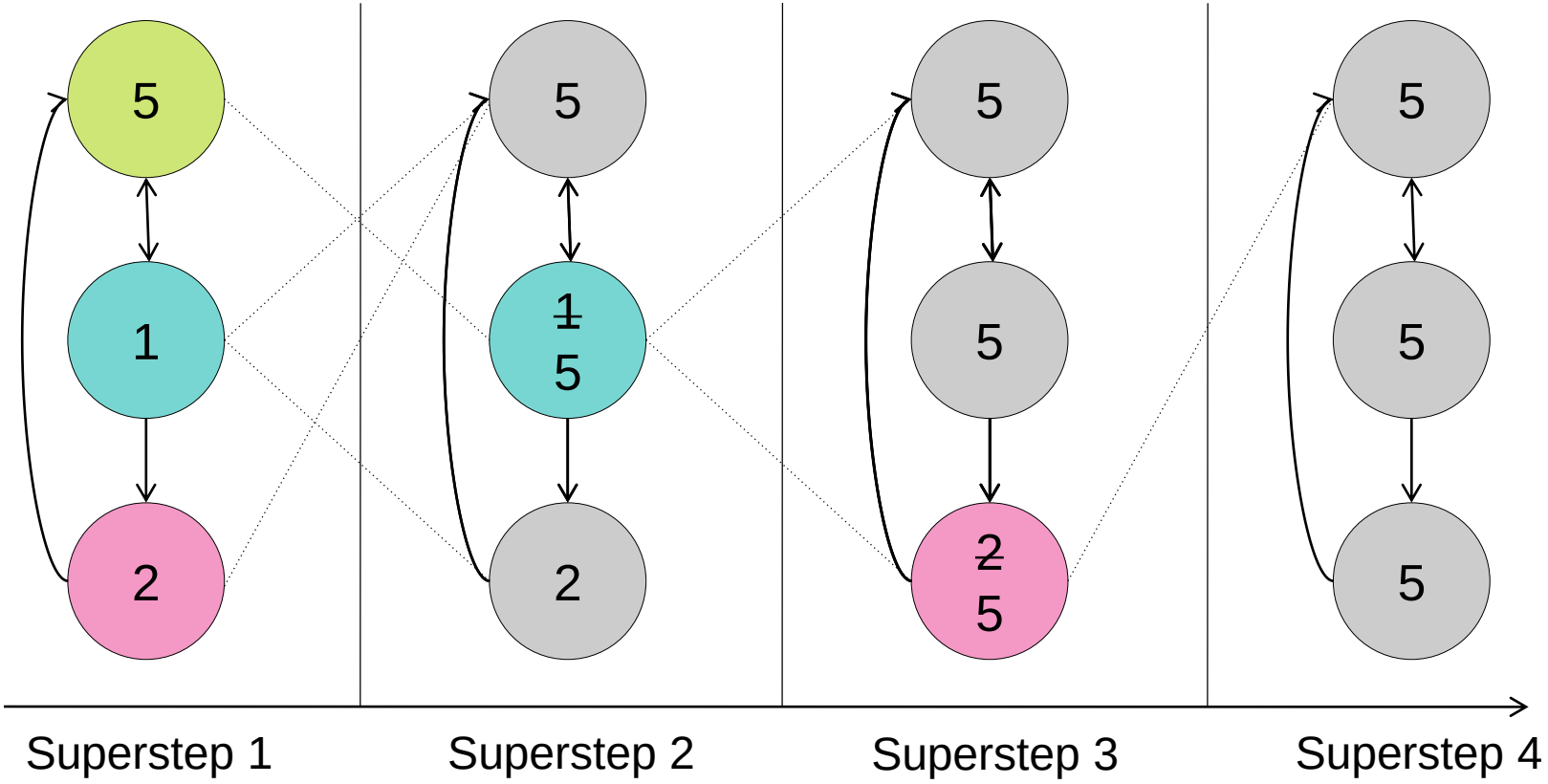
- Sequential computation on a single physical machine restricts the computational problem domain
- BSP is a proposal of a “bridging” model for parallel computation
  - High-level languages bridged to parallel machines
- 3 main attributes
  - **Components** that process and/or provide storage
  - **Router** to deliver point-to-point messages
  - **Synchronization** of all or a subset of components through regular intervals (supersteps)
- Computation is done when all components are done
- Only a model, does not describe an implementation

Why use BSP?

- A relatively simple computational model
- Parallelization of computation/messaging during a superstep
  - Components can only communicate by messages delivered out-of-order in the next superstep
- Fault-tolerant/dynamic resource utilization
  - Supersteps are atomic units of parallel computation
  - Any superstep can be restarted from a checkpoint (need not be user defined)
  - A new superstep provides an opportunity for rebalancing of components among available resources

Maximum vertex value example

- All vertices find the maximum value in a strongly connected graph
- If 1st superstep or the set a new maximum value from incoming messages, send new value maximum to edges, otherwise vote to halt (gray vertices)



YAHOO!

Writing a Giraph application

- Every active vertex will call compute() method once during a superstep
  - Analogous to map() method in Hadoop for a <key, value> tuple
- Users chooses 4 types for their implementation of Vertex (*I* → VertexId, *V* → VertexValue, *E* → EdgeValue, *M* → MsgValue)

Map Reduce	Giraph

public class Mapper<
KEYIN,
VALUEIN,
KEYOUT,
VALUEOUT> {
void map(KEYIN key,
VALUEIN value,
Context context)
throws IOException,
InterruptedException;
}
	public class Vertex<
	I extends WritableComparable,
	V extends Writable,
	E extends Writable,
	M extends Writable> {
	void compute(
	Iterator<M> msgIterator);
	}

Basic Giraph API

Get/set local vertex values	Voting

I getVertexId();
V getVertexValue();
void setVertexValue(V vertexValue);
SortedMap<I, Edge<I, E>> getOutEdgeMap();
long getSuperstep();
	void voteToHalt();
	boolean isHalted**();**

Graph-wide mutation	**Messaging**

void addVertexRequest(MutableVertex<I, V, E, M> vertex);
void removeVertexRequest(I vertexId);
void addEdgeRequest(I sourceVertexId, Edge<I, E> edge);
void removeEdgeRequest(I sourceVertexId, I destVertexId);
	void sendMsg(I id, M msg);
	void sentMsgToAllEdges(M msg);

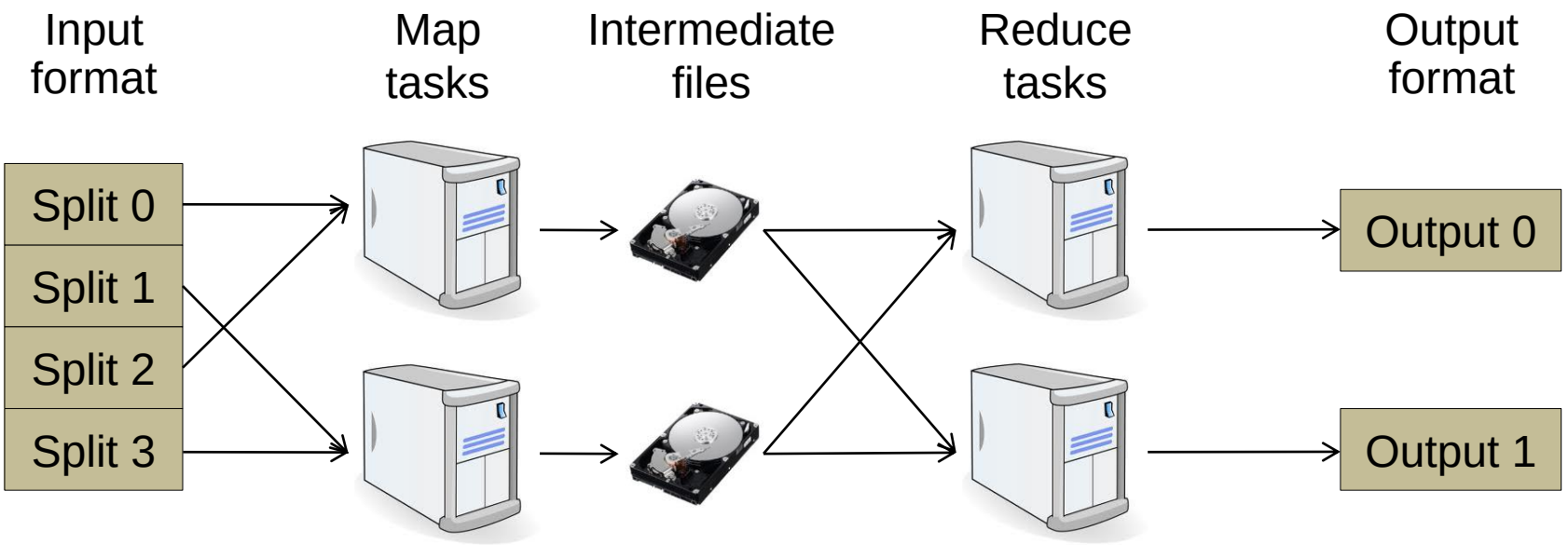
- Local vertex mutations happen immediately
- Vertices “vote” to end the computation
  - Once all vertices have voted to end the computation, the application is finished
- Graph mutations are processed just prior to the next superstep
- Sent messages are available at the next superstep during compute

PageRank example

```
public class SimplePageRankVertex extends Vertex<LongWritable, DoubleWritable,
    FloatWritable, DoubleWritable> {
    public void compute(Iterator<DoubleWritable> msgIterator) {
        if (getSuperstep() >= 1) {
            double sum = 0;
            while (msgIterator.hasNext()) {
                sum += msgIterator.next().get();
            }
            setVertexValue(new DoubleWritable((0.15f / getNumVertices()) + 0.85f * sum);
        }
        if (getSuperstep() < 30) {
            long edges = getOutEdgeIterator().size();
            sentMsgToAllEdges(new DoubleWritable(getVertexValue().get() / edges));
        } else {
            voteToHalt();
        }
    }
```

Hadoop

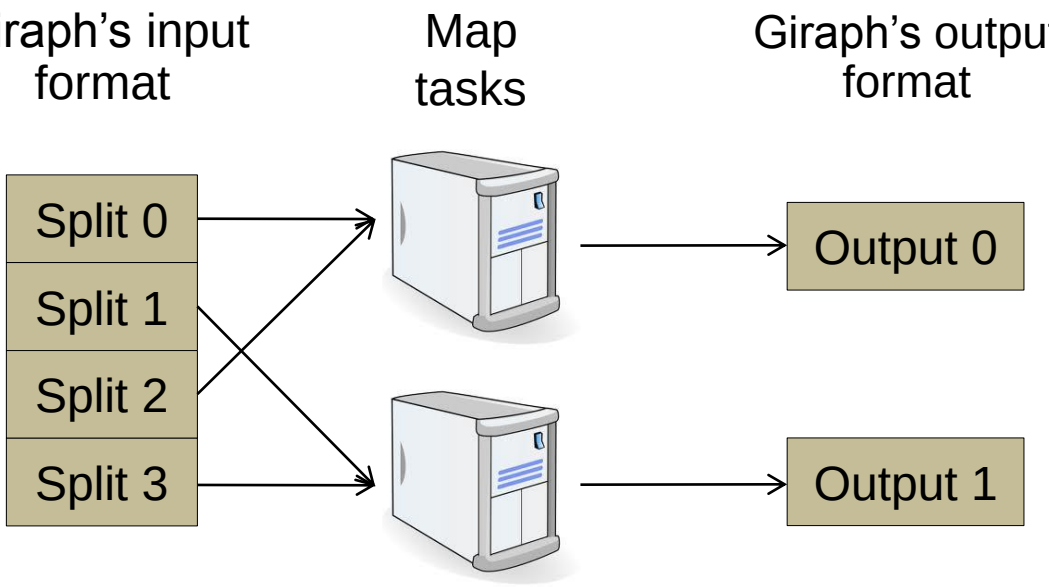
- Open-source implementation of Map-Reduce and GFS (HDFS)
- Meant to solve “big data” challenges such as distributed grep, process log data, sorting, etc.
- Includes resource management (JobTracker)
- Not good for message passing (every message passing step requires a Map-Reduce job)



YAHOO!

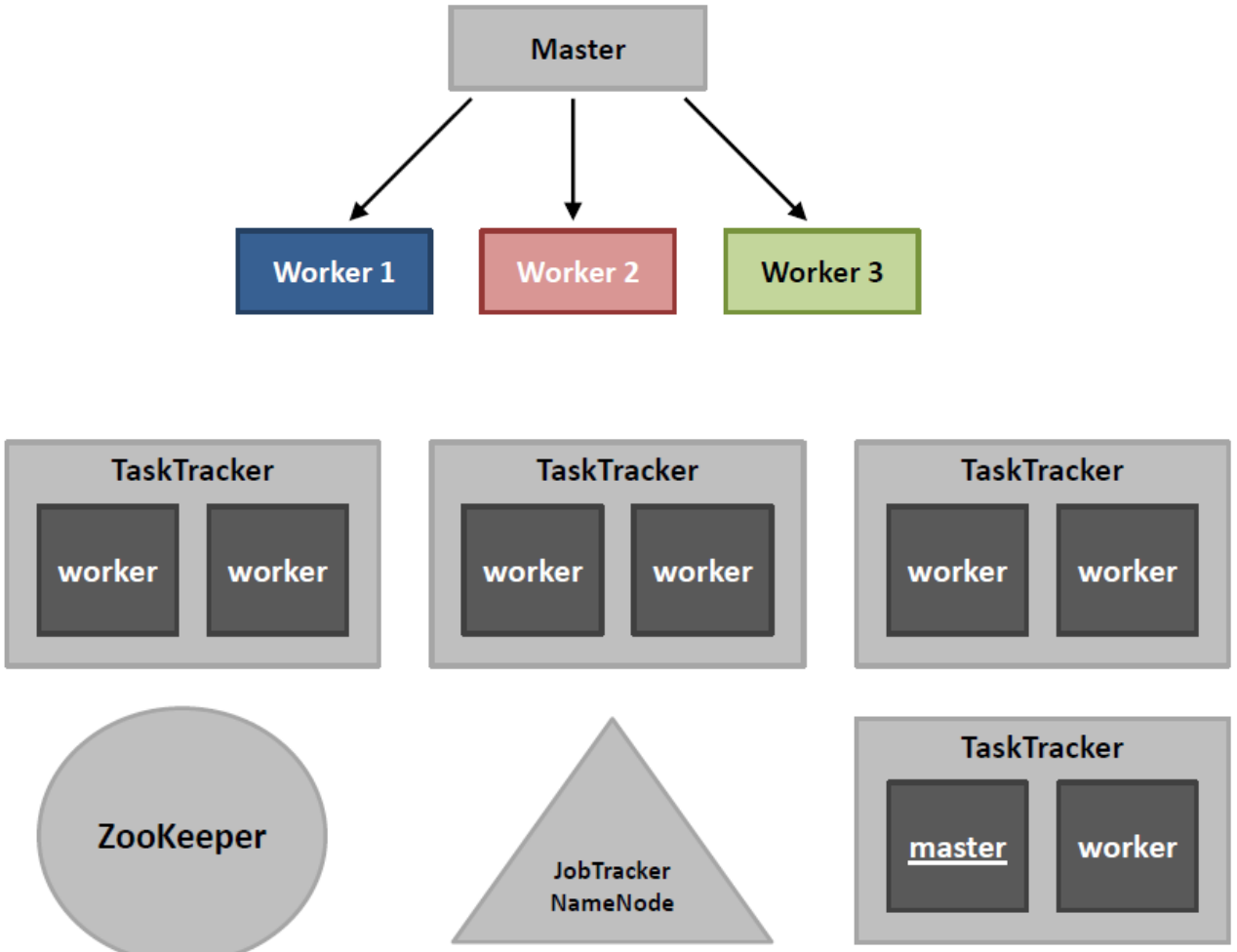
Giraph job from Hadoop's perspective

- Giraph has its own InputFormat that calls the user's **VertexInputFormat**
  - Hadoop will start up the workers requested by Giraph, not based on the InputSplit objects generated by the VertexInputFormat
- Giraph has its own OutputFormat that will call the user's **VertexOutputFormat**
  - Giraph's internal structure will write the data with the user's **VertexWriter**



YAHOO!

Giraph‘s Hadoop usage



YAHOO!

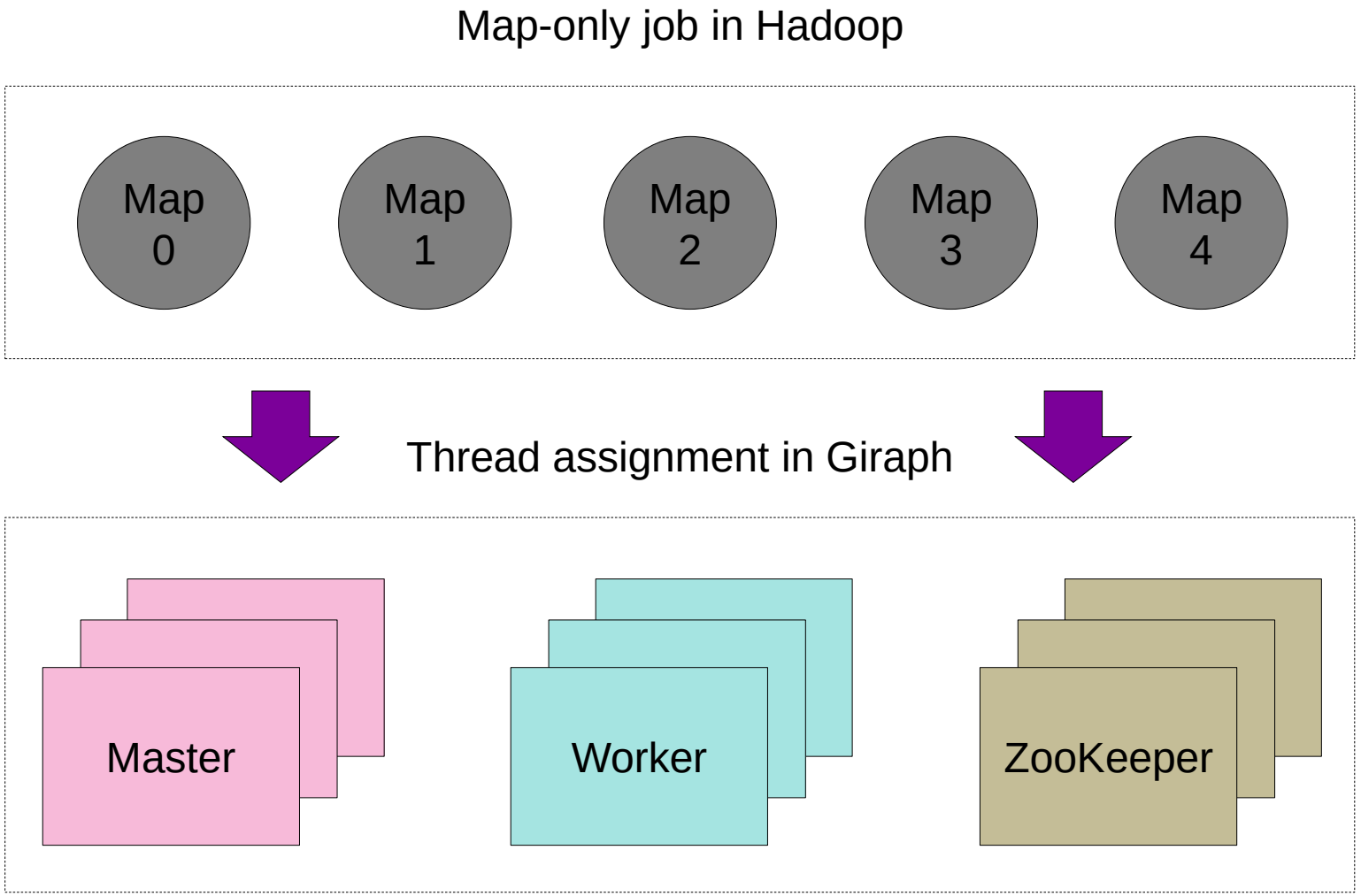
Watching a Giraph job in Hadoop

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.99%	101	0	101	0	0	0 / 0
reduce	0.00%	0	0	0	0	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	4,200
	Total time spent by all maps waiting after reserving slots (ms)	0	0	122,679
	Launched map tasks	0	0	101
Giraph Timers	Setup (milliseconds)	3,184	0	3,184
	Superstep 0 (milliseconds)	8,329	0	8,329
	Superstep 2 (milliseconds)	22,393	0	22,393
	Superstep 1 (milliseconds)	31,609	0	31,609
File Output Format Counters	Bytes Written	0	0	0
Giraph Stats	Aggregate edges	50,000,000	0	50,000,000
	Superstep	3	0	3
	Current workers	100	0	100
	Aggregate finished vertices	0	0	0
	Aggregate vertices	50,000,000	0	50,000,000
File Input Format Counters	Bytes Read	0	0	0
FileSystemCounters	FILE_BYTES_READ	24,900	0	24,900
	HDFS_BYTES_READ	4,444	0	4,444
	FILE_BYTES_WRITTEN	3,060,291	0	3,060,291
	Map input records	101	0	101

YAHOO!

Thread architecture

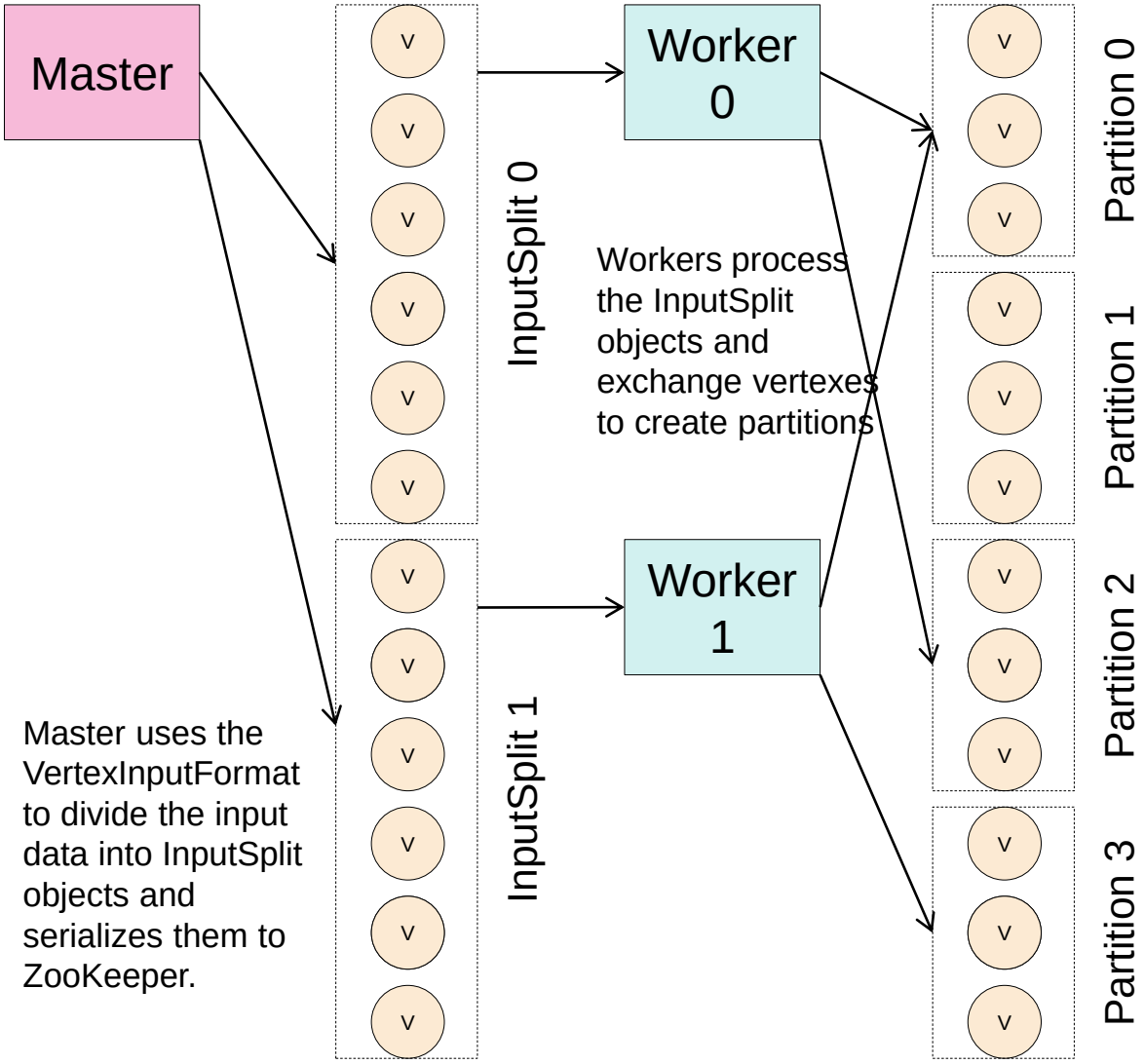


YAHOO!

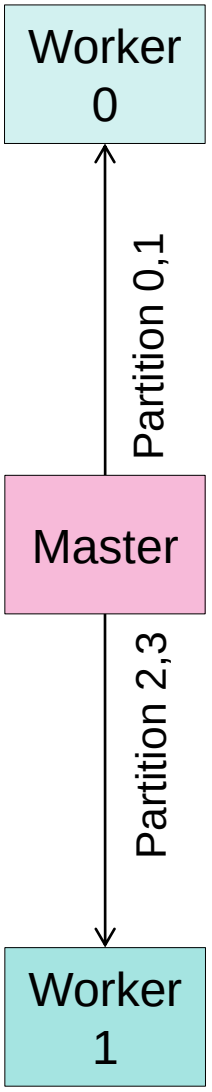
Thread responsibilities

- Master
  - Only one active master at a time
  - Runs the VertexInputFormat getSplits() to get the appropriate number of InputSplit objects for the application and writes it to ZooKeeper
  - Coordinates application
    - Synchronizes supersteps, end of application
    - Handles changes within supersteps (i.e. vertex movement, change in number of workers, etc.)
- Worker
  - Reads vertices from InputSplit objects
  - Executes the compute() method for every Vertex it is assigned once per superstep
  - Buffers the incoming messages to every Vertex it is assigned for the next superstep
- ZooKeeper
  - Manages a server that is a part of the ZooKeeper quorum (maintains global application state)

Vertex distribution

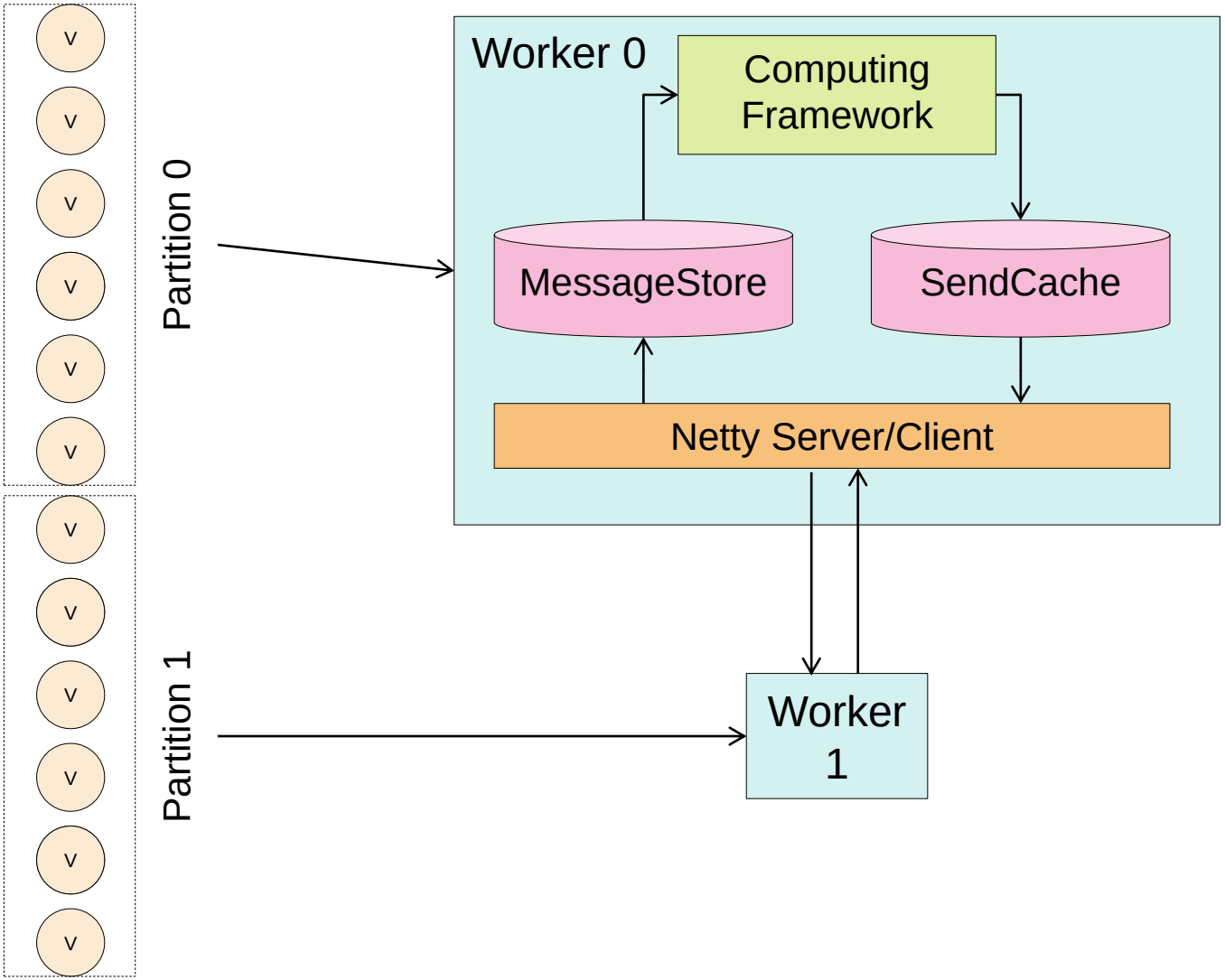


Prior to every superstep, the master assigns every worker 0 or more Partitions according to balancer



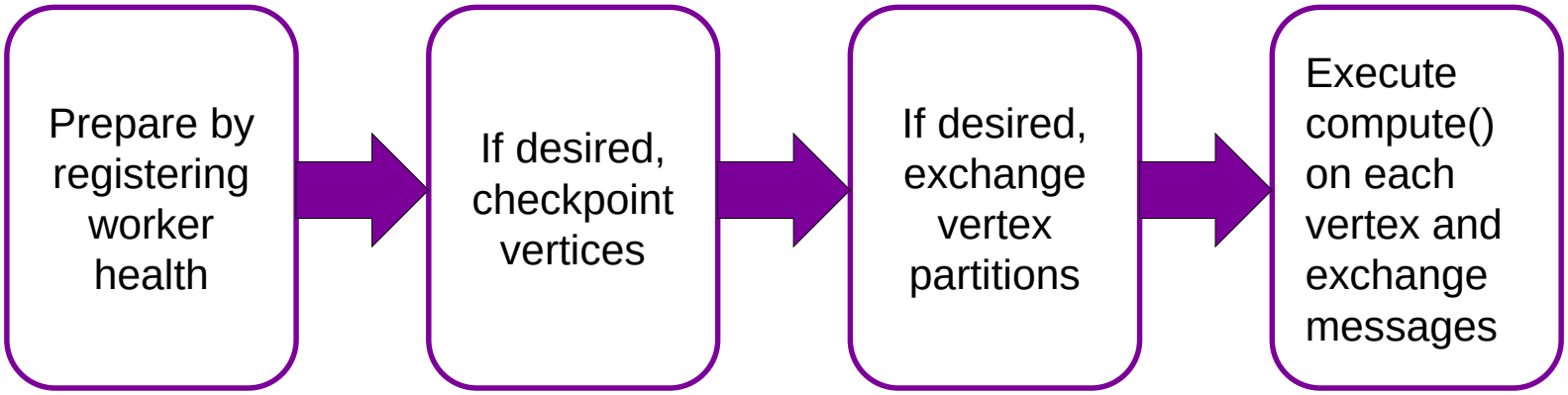
YAHOO!

Work flow



YAHOO!

Worker phases in a superstep

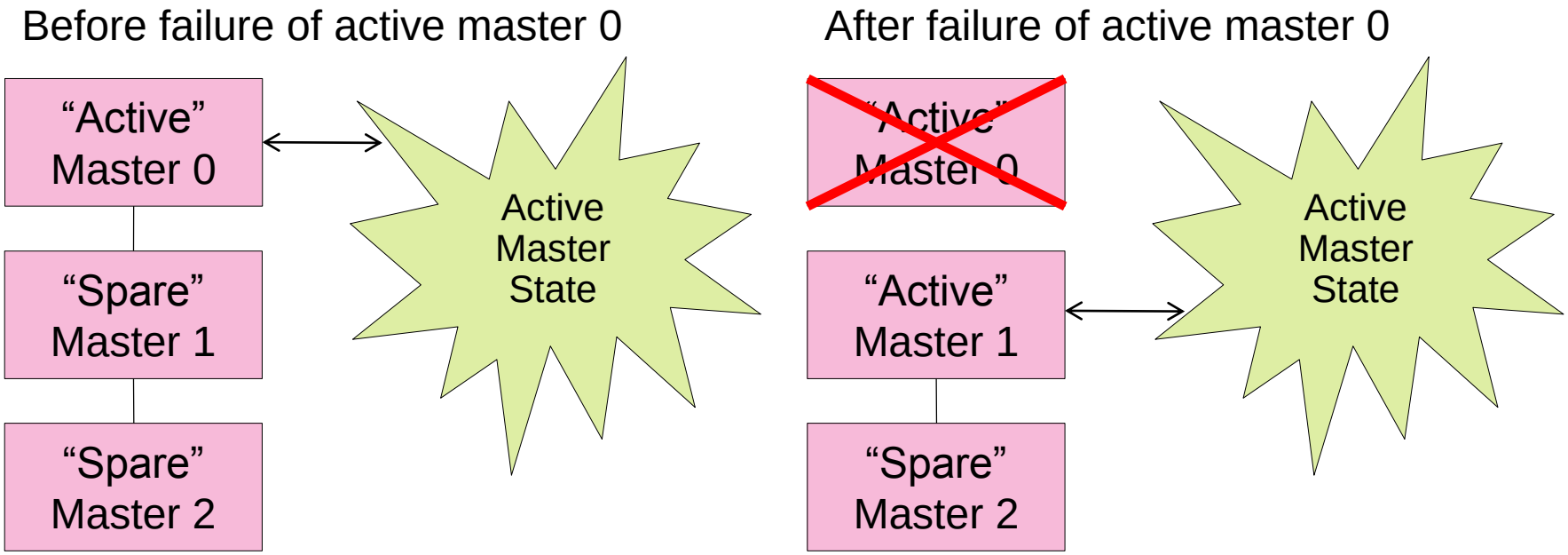


- Master selects one or more of the available workers to use for the superstep
- Users can set the checkpoint frequency
  - Checkpoints are implemented by Giraph (all types implement Writable)
- Users can determine how to distribute vertex partitions on the set of available workers
- BSP model allows for dynamic resource usage
  - Every superstep is an atomic unit of computation
  - Resources can change between supersteps and used accordingly (shrink or grow)

Fault tolerance

- No single point of failure from Giraph threads
  - With multiple master threads, if the current master dies, a new one will automatically take over.
  - If a worker thread dies, the application is rolled back to a previously checkpointed superstep. The next superstep will begin with the new amount of workers
  - If a zookeeper server dies, as long as a quorum remains, the application can proceed
- Hadoop single points of failure still exist
  - Namenode, jobtracker
  - Restarting manually from a checkpoint is always possible

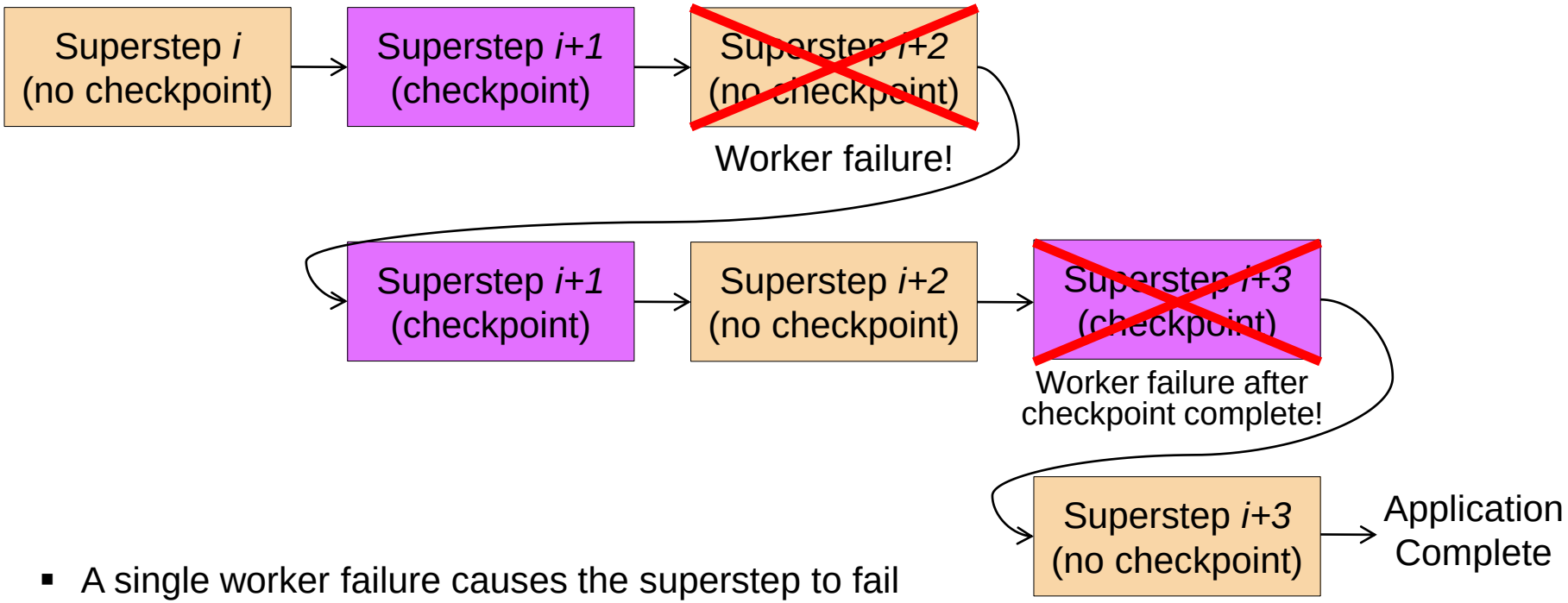
Master thread fault tolerance



- One active master, with spare masters taking over in the event of an active master failure
- All active master state is stored in ZooKeeper so that a spare master can immediately step in when an active master fails
- "Active" master implemented as a queue in ZooKeeper

YAHOO!

Worker thread fault tolerance



- A single worker failure causes the superstep to fail
- In order to disrupt the application, a worker must be registered and chosen for the current superstep
- Application reverts to the last committed superstep automatically
  - Master detects worker failure during any superstep with a ZooKeeper “health” znode
  - Master chooses the last committed superstep and sends a command through ZooKeeper for all workers to restart from that superstep

YAHOO!

Optional features

- Combiners
  - Similar to Map-Reduce combiners
  - Users implement a combine() method that can reduce the amount of messages ~~sent~~ and received
  - Run on both the client side and server side
    - Client side saves memory and message traffic
    - ~~Server side save memory~~ [no combine at all]
- Aggregators
  - Similar to MPI aggregation routines (i.e. max, min, sum, etc.)
  - Users can write their own aggregators
  - Commutative and associate operations that are performed globally
  - Examples include global communication, monitoring, and statistics

What do you have to implement

- Define <I, V, E, M> type
  - Vertex Id, Vertex Data, Edge Data, Message Data
- Subclass **Vertex**
  - Override compute() to implement customize algorithms
- Subclass **VertexInputFormat** to read your graph
  - TextVertexInputFormat
  - From a text file with adjacency lists like *<vertex> <neighbor1> <neighbor2> ...*
- Subclass **VertexOutputFormat** to write back the result
  - IdWithValueTextOutputFormat

YAHOO!

What do you have to implement (cond.)

- [Optional] Combiner
  - Combined on message arrived (!!)
  - public void combine(I vertexIndex, M originalMessage, M messageToCombine);
- [Optional] Partitioner
  - Hash for default implementation (% no. of workers)
  - Range hash
  - Others ? Memcached?

YAHOO!

What do you have to implement (cond.)

- [Optional] Observer
  - Both Worker & Master
  - Pre/Post Application/SuperStep
- [Optional] VertexResolver
  - Resolve graph mutations

- [Optional] Checkpoint & Restart SuperStep
- [Optional] Out-of-core message/graph
  - DiskBackedMessageStore
  - DiskBackedPartitionStore

YAHOO!

Key Packages

Package	Usage
org.apache.giraph.combiner	Def. & Impl. of Combiner
org.apache.giraph.comm.aggregators	Def. & Impl. of Aggregator
org.apache.giraph.io (.formats)	Def. & Impl. of InputFormat/OutputFormat
org.apache.giraph.partition	Def. & Impl. & Partitioner
org.apache.giraph.graph	Graph computing framework
org.apache.giraph.master	Graph computing framework – Master
org.apache.giraph.worker	Graph computing framework – Worker
org.apache.giraph.comm.messages	MessageStore
org.apache.giraph.comm.netty	Networking & Communication related logic
org.apache.giraph.conf	Job conf parameters

YAHOO!

Key Class

Class	Usage
GraphMapper	Glue to Hadoop framework
GraphTaskManager	Main entry of computing framework
BspService	Zookeeper based impl.
-> BspServiceMaster	Master role
-> BspServiceWorker	Worker role
NettyWorkerClientRequestProcessor	Deal with outgoing messages
WorkerRequestServerHandler	Deal with incoming messages
TextVertexInputFormat/TextVertexOutputFormat	Basic input & output
OneMessagePerVertexStore	Used when Combiner is provided
SendMessageCache	Basic Send Cache

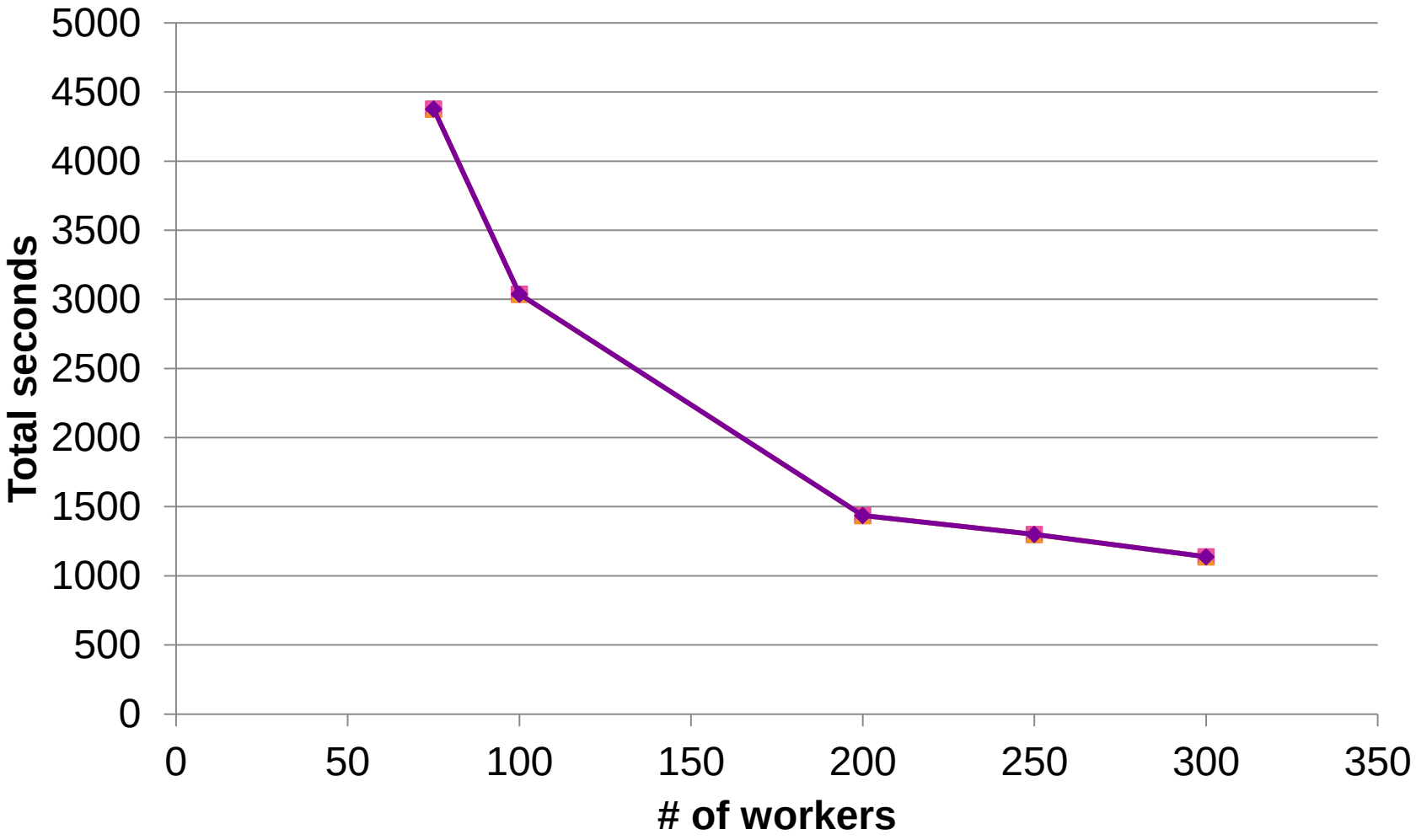
Early Yahoo! customers

- Web of Objects
  - Currently used for the movie database (10's of millions of records, run with 400 workers)
  - Popularity rank, shared connections, personalized page rank
- Web map
  - Next generation page-rank related algorithms will use this framework (250 billion web pages)
  - Current graph processing solution uses MPI (no fault-tolerance, customized code)

Page rank benchmarks

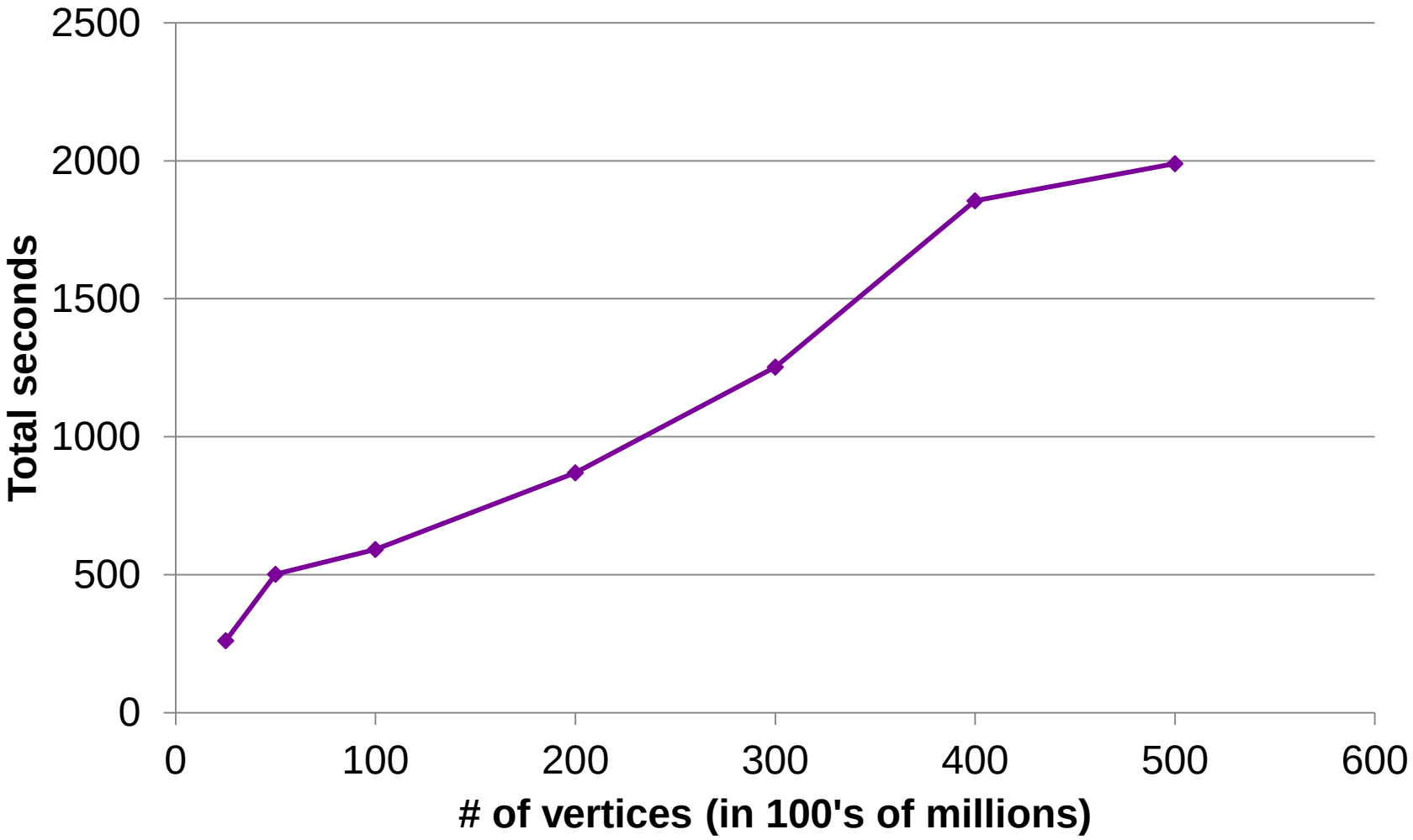
- Tiberium Tan
  - Almost 4000 nodes, shared among numerous groups in Yahoo!
  - Hadoop 0.20.204 (secure Hadoop)
  - 2x Quad Core 2.4GHz, 24 GB RAM, 1x 6TB HD
- org.apache.giraph.benchmark.PageRankBenchmark
  - Generates data, number of edges, number of vertices, # of supersteps
  - 1 master/ZooKeeper
  - 20 supersteps
  - No checkpoints
  - 1 random edge per vertex

Worker scalability (250M vertices)



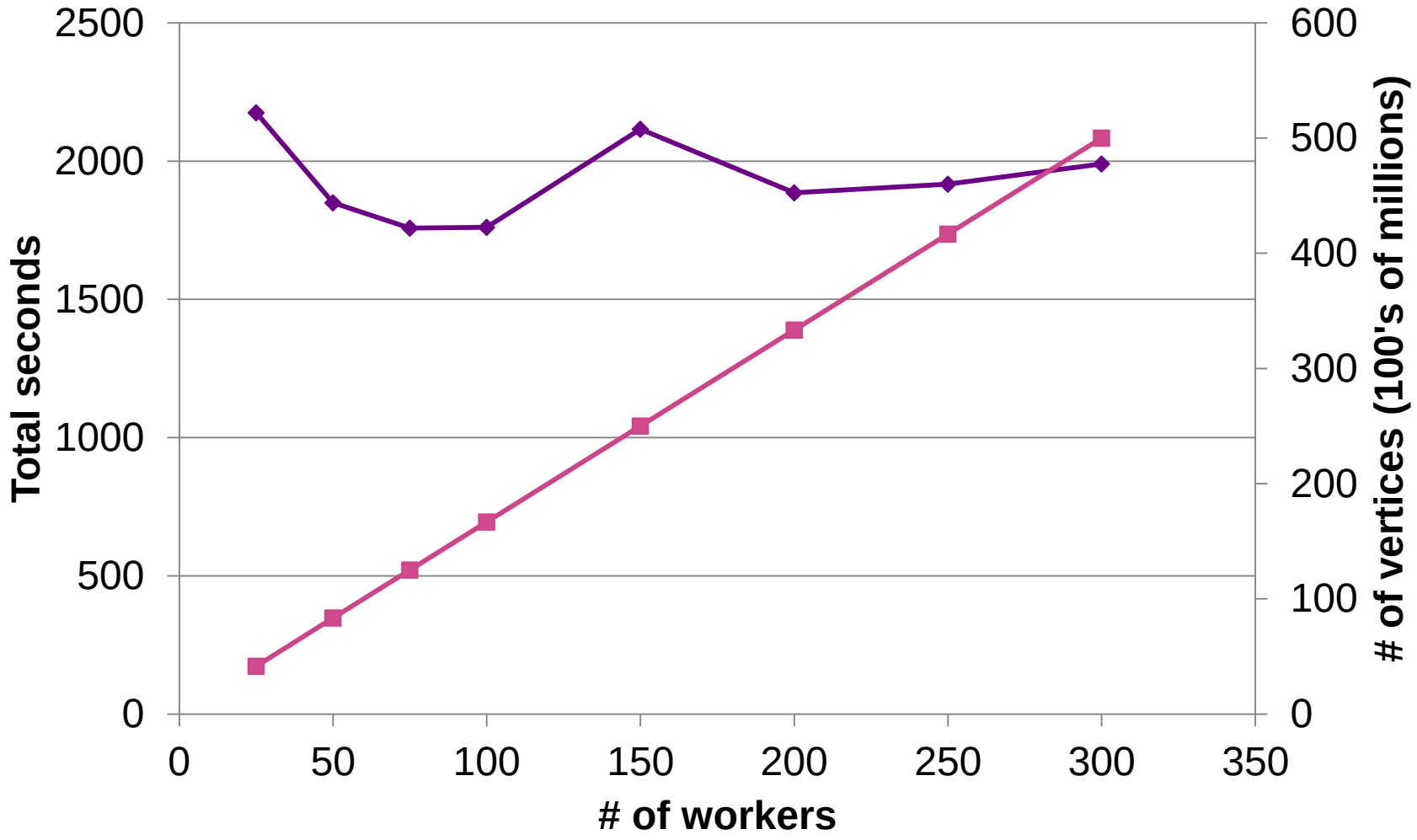
YAHOO!

Vertex scalability (300 workers)



YAHOO!

Vertex/worker scalability



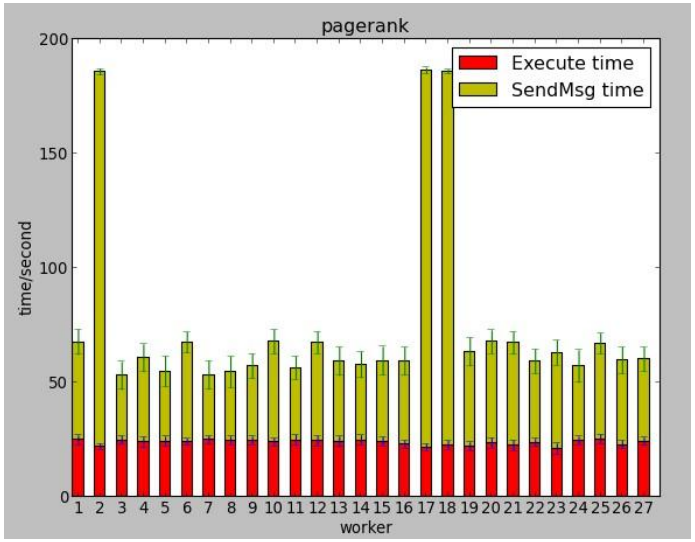
YAHOO!

Conclusion

- Giraph is a graph processing infrastructure that runs on existing Hadoop infrastructure
  - Already being used at Yahoo!
  - Lots of opportunity for new parallel graph algorithms!
- Open source
  - http://incubator.apache.org/giraph/
  - https://issues.apache.org/jira/browse/GIRAPH
- Questions/comments?
  - aching@apache.org

Ongoing work

- Benchmark
  - Pagerank/SSSP of 110M Vertex and 6.7 Billion edges



- Observations
  - 27 machines: 180 sec / ss (superstep)
  - 24 machines: 84 sec / ss (superstep)
  - Great hit of network

- Lagging & Over-distributed
  - Bandwidth sensitive - Benchmark is not productive cluster
  - Message count is of no use
  - No speculative execution, SS Time = Slowest worker's execution Time
    - Dynamic load balance is needed. - How about index?
    - Over distributed problem

YAHOO!

Ongoing work (cont.)

- Index of custom partition
  - Local + Global cache
    - Central / Distributed Cache – Memcache / Redis
    - DHT
  - With-index vertex
    - <Partition tag>|<Vertex id>
- Node movement
  - Index update and out-of-sync
  - Home agent + Message forwarding
- Effective metrics of load balance