Cluster Computing

Big Data Parallelism

- Huge data set
  - crawled documents, web request logs, etc.
- Natural parallelism:
  - can work on different parts of data independently
  - image processing, grep, indexing, many more

Challenges

- Parallelize application
  - Where to place input and output data?
  - Where to place computation?
  - How to avoid network bottleneck?
- How to write the application? Programmer decides or can the system figure it out?
- Balance computations
- Handle failures of nodes during computation
- Scheduling several applications who want to share infrastructure

Map Reduce

- Overview:
  - Partition large data set into M splits
  - Run map on each partition, which produces R local partitions; using a partition function R
  - Run reduce on each intermediate partition, which produces R output files

Details

- Input values: set of key-value pairs
  - Job will read chunks of key-value pairs
- Map(key, value):
  - System will execute this function on each key-value pair
  - Generate a set of intermediate key-value pairs
- Reduce(key, values):
  - Intermediate key-value pairs are sorted
  - Reduce function is executed on these intermediate key-values

Example: Simple Math

Given a set of integers, compute the sum of their square values.

e.g., 1 2 3 4 → 1 + 4 + 9 + 16 → 30

```
Map(key, value) {
    Generate (1, value*value)
}

Reduce(key, values) {
    Int sum = 0;
    For (all values)
        sum += values[i];
}
```

Count words in web-pages

```
Map(key, value) {
    // key is url
    // value is the content of the url
    For each word W in the content
        Generate(W, 1);
}

Reduce(key, values) {
    // key is word (W)
    // values are basically all 1s
    Sum = Sum all 1s in values

    // generate word-count pairs
    Generate (key, sum);
}
```

What are the performance issues with this code?

Reverse web-link graph

Go to google advanced search:
"find pages that link to the page:" cnn.com

```
Map(key, value) {
    // key = url
    // value = content
    For each url, linking to target
        Generate(output target, url);
}

Reduce(key, values) {
    // key = target url
    // values = all urls that point to the target url
    Generate(key, list of values);
}
```

Implementation

- Depends on the underlying hardware: shared memory, message passing, NUMA shared memory, etc.
- Inside Google:
  - commodity workstations
  - commodity networking hardware (1Gbps at node level and much smaller bisection bandwidth)
  - cluster = 100s or 1000s of machines
  - storage is through GFS

Implementation

- Partition input data into M splits
  - starts up many copies of the program on a cluster
  - one master and multiple slaves
  - Map function invoked on key-values
  - Output is buffered in memory and periodically logged to disk (local disk)
- Reduce invocations: partition the intermediate key space into R pieces (e.g., hash(key) % R)
- R and partition function is specified by user

Implementation

- Master keeps track of locations of intermediate keys
- Reducer accesses these values through RPCs
    - reducer sorts all keys assigned to it
    - iterates over each unique key and performs reduce over associated values
    - emits output values that are appended to a final output file for this reduce partition (in GFS)

Issues

- How should M and R compare to no. of workers?
- What optimizations are possible/required?

Role of the Master

- Keeps state regarding the state of each worker machine (pings each machine)
- Reschedules work corresponding to failed machines
- Orchestrates the passing of locations to reduce functions

Discussion

- what are the performance limitations of map reduce?
- what are the constraints imposed on map and reduce functions?
- how would you like to expand the capability of map reduce?

Piccolo

- MapReduce restrictions:
  - just two phases
  - map can see only its split
  - reduce sees just one key at a time
- Piccolo programming model:
  - any number of phases (determined by controller)
  - computation proceeds in rounds:
    - example: page rank
    - global key/value tables store intermediate data

Naive PageRank

```
curr = Table(key=PageID, value=double)
next = Table(key=PageID, value=double)
```

```
def pr_kernel(graph, curr, next):
  i = my_instance
  n = len(graph)/NUM_MACHINES
  for s in graph[(i-1)*n:i*n]
    for t in s.out:
      next[t] += curr[s.id] / len(s.out)
```

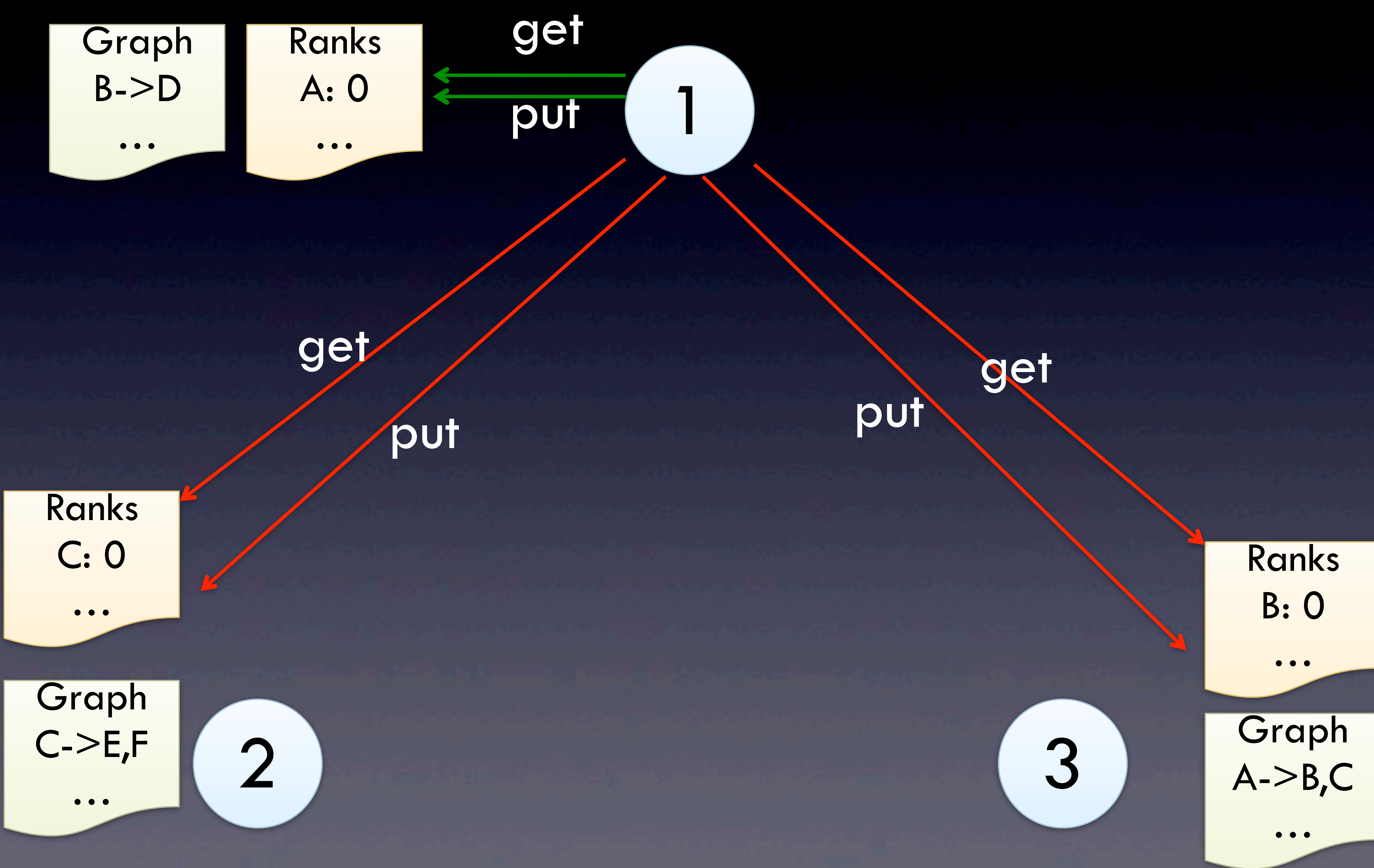
Jobs run by many machines

```
def main():
  for i in range(50):
    launch_jobs(NUM_MACHINES, pr_kernel,
                graph, curr, next)
    swap(curr, next)
    next.clear()
```

Controller launches jobs in parallel

Run by a single controller

Naive PR is Slow



PageRank: Locality

```
curr = Table(…,partitions=100,partition_by=site)
next = Table(…,partitions=100,partition_by=site)
group_tables(curr,next,graph)

def pr_kernel(graph, curr, next):
   for s in graph.get_iterator(my_instance)
      for t in s.out:
         next[t] += curr[s.id] / len(s.out)

def main():
   for i in range(50):
      launch_jobs(curr.num_partitions,
                  pr_kernel,
                  graph, curr, next,
                  locality=curr)
      swap(curr, next)
      next.clear()
```

Control table partitioning

Co-locate tables

Co-locate execution with table

PageRank: Synchronization

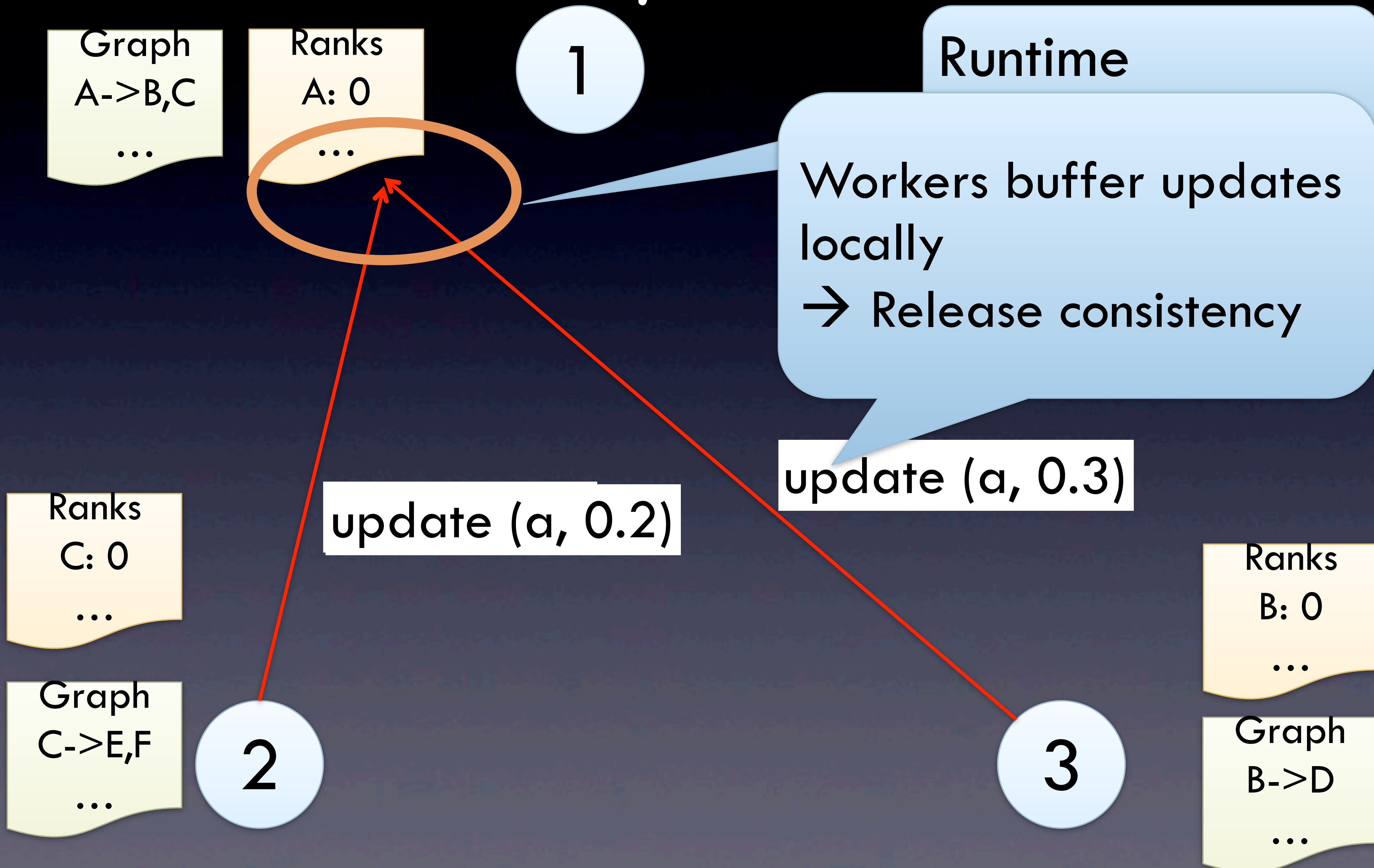
```
curr = Table(…,partition_by=site,accumulate=sum)
next = Table(…,partition_by=site,accumulate=sum)
group_tables(curr,next,graph)

def pr_kernel(graph, curr, next):
  for s in graph.get_iterator(my_instance)
    for t in s.out:
      next.update(t, curr.get(s.id)/len(s.out))

def main():
  for i in range(50):
    handle = launch_jobs(curr.num_partitions,
                         pr_kernel,
                         graph, curr, next,
                         locality=curr)
    barrier(handle)
    swap(curr, next)
    next.clear()
```

Accumulation via sum

Update invokes accumulation function

Explicitly wait between iterations

Efficient Synchronization



PageRank: Checkpointing

```
curr = Table(…,partition_by=site,accumulate=sum)
next = Table(…,partition_by=site,accumulate=sum)
group_tables(curr,next)
def pr_kernel(graph, curr, next):
  for node in graph.get_iterator(my_instance)
    for t in s.out:
      next.update(t,curr.get(s.id)/len(s.out))

def main():
  curr, userdata = restore()
  last = userdata.get('iter', 0)
  for i in range(last,50):
    handle = launch_jobs(curr.num_partitions, pr_kernel,
                         graph, curr, next,
                         locality=curr)
    cp_barrier(handle, tables=(next), userdata={'iter':i})
    swap(curr, next)
    next.clear()
```

Restore previous computation

User decides which tables to checkpoint and when

- How does Piccolo compare to MapReduce:
  - in terms of programmability
  - in terms of performance (stragglers, load balance, etc.)
  - in terms of fault tolerance