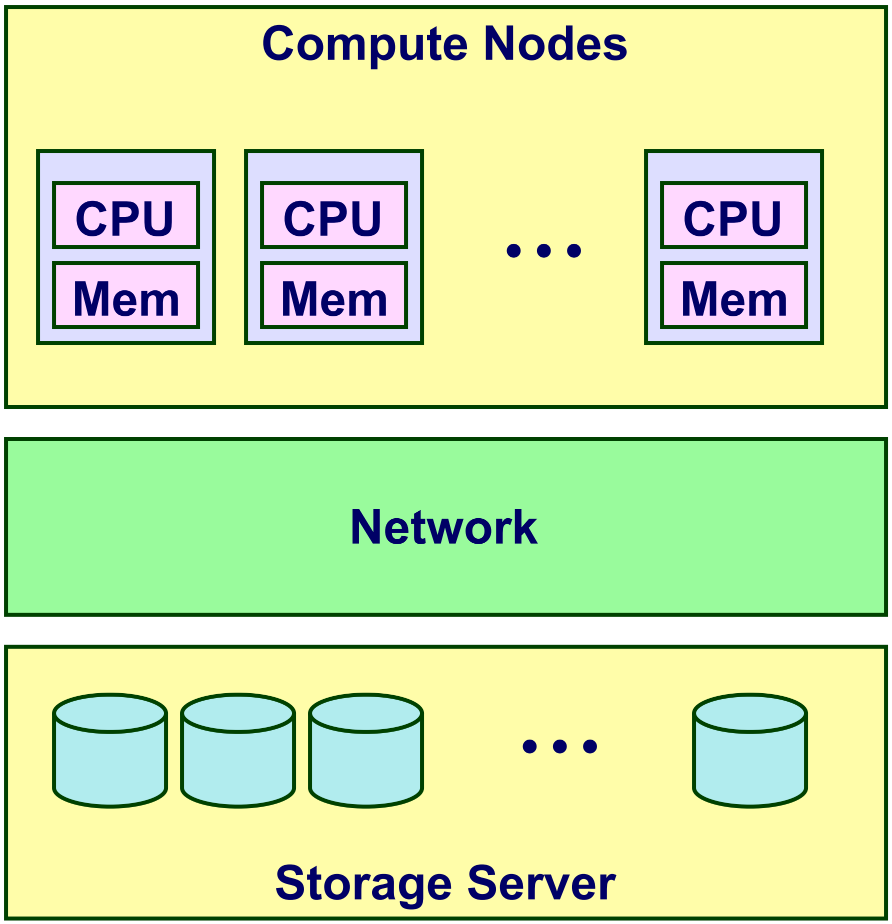
15-440

MapReduce Programming
Oct 30, 2012

Topics

- **Large-scale computing**
  - Traditional high-performance computing (HPC)
  - Cluster computing
- **MapReduce**
  - Definition
  - Examples
- **Implementation**
- **Alternatives to MapReduce**
- **Properties**

Typical HPC Machine



Compute Nodes

- High end processor(s)
- Lots of RAM

Network

- Specialized
- Very high performance

Storage Server

- RAID-based disk array

HPC Machine Example

Jaguar Supercomputer

- 3rd fastest in world

Compute Nodes

- 18,688 nodes in largest partition
- 2X 2.6Ghz 6-core AMD Opteron
- 16GB memory
- Total: 2.3 petaflop / 300 TB memory

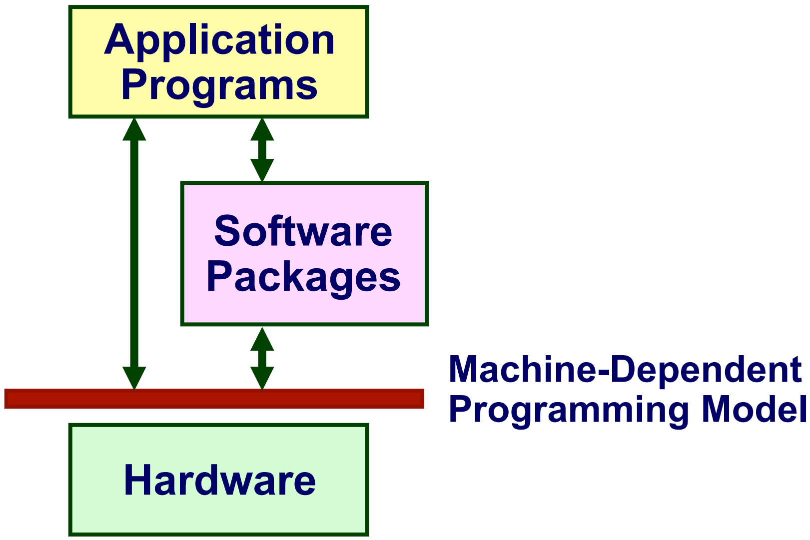
Network

- 3D torus
  - Each node connected to 6 neighbors via 6.0 GB/s links

Storage Server

- 10PB RAID-based disk array

HPC Programming Model



- **Programs described at very low level**
  - Specify detailed control of processing & communications
- **Rely on small number of software packages**
  - Written by specialists
  - Limits classes of problems & solution methods

Bulk Synchronous Programming

Solving Problem Over Grid

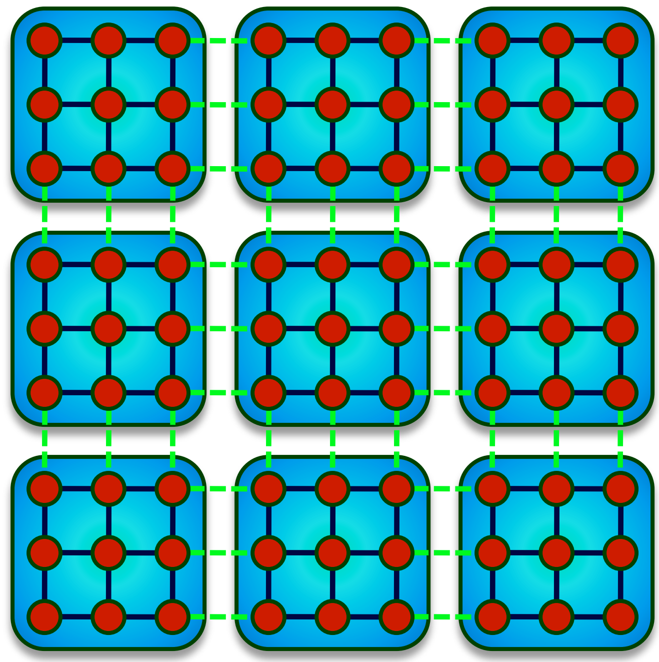
- E.g., finite-element computation

Partition into Regions

- p regions for p processors

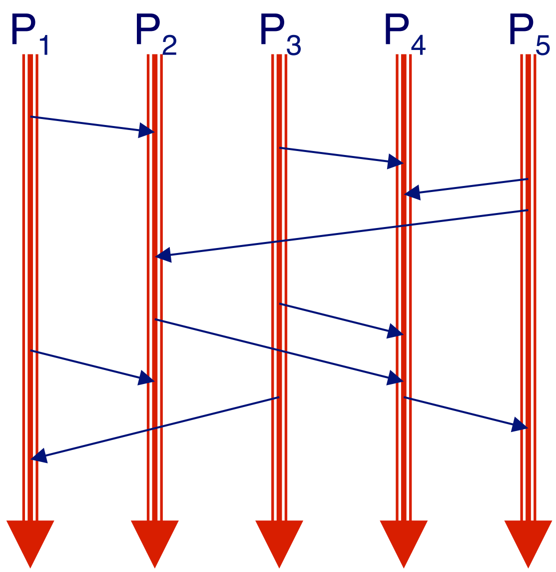
Map Region per Processor

- Local computation sequential
- Periodically communicate boundary values with neighbors



Typical HPC Operation

Message Passing



Characteristics

- Long-lived processes
- Make use of spatial locality
- Hold all program data in memory (no disk access)
- High bandwidth communication

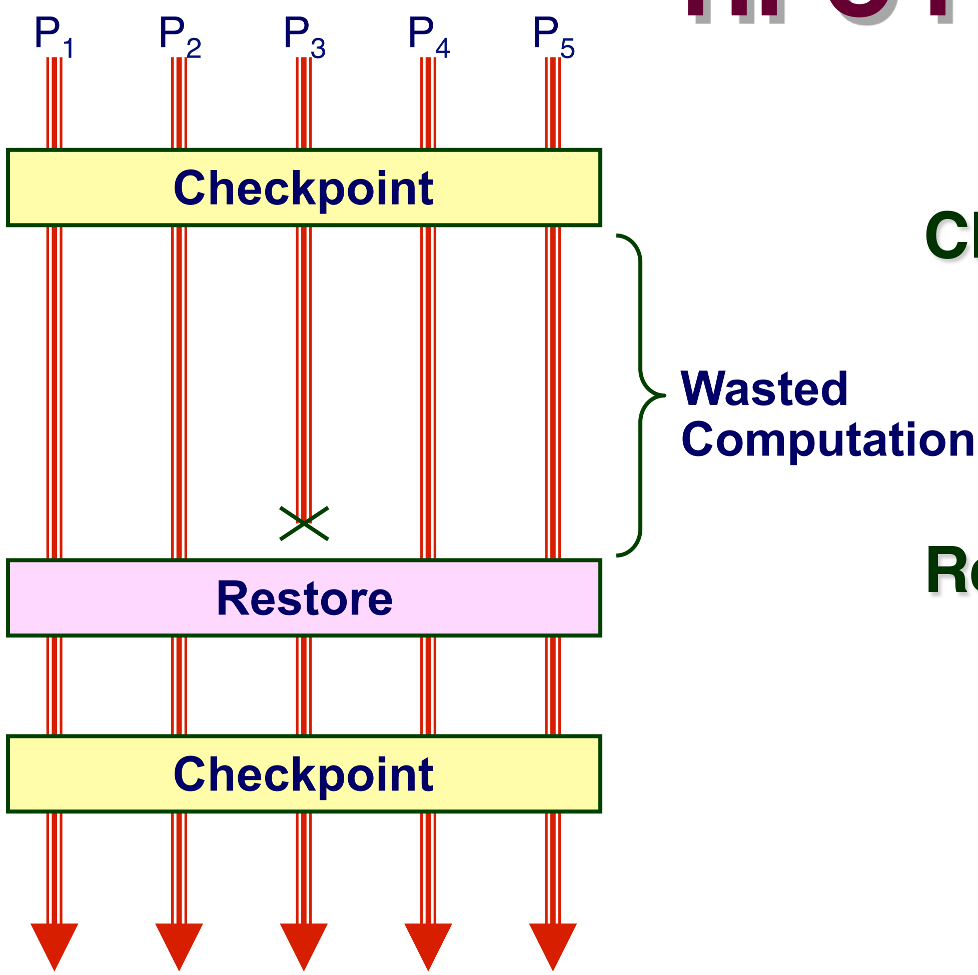
Strengths

- High utilization of resources
- Effective for many scientific applications

Weaknesses

- Requires careful tuning of application to resources
- Intolerant of any variability

HPC Fault Tolerance



Checkpoint

- Periodically store state of all processes
- Significant I/O traffic

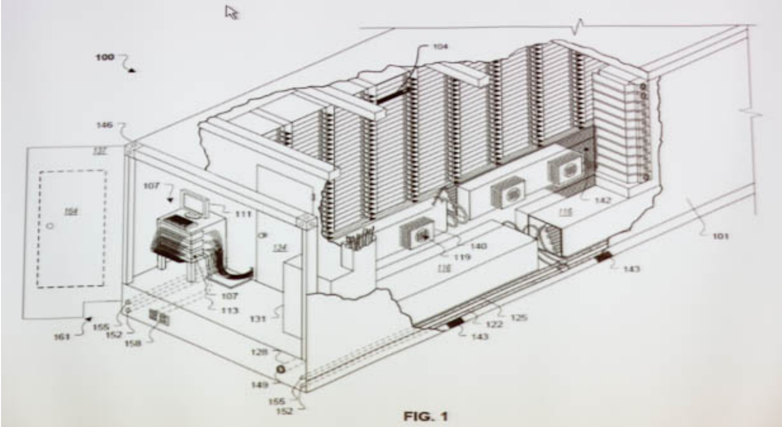
Restore

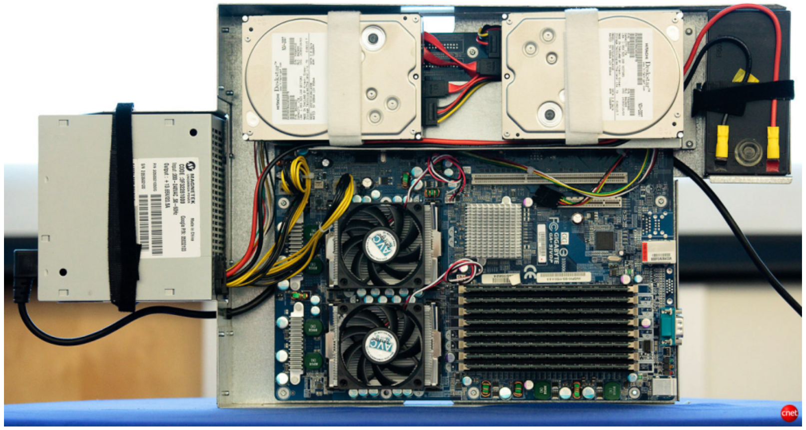
- When failure occurs
- Reset state to that of last checkpoint
- All intervening computation wasted

Performance Scaling

- Very sensitive to number of failing components

Google Data Centers



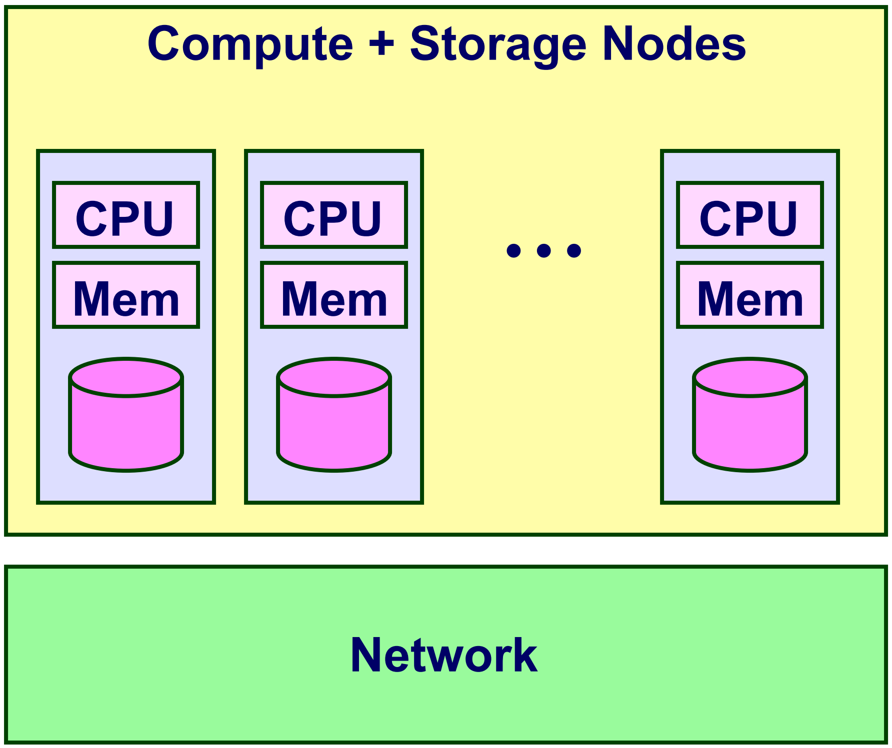


Dalles, Oregon

- Hydroelectric power @ 2¢ / KW Hr
- 50 Megawatts
  - Enough to power 60,000 homes

- Engineered for maximum modularity & power efficiency
- Container: 1160 servers, 250KW
- Server: 2 disks, 2 processors

Typical Cluster Machine



Compute + Storage Nodes

- Medium-performance processors
- Modest memory
- 1-2 disks

Network

- Conventional Ethernet switches
  - 10 Gb/s within rack
  - 100 Gb/s across racks

Machines with Disks

Lots of storage for cheap

- Seagate Barracuda
- 3 TB @ $130
  (4.3¢ / GB)
- Compare 2007:
  0.75 TB @ $266
  35¢ / GB

Seagate Barracuda 7200 3 TB 7200RPM SATA 6 Gb/s NCQ 64MB Cache 3.5-Inch Internal Bare Drive ST3000DM001
by Seagate
(378 customer reviews) | **#1 Best Seller** in Internal Hard Drives

List Price: ~~$269.99~~
Price: $129.99 & this item ships for **FREE with Super Saver Shipping**. Details
You Save: $140.00 (52%)

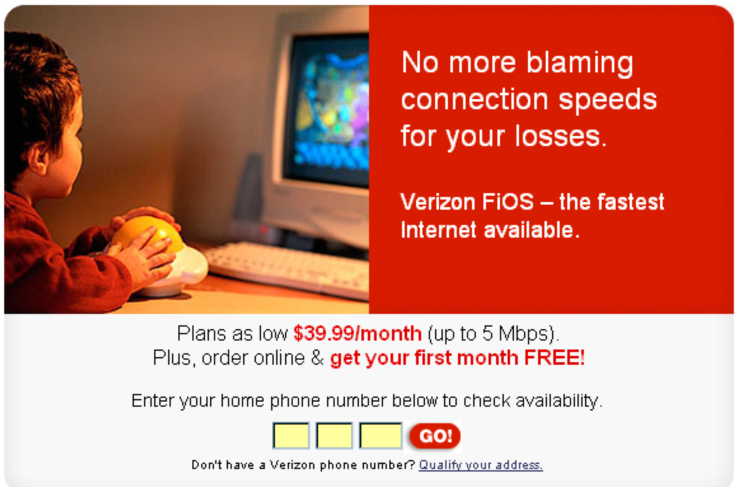
In Stock.
Ships from and sold by **Amazon.com** in certified Frustration-Free Packaging. Gift-wrap available.

Drawbacks

- Long and highly variable delays
- Not very reliable

Not included in HPC Nodes

Oceans of Data, Skinny Pipes



Seagate

1 Terabyte

- Easy to store
- Hard to move

Disks	MB / s	Time
Seagate Barracuda	115	2.3 hours
Seagate Cheetah	125	2.2 hours
Networks	**MB / s**	**Time**
Home Internet	< 0.625	> 18.5 days
Gigabit Ethernet	< 125	> 2.2 hours
PSC Teragrid Connection	< 3,750	> 4.4 minutes

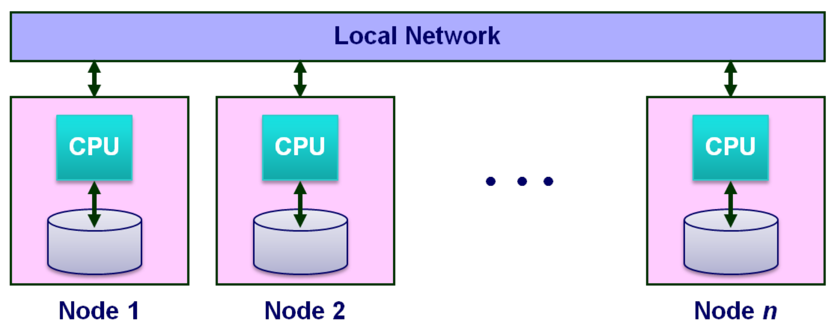
Data-Intensive System Challenge

For Computation That Accesses 1 TB in 5 minutes

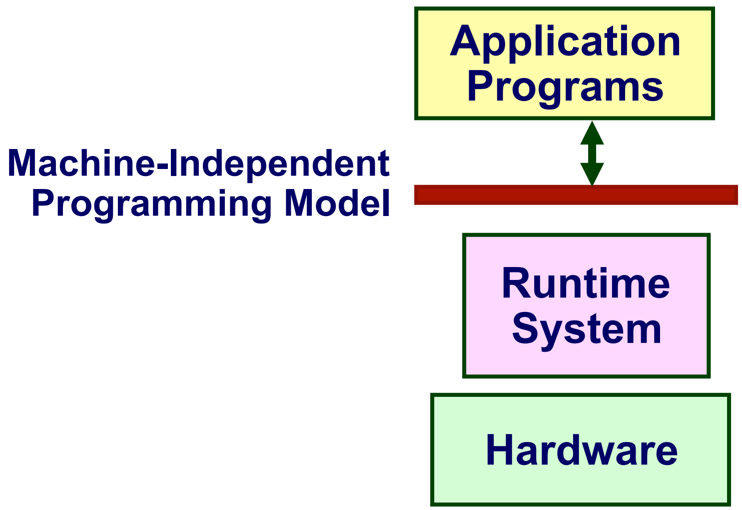
- Data distributed over 100+ disks
  - Assuming uniform data partitioning
- Compute using 100+ processors
- Connected by gigabit Ethernet (or equivalent)

System Requirements

- Lots of disks
- Lots of processors
- Located in close proximity
  - Within reach of fast, local-area network

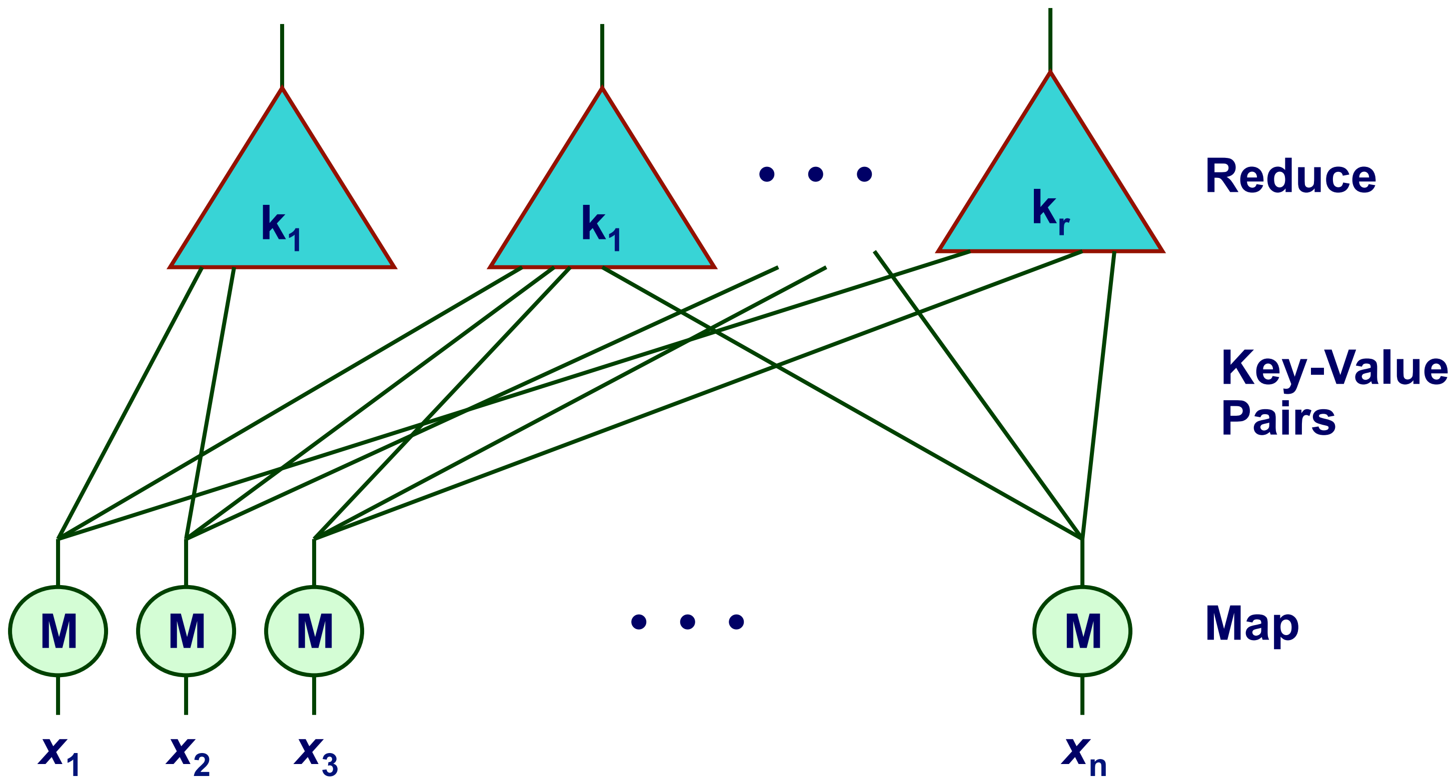


Ideal Cluster Programming Model



- Application programs written in terms of high-level operations on data
- Runtime system controls scheduling, load balancing, …

Map/Reduce Programming Model



- **Map computation across many objects**
  - E.g., 10^{10} Internet web pages
- **Aggregate results in many different ways**
- **System deals with issues of resource allocation & reliability**

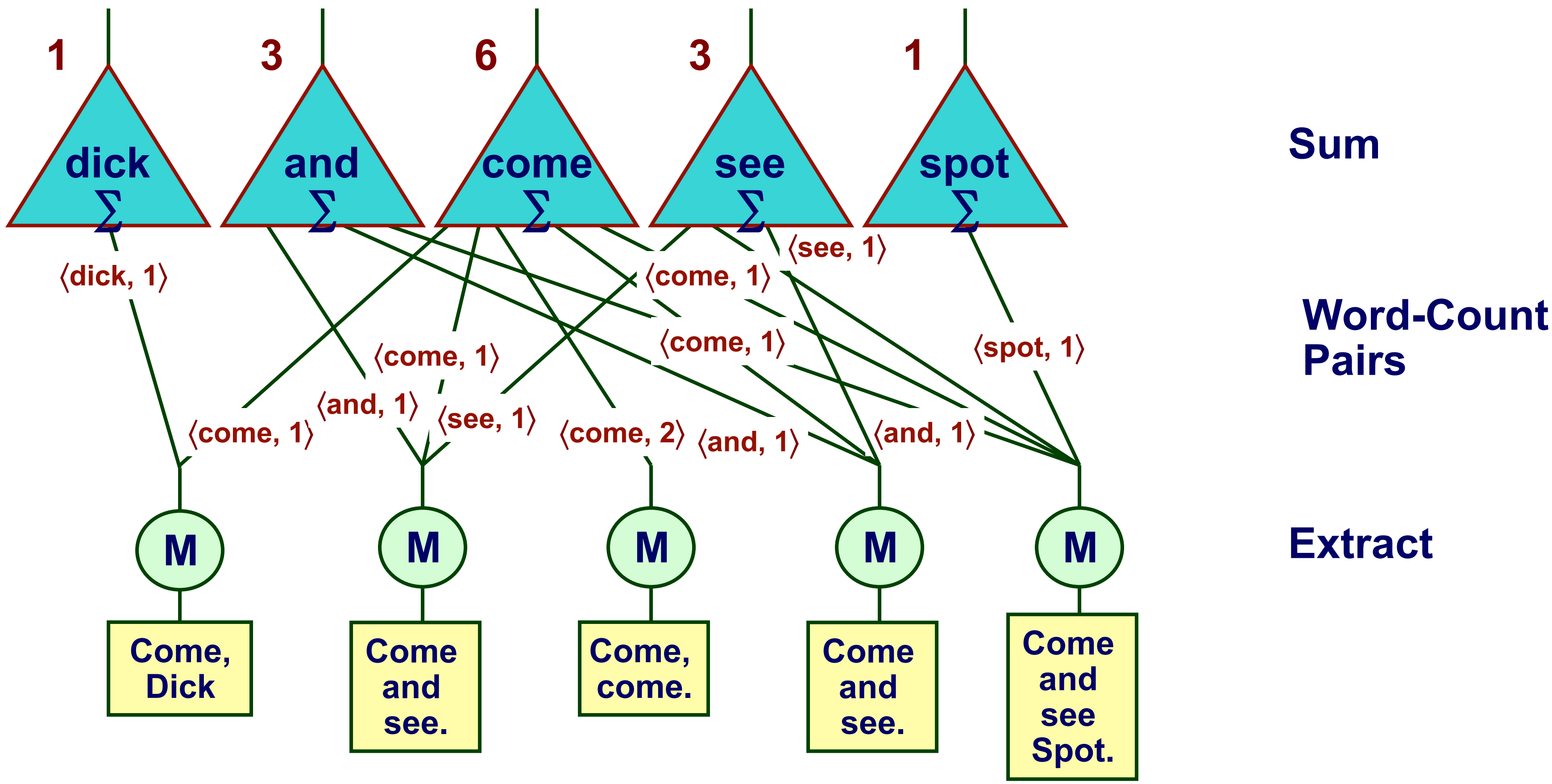
Dean & Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004

MapReduce Example

Come, Dick	Come and see.	Come, come.	Come and see.	Come and see Spot.

- Create an word index of set of documents

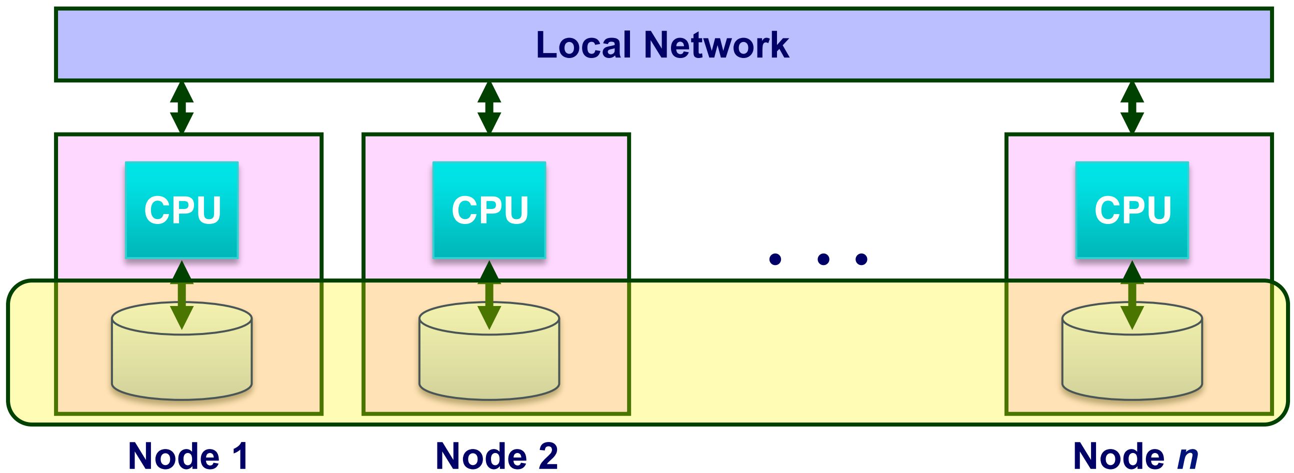
MapReduce Example



- Map: generate ⟨word, count⟩ pairs for all words in document
- Reduce: sum word counts across documents

Hadoop Project

File system with files distributed across nodes



- **Store multiple (typically 3 copies of each file)**
  - If one node fails, data still available
- **Logically, any node has access to any file**
  - May need to fetch across network

Map / Reduce programming environment

- **Software manages execution of tasks on nodes**

Hadoop MapReduce API

Requirements

- Programmer must supply Mapper & Reducer classes

Mapper

- Steps through file one line at a time
- Code generates sequence of <key, value> pairs
  - Call output.collect(key, value)
- Default types for keys & values are strings
  - Lots of low-level machinery to convert to & from other data types
  - But can use anything "writable"

Reducer

- Given key + iterator that generates sequence of values
- Generate one or more <key, value> pairs
  - Call output.collect(key, value)

Hadoop Word Count Mapper

```
public class WordCountMapper extends MapReduceBase
        implements Mapper {

    private final static Text word = new Text();

    private final static IntWritable count = new IntWritable(1);

    public void map(WritableComparable key, Writable values,
                    OutputCollector output, Reporter reporter)
                throws IOException {
        /* Get line from file */
        String line = values.toString();
        /* Split into tokens */
        StringTokenizer itr = new StringTokenizer(line.toLowerCase(),
                                    " \t.!?:()[],'&-;|0123456789");
        while(itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            /* Emit <token,1> as key + value
            output.collect(word, count);
        }
    }

}
```

Hadoop Word Count Reducer

```
public class WordCountReducer extends MapReduceBase
        implements Reducer {

        public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output, Reporter reporter)
                       throws IOException {
            int cnt = 0;
            while(values.hasNext()) {
                IntWritable ival = (IntWritable) values.next();
                cnt += ival.get();
            }
            output.collect(key, new IntWritable(cnt));
        }

}
```

Cluster Scalability Advantages

- **Distributed system design principles lead to scalable design**
- **Dynamically scheduled tasks with state held in replicated files**

Provisioning Advantages

- **Can use consumer-grade components**
  - maximizes cost-peformance
- **Can have heterogenous nodes**
  - More efficient technology refresh

Operational Advantages

- **Minimal staffing**
- **No downtime**

Example: Sparse Matrices with Map/Reduce

$$
\overset{\mathbf{A}}{\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}}
\;\mathbf{X}\;
\overset{\mathbf{B}}{\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}}
\;=\;
\overset{\mathbf{C}}{\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}}
$$

- Task: Compute product C = A·B
- Assume most matrix entries are 0

Motivation

- Core problem in scientific computing
- Challenging for parallel execution
- Demonstrate expressiveness of Map/Reduce

Computing Sparse Matrix Product

A

$$\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}$$

$$1 \xrightarrow[\mathtt{A}]{10} 1$$
$$1 \xrightarrow[\mathtt{A}]{20} 3$$
$$2 \xrightarrow[\mathtt{A}]{30} 2$$
$$2 \xrightarrow[\mathtt{A}]{40} 3$$
$$3 \xrightarrow[\mathtt{A}]{50} 1$$
$$3 \xrightarrow[\mathtt{A}]{60} 2$$
$$3 \xrightarrow[\mathtt{A}]{70} 3$$

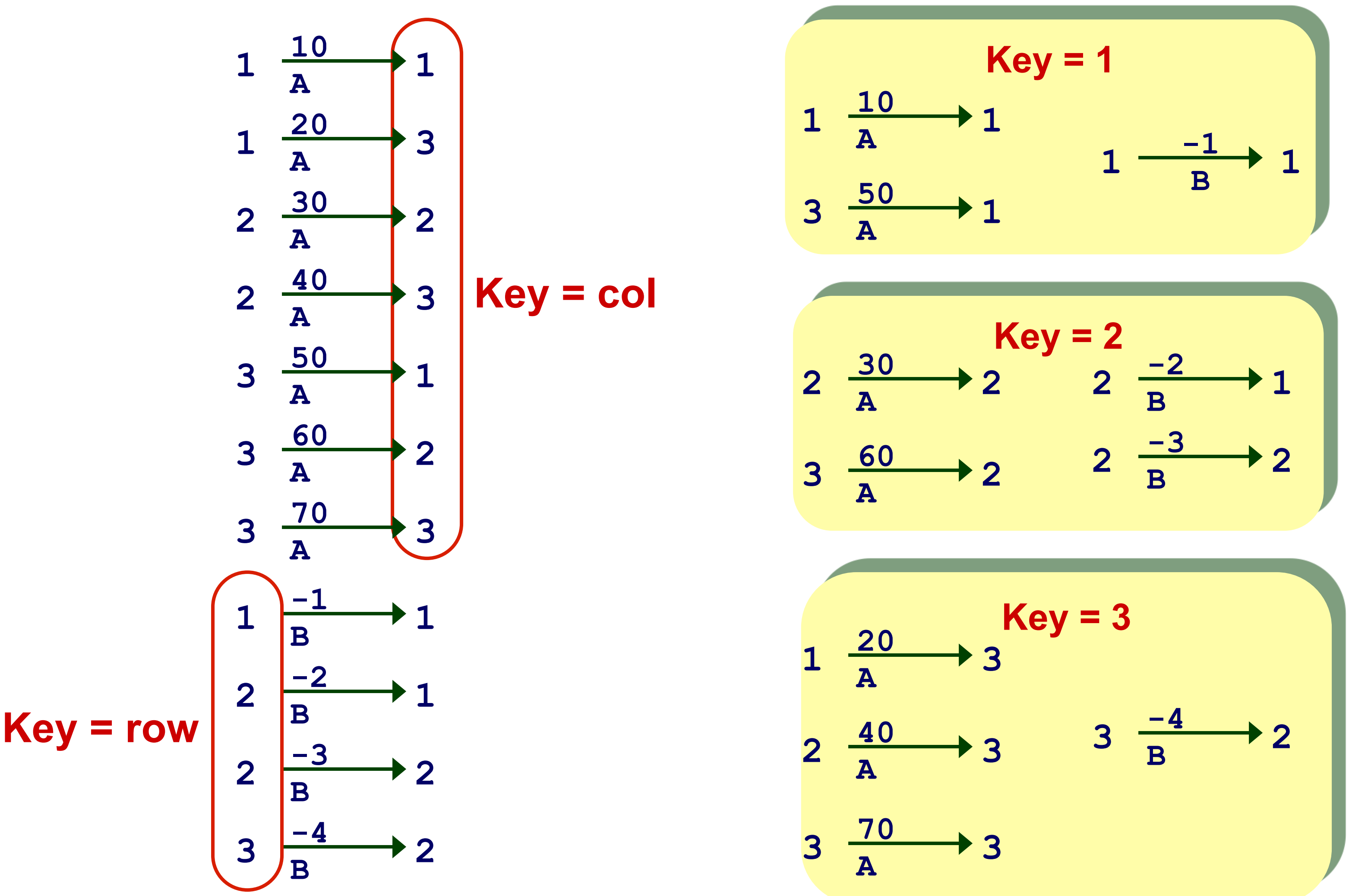
B

$$\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}$$

$$1 \xrightarrow[\mathtt{B}]{-1} 1$$
$$2 \xrightarrow[\mathtt{B}]{-2} 1$$
$$2 \xrightarrow[\mathtt{B}]{-3} 2$$
$$3 \xrightarrow[\mathtt{B}]{-4} 2$$

- **Represent matrix as list of nonzero entries**
  ⟨row, col, value, matrixID⟩
- **Strategy**
  - **Phase 1: Compute all products $a_{i,k} \cdot b_{k,j}$**
  - **Phase 2: Sum products for each entry i,j**
  - **Each phase involves a Map/Reduce**

Phase 1 Map of Matrix Multiply



- Group values $a_{i,k}$ and $b_{k,j}$ according to key k

Phase 1 "Reduce" of Matrix Multiply

Key = 1

$1 \xrightarrow[A]{10} 1$

$3 \xrightarrow[A]{50} 1$

X

$1 \xrightarrow[B]{-1} 1$

Key = 2

$2 \xrightarrow[A]{30} 2$

$3 \xrightarrow[A]{60} 2$

X

$2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

Key = 3

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{70} 3$

X

$3 \xrightarrow[B]{-4} 2$

$1 \xrightarrow[C]{-10} 1$

$3 \xrightarrow[A]{-50} 1$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-90} 2$

$3 \xrightarrow[C]{-120} 1$

$3 \xrightarrow[C]{-180} 2$

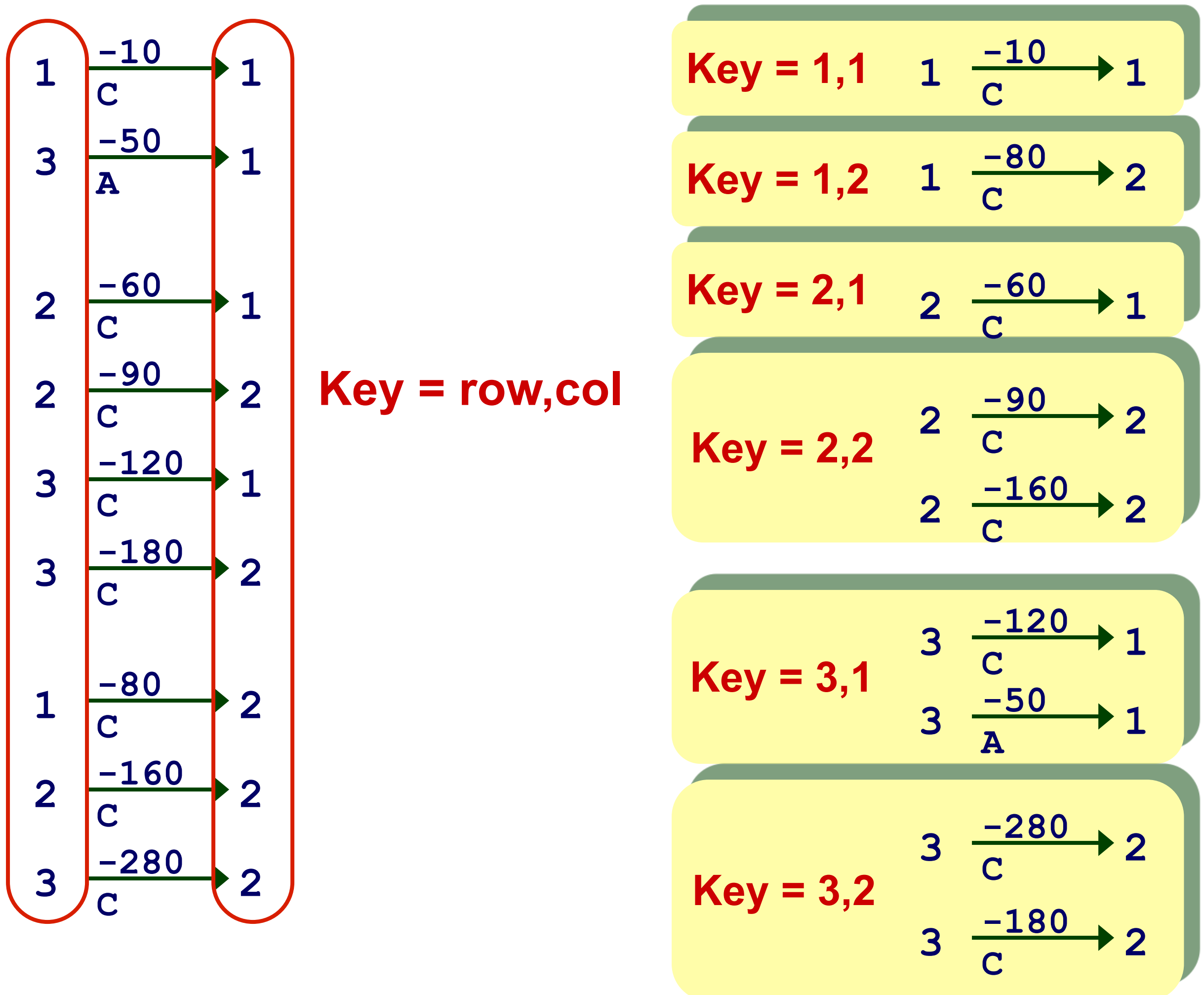
$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-160} 2$

$3 \xrightarrow[C]{-280} 2$

- Generate all products $a_{i,k} \cdot b_{k,j}$

Phase 2 Map of Matrix Multiply



- **Group products $a_{i,k} \cdot b_{k,j}$ with matching values of i and j**

Phase 2 Reduce of Matrix Multiply

Key = 1,1	$1 \xrightarrow[C]{-10} 1$	$1 \xrightarrow[C]{-10} 1$
Key = 1,2	$1 \xrightarrow[C]{-80} 2$	$1 \xrightarrow[C]{-80} 2$
Key = 2,1	$2 \xrightarrow[C]{-60} 1$	$2 \xrightarrow[C]{-60} 1$
Key = 2,2	$2 \xrightarrow[C]{-90} 2$ $2 \xrightarrow[C]{-160} 2$	$2 \xrightarrow[C]{-250} 2$
Key = 3,1	$3 \xrightarrow[C]{-120} 1$ $3 \xrightarrow[A]{-50} 1$	$3 \xrightarrow[C]{-170} 1$
Key = 3,2	$3 \xrightarrow[C]{-280} 2$ $3 \xrightarrow[C]{-180} 2$	$3 \xrightarrow[C]{-460} 2$

C

$$\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

- Sum products to get final entries

Matrix Multiply Phase 1 Mapper

```
public class P1Mapper extends MapReduceBase implements Mapper {

    public void map(WritableComparable key, Writable values,
                    OutputCollector output, Reporter reporter) throws
IOException {
        try {
            GraphEdge e = new GraphEdge(values.toString());
            IntWritable k;
            if (e.tag.equals("A"))
                k = new IntWritable(e.toNode);
            else
                k = new IntWritable(e.fromNode);
            output.collect(k, new Text(e.toString()));
        } catch (BadGraphException e) {}
    }
}
```

Matrix Multiply Phase 1 Reducer

```
public class P1Reducer extends MapReduceBase implements Reducer {

        public void reduce(WritableComparable key, Iterator values,
                        OutputCollector output, Reporter reporter)
                        throws IOException
    {
        Text outv = new Text(""); // Don't really need output values
        /* First split edges into A and B categories */
        LinkedList<GraphEdge> alist = new LinkedList<GraphEdge>();
        LinkedList<GraphEdge> blist = new LinkedList<GraphEdge>();
        while(values.hasNext()) {
                try {
                    GraphEdge e =
                        new GraphEdge(values.next().toString());
                    if (e.tag.equals("A")) {
                        alist.add(e);
                    } else {
                        blist.add(e);
                    }
                } catch (BadGraphException e) {}
        }
        // Continued
```

MM Phase 1 Reducer (cont.)

```
            // Continuation

            Iterator<GraphEdge> aset = alist.iterator();
            // For each incoming edge
            while(aset.hasNext()) {
                GraphEdge aedge = aset.next();
                // For each outgoing edge
                Iterator<GraphEdge> bset = blist.iterator();
                while (bset.hasNext()) {
                    GraphEdge bedge = bset.next();
                    GraphEdge newe = aedge.contractProd(bedge);
                    // Null would indicate invalid contraction
                    if (newe != null) {
                        Text outk = new Text(newe.toString());
                        output.collect(outk, outv);
                    }
                }
            }
    }
}
```

Matrix Multiply Phase 2 Mapper

```
public class P2Mapper extends MapReduceBase implements Mapper {

    public void map(WritableComparable key, Writable values,
                    OutputCollector output, Reporter reporter)
                        throws IOException {
        String es = values.toString();
        try {
            GraphEdge e = new GraphEdge(es);
            // Key based on head & tail nodes
            String ks = e.fromNode + " " + e.toNode;
            output.collect(new Text(ks), new Text(e.toString()));
        } catch (BadGraphException e) {}

    }
}
```

Matrix Multiply Phase 2 Reducer

```
public class P2Reducer extends MapReduceBase implements Reducer {

        public void reduce(WritableComparable key, Iterator values,
                           OutputCollector output, Reporter reporter)
                                throws IOException
    {
        GraphEdge efinal = null;
        while (efinal == null && values.hasNext()) {
            try {
                efinal = new GraphEdge(values.next().toString());
            } catch (BadGraphException e) {}
        }
        if (efinal != null) {
            while(values.hasNext()) {
                try {
                    GraphEdge eother =
                        new GraphEdge(values.next().toString());
                    efinal.weight += eother.weight;
                } catch (BadGraphException e) {}
            }
            if (efinal.weight != 0)
                output.collect(new Text(efinal.toString()),
                        new Text(""));
        }
    }
}
```

Lessons from Sparse Matrix Example

Associative Matching is Powerful Communication Primitive

- Intermediate step in Map/Reduce

Similar Strategy Applies to Other Problems

- Shortest path in graph
- Database join

Many Performance Considerations

- Kiefer, Volk, Lehner, TU Dresden
- Should do systematic comparison to other sparse matrix implementations

MapReduce Implementation

Built on Top of Parallel File System

- Google: GFS, Hadoop: HDFS
- Provides global naming
- Reliability via replication (typically 3 copies)

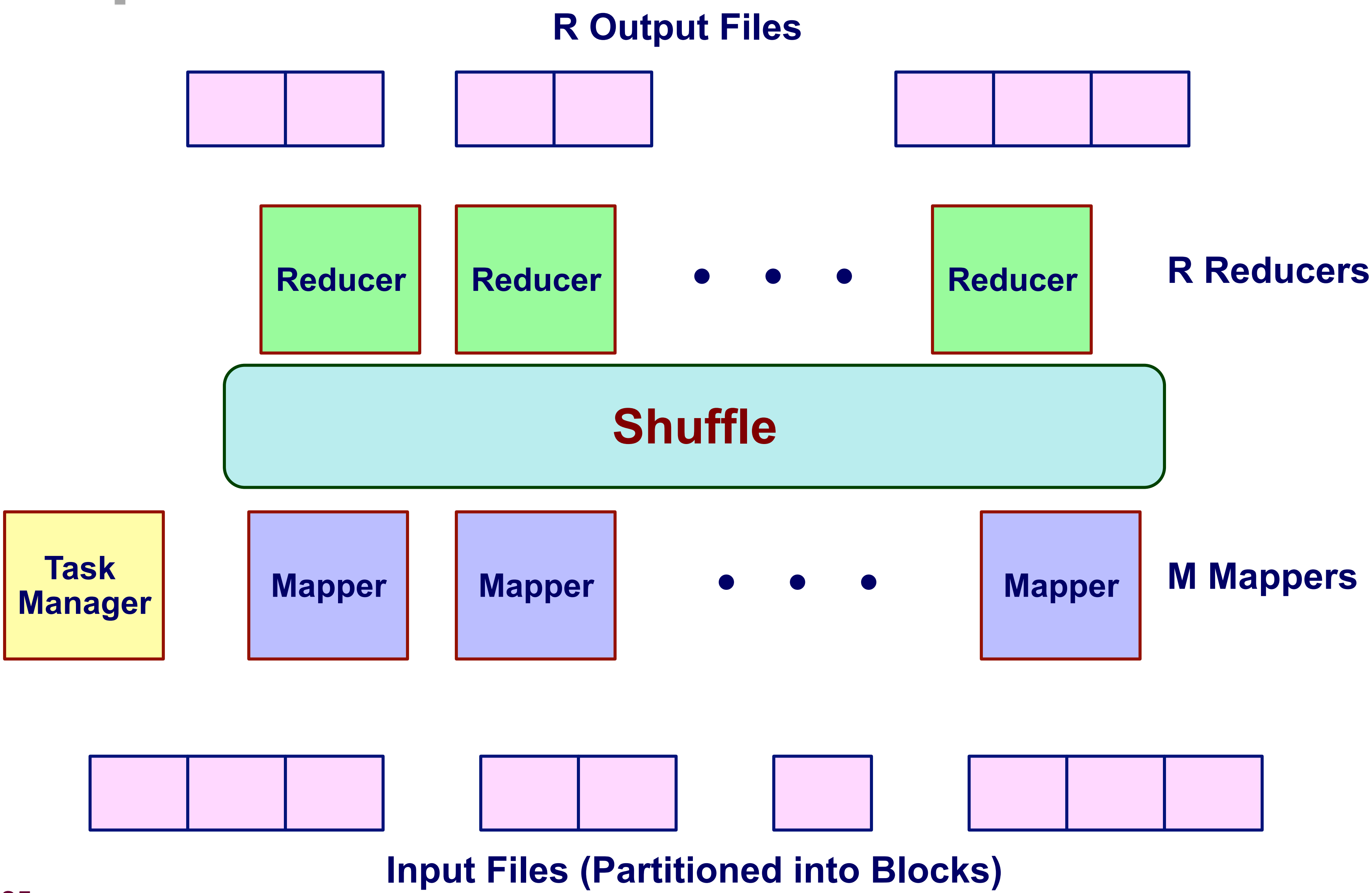
Breaks work into tasks

- Master schedules tasks on workers dynamically
- Typically #tasks >> #processors

Net Effect

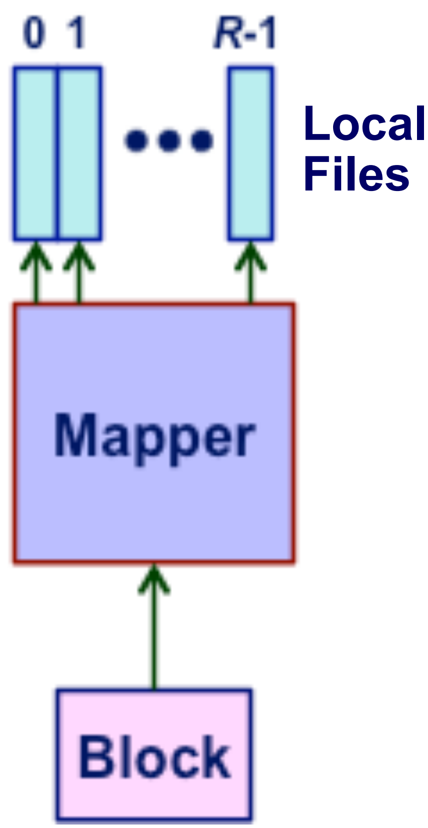
- Input: Set of files in reliable file system
- Output: Set of files in reliable file system

MapReduce Execution



Mapping

$$K \rightarrow \boxed{h} \rightarrow h(K) \in \{0,\ldots,R-1\}$$



Hash Function *h*

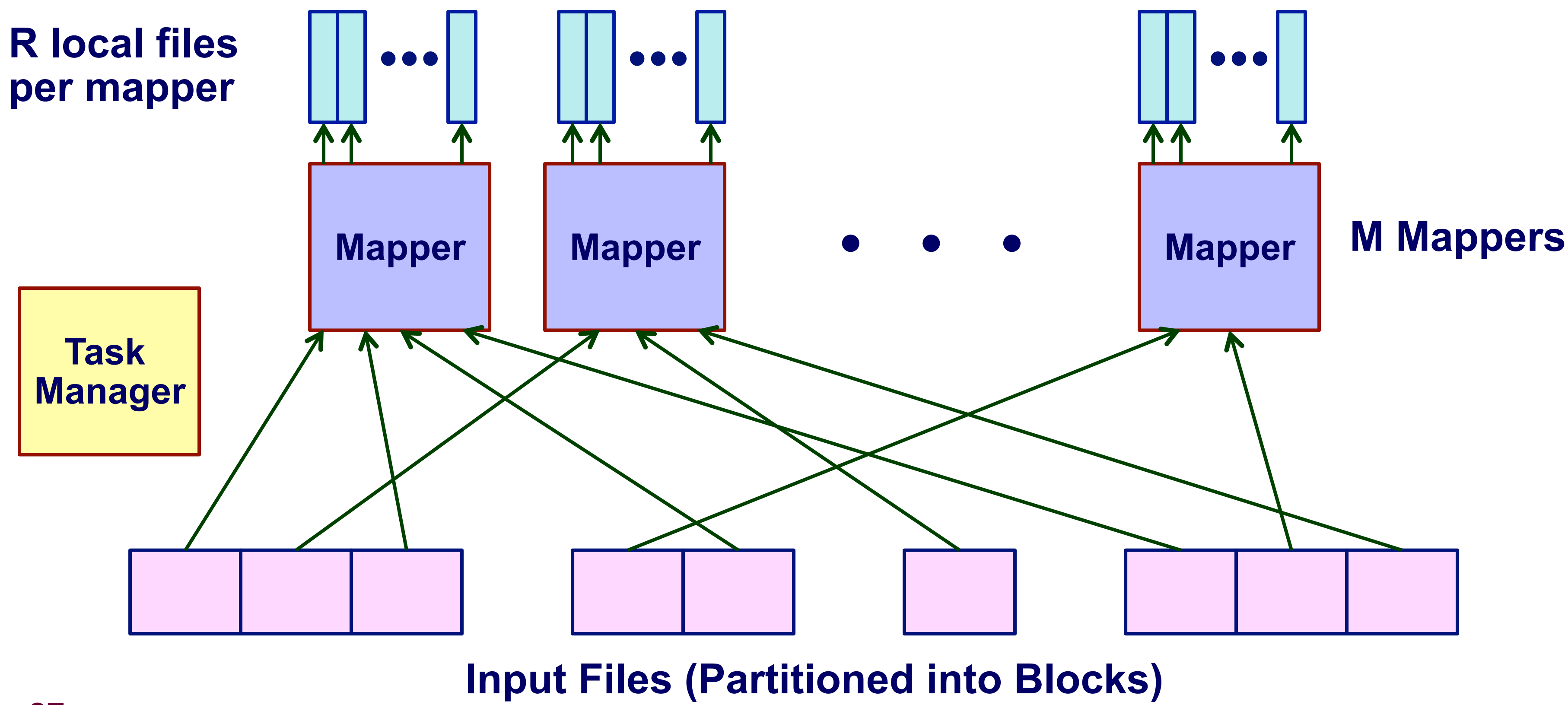
- Maps each key *K* to integer *i* such that $0 \leq i < R$

Mapper Operation

- Reads input file blocks
- Generates pairs $\langle K, V \rangle$
- Writes to local file $h(K)$

Mapping

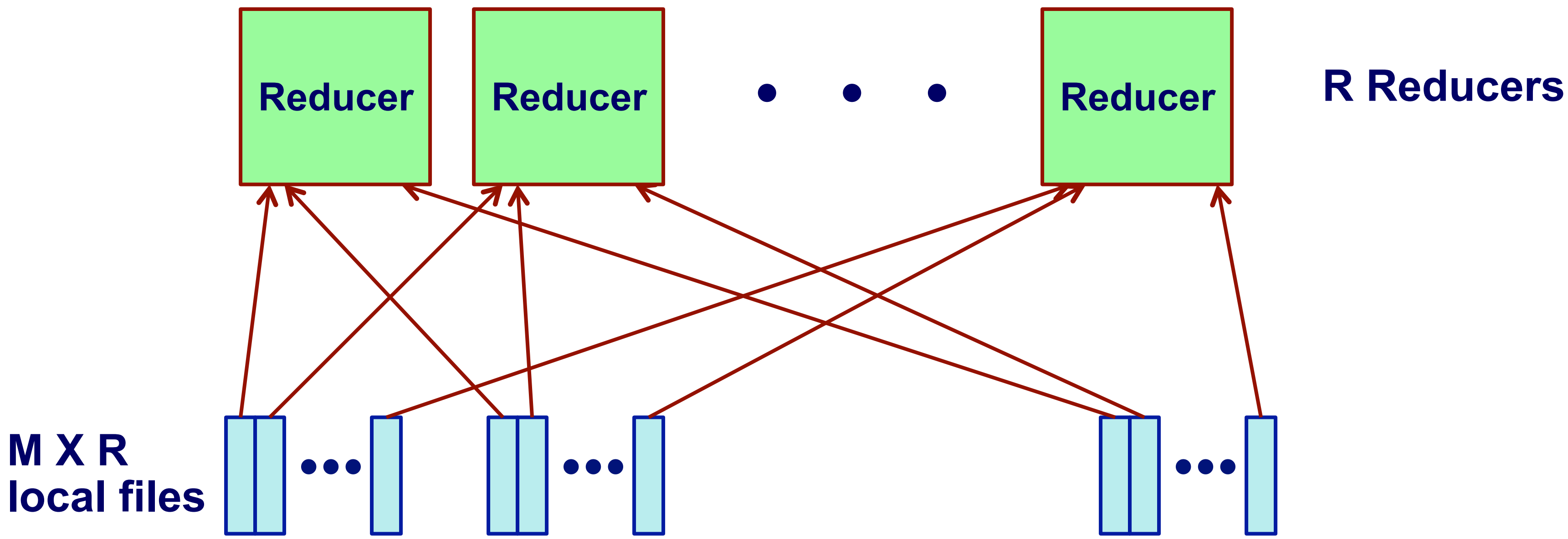
- Dynamically map input file blocks onto mappers
- Each generates key/value pairs from its blocks
- Each writes R files on local file system



Shuffling

Each Reducer:

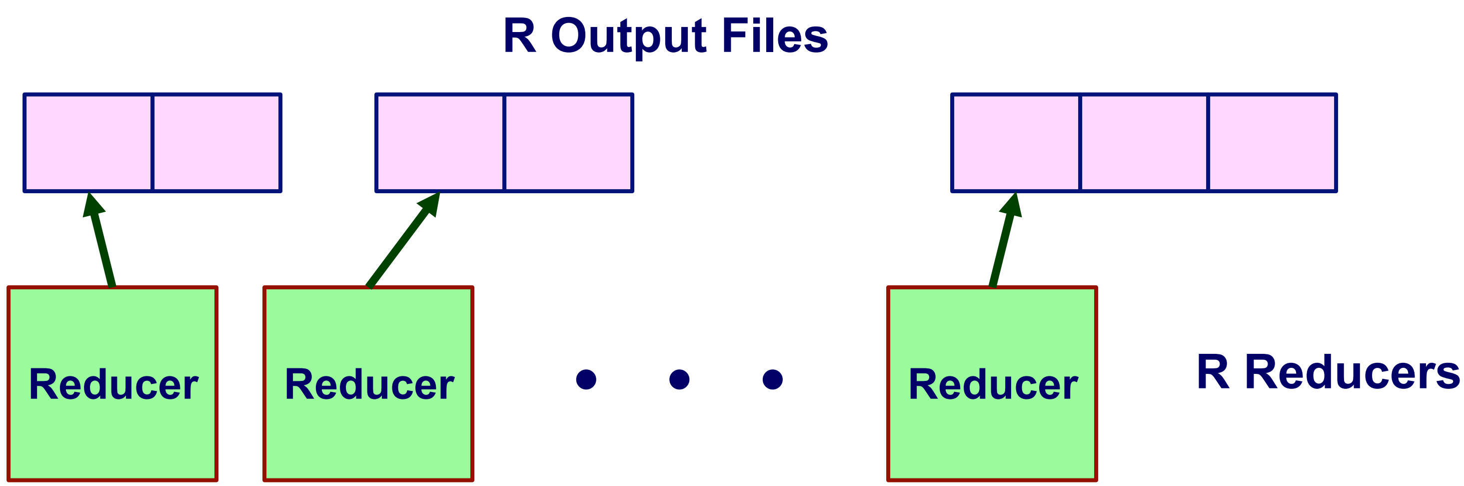
- Handles 1/R of the possible key values
- Fetches its file from each of M mappers
- Sorts all of its entries to group values by keys



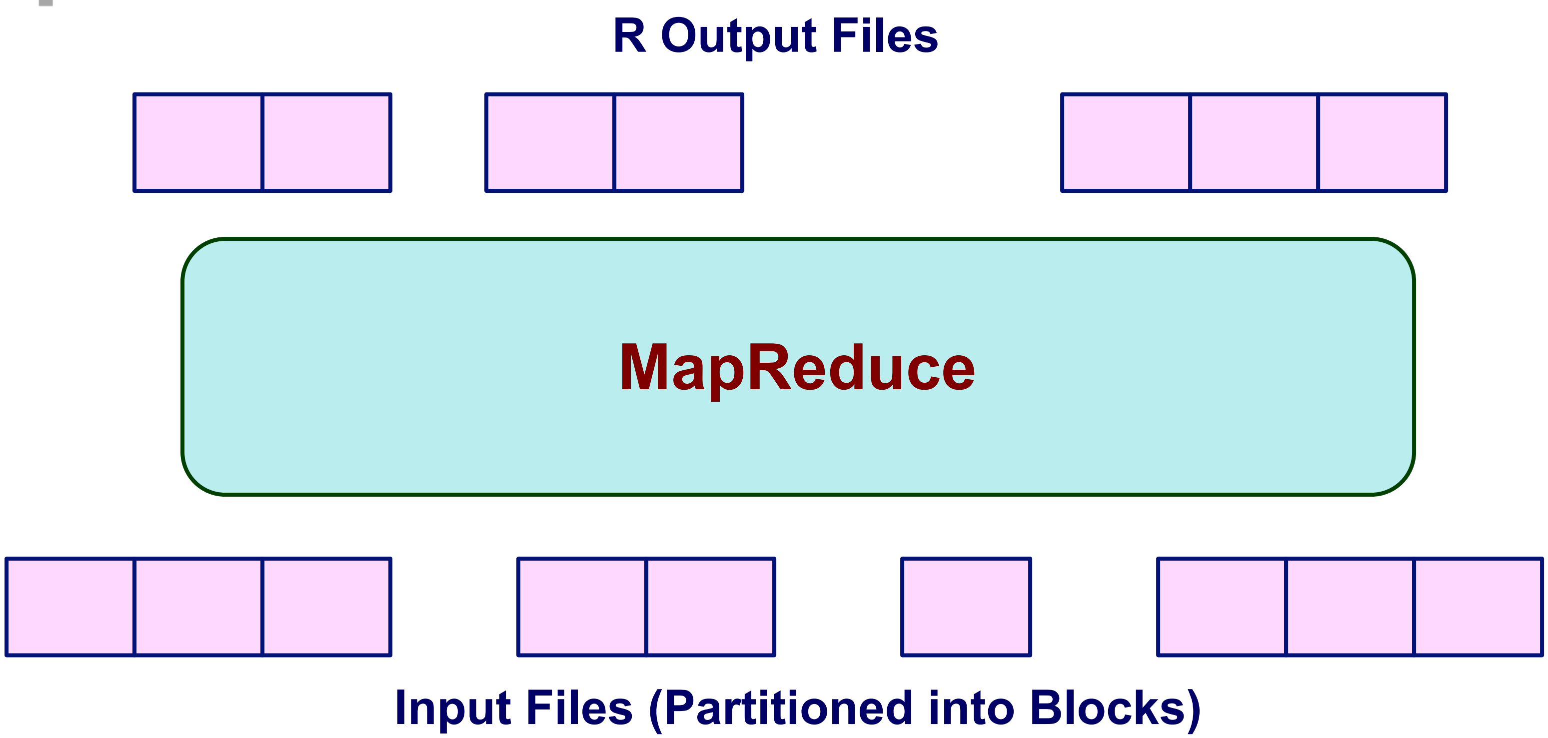
Reducing

Each Reducer:

- Executes reducer function for each key
- Writes output values to parallel file system



MapReduce Effect

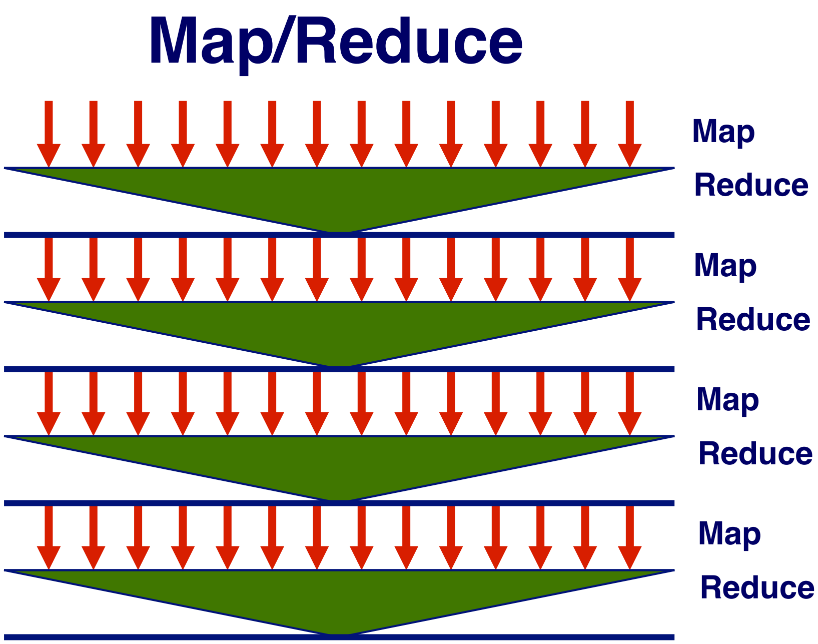


MapReduce Step

- Reads set of files from file system
- Generates new set of files

Can iterate to do more complex processing

Map/Reduce Operation



Characteristics

- Computation broken into many, short-lived tasks
  - Mapping, reducing
- Use disk storage to hold intermediate results

Strengths

- Great flexibility in placement, scheduling, and load balancing
- Can access large data sets

Weaknesses

- Higher overhead
- Lower raw performance

Example Parameters

Sort Benchmark

- 10^{10} 100-byte records
- Partition into M = 15,000 64MB pieces
  - Key = value
  - Partition according to most significant bytes
- Sort locally with R = 4,000 reducers

Machine

- 1800 2Ghz Xeons
- Each with 2 160GB IDE disks
- Gigabit ethernet
- 891 seconds total

Interesting Features

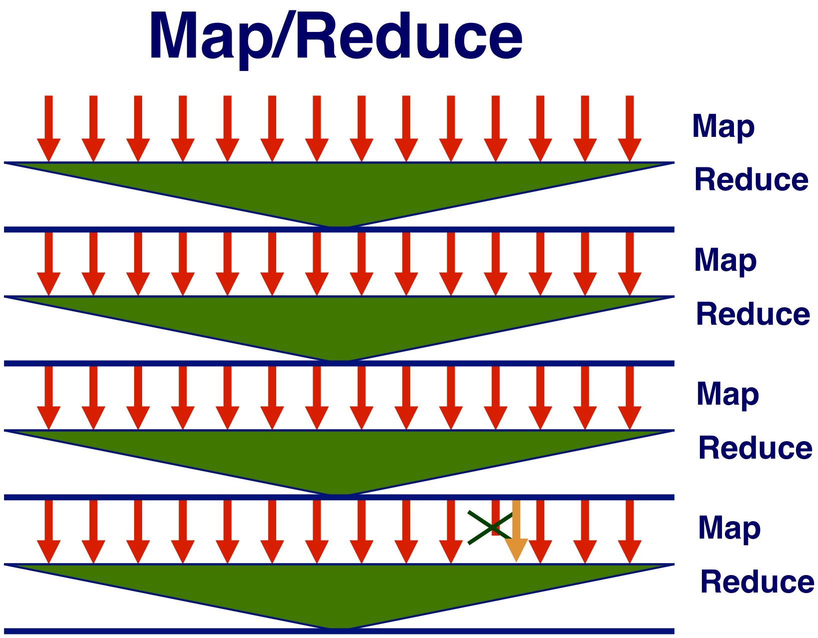
Fault Tolerance

- **Assume reliable file system**
- **Detect failed worker**
  - Heartbeat mechanism
- **Reschedule failed task**

Stragglers

- **Tasks that take long time to execute**
- **Might be bug, flaky hardware, or poor partitioning**
- **When done with most tasks, reschedule any remaining executing tasks**
  - Keep track of redundant executions
  - Significantly reduces overall run time

Map/Reduce Fault Tolerance



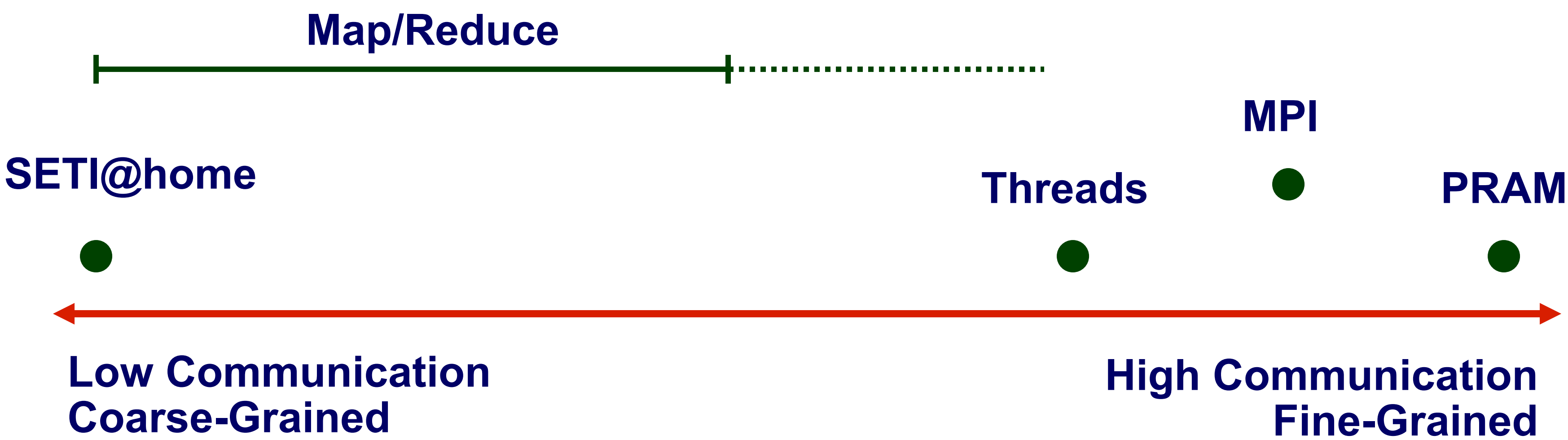
Data Integrity

- Store multiple copies of each file
- Including intermediate results of each Map / Reduce
  - Continuous checkpointing

Recovering from Failure

- Simply recompute lost result
  - Localized effect
- Dynamic scheduler keeps all processors busy

Exploring Parallel Computation Models



Map/Reduce Provides Coarse-Grained Parallelism

- Computation done by independent processes
- File-based communication

Observations

- Relatively “natural” programming model
- Research issue to explore full potential and limits

Beyond Map/Reduce

Typical Map/Reduce Applications

- Sequence of steps, each requiring map & reduce
- Series of data transformations
- Iterating until reach convergence

Strengths of Map/Reduce

- User writes simple functions, system manages complexities of mapping, synchronization, fault tolerance
- Very general
- Good for large-scale data analysis

Limitations

- No locality of data or activity
- Each map/reduce step must complete before next begins

Generalizing Map/Reduce

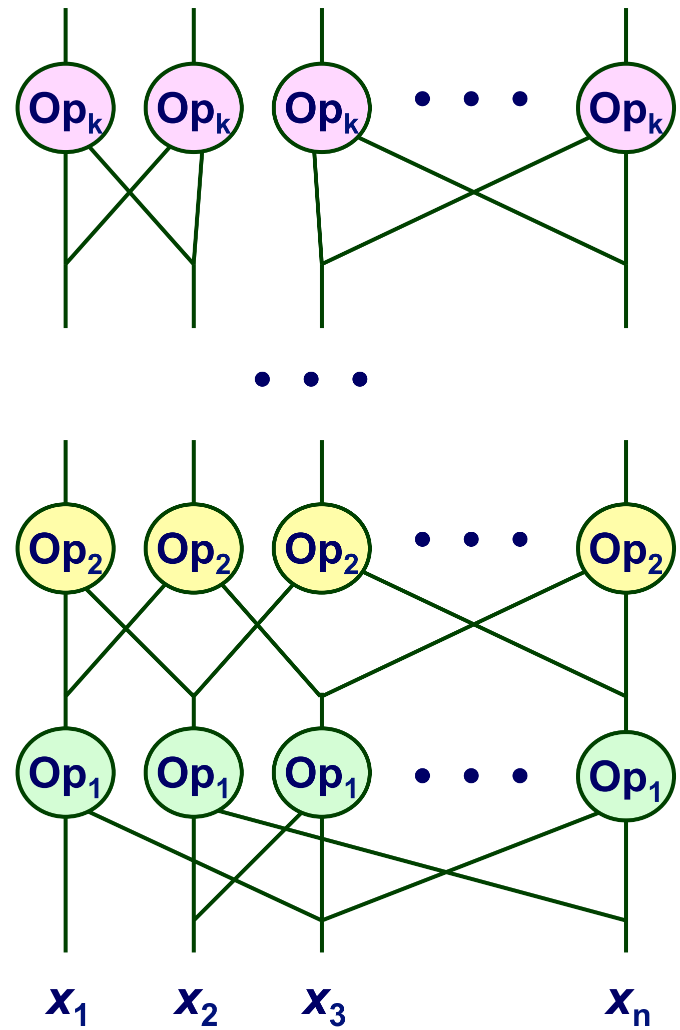
- Microsoft Dryad Project

Computational Model

- Acyclic graph of operators
  - But expressed as textual program
- Each takes collection of objects and produces objects
  - Purely functional model

Implementation Concepts

- Objects stored in files or memory
- Any object may be lost; any operator may fail
- Replicate & recompute for fault tolerance
- Dynamic scheduling
  - # Operators >> # Processors

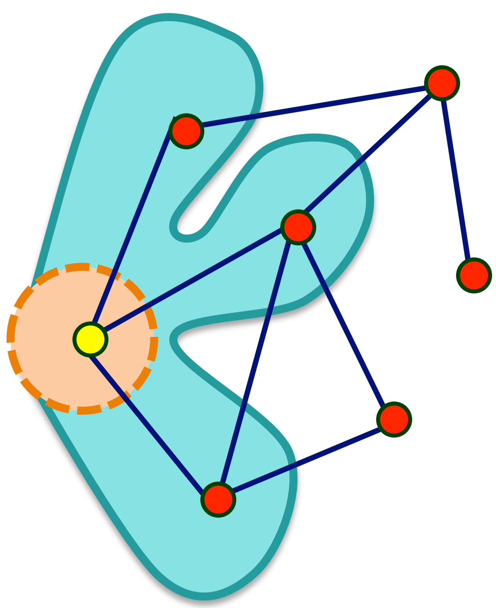


CMU GraphLab

- Carlos Guestrin, et al.
- Graph algorithms used in machine learning

View Computation as Localized Updates on Graph

- New value depends on own value + those of neighbors
- Update repeatedly until converge

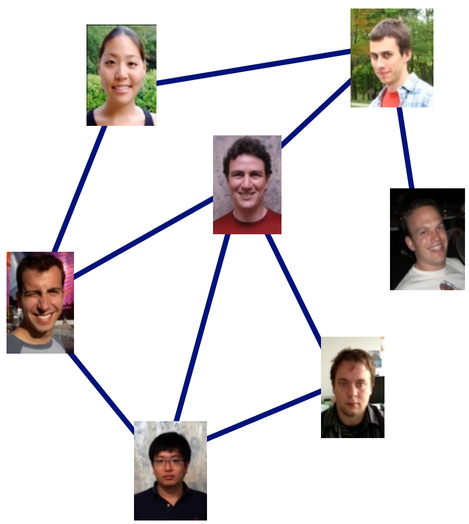


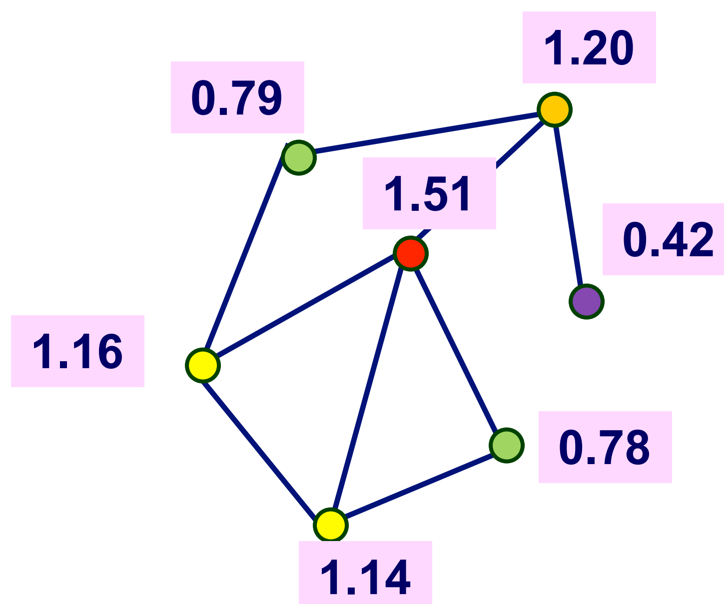
Machine Learning Example

PageRank Computation

- Larry Page & Sergey Brinn, 1998

Rank "Importance" of Web Pages





PageRank Computation

Initially

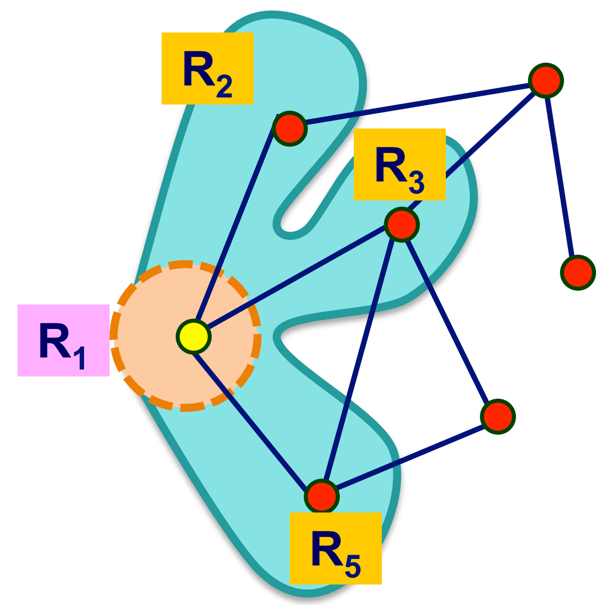
- Assign weight 1.0 to each page

Iteratively

- Select arbitrary node and update its value

Convergence

- Results unique, regardless of selection ordering



$R_1 \leftarrow 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$

PageRank with Map/Reduce

$R_1 \leftarrow 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$

Each Iteration: Update all nodes

- **Map: Generate values to pass along each edge**
  - Key value 1: $(1, \frac{1}{2} R_2)$ $(1, \frac{1}{4} R_3)$ $(1, \frac{1}{3} R_5)$
  - Similar for all other keys
- **Reduce: Combine edge values to get new rank**
  - $R_1 \leftarrow 0.1 + 0.9 * (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5)$
  - Similar for all other nodes

Performance

- **Very slow!**
- **Altavista Webgraph 2002**
  - 1.4B vertices, 6.7B edges

Hadoop	**800 cores**	**9000s**

PageRank with GraphLab

Operation

- **Graph partitioned across multiple processors**
  - Each doing updates to its portion of graph
  - Exploits locality
  - Greater asynchrony
  - Only iterate over portions of graph where values are changing

Performance

- **Altavista Webgraph 2002**
  - 1.4B vertices, 6.7B edges

Hadoop	**800 cores**	**9000s**
Prototype GraphLab2	**512 cores**	**431s**

Conclusions

Distributed Systems Concepts Lead to Scalable Machines

- **Loosely coupled execution model**
- **Lowers cost of procurement & operation**

Map/Reduce Gaining Widespread Use

- **Hadoop makes it widely available**
- **Great for some applications, good enough for many others**

Lots of Work to be Done

- **Richer set of programming models and implementations**
- **Expanding range of applicability**
  - Problems that are data *and* compute intensive
  - The future of supercomputing?