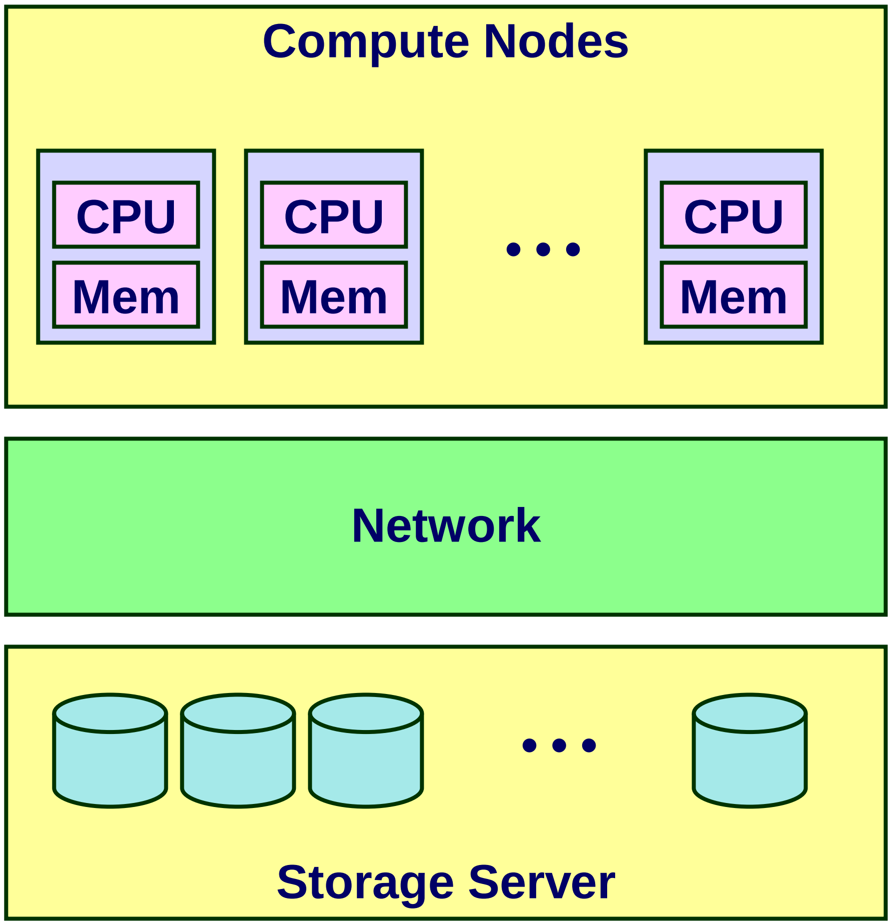
MapReduce Programming
Oct 25, 2011

Topics

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

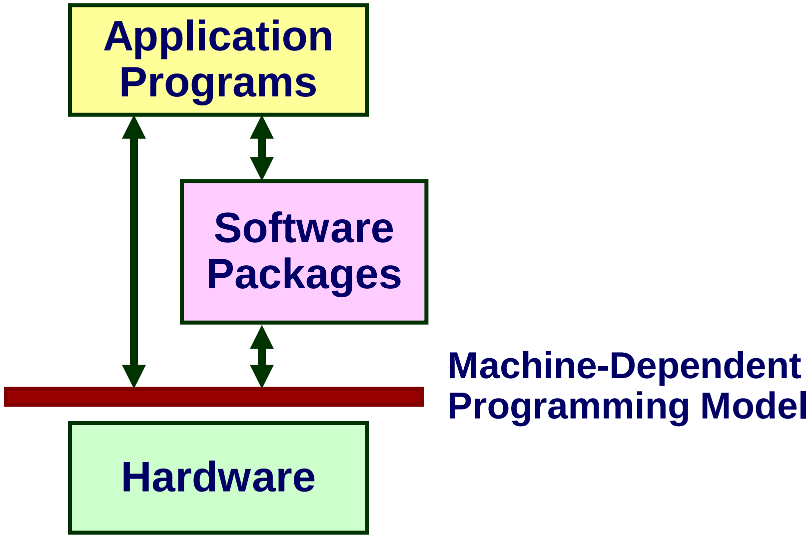
Network

- 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

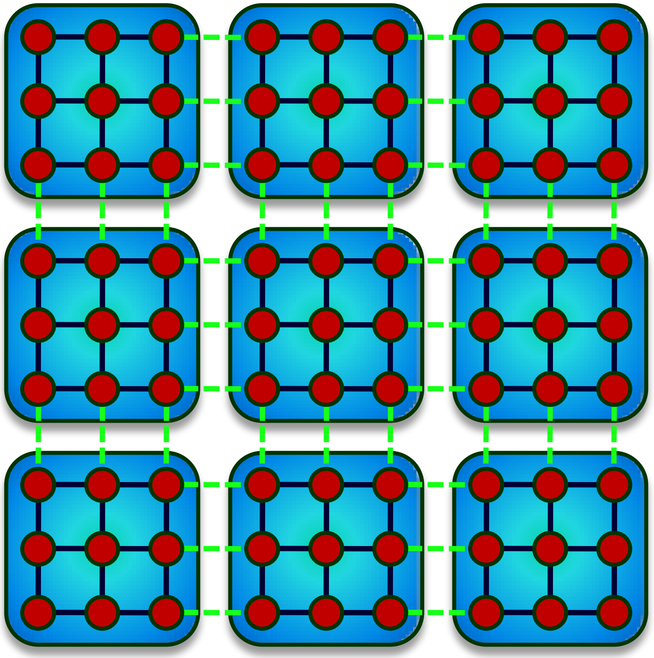
- E.g., finite-element computation

Partition into Regions

- p regions for p processors

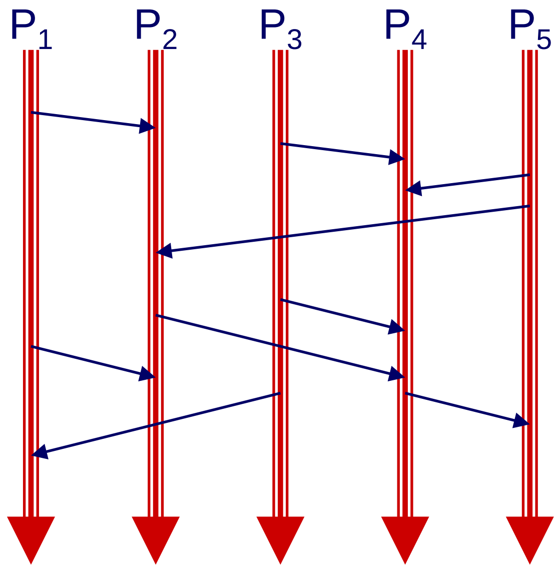
Map Region per Processor

- 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

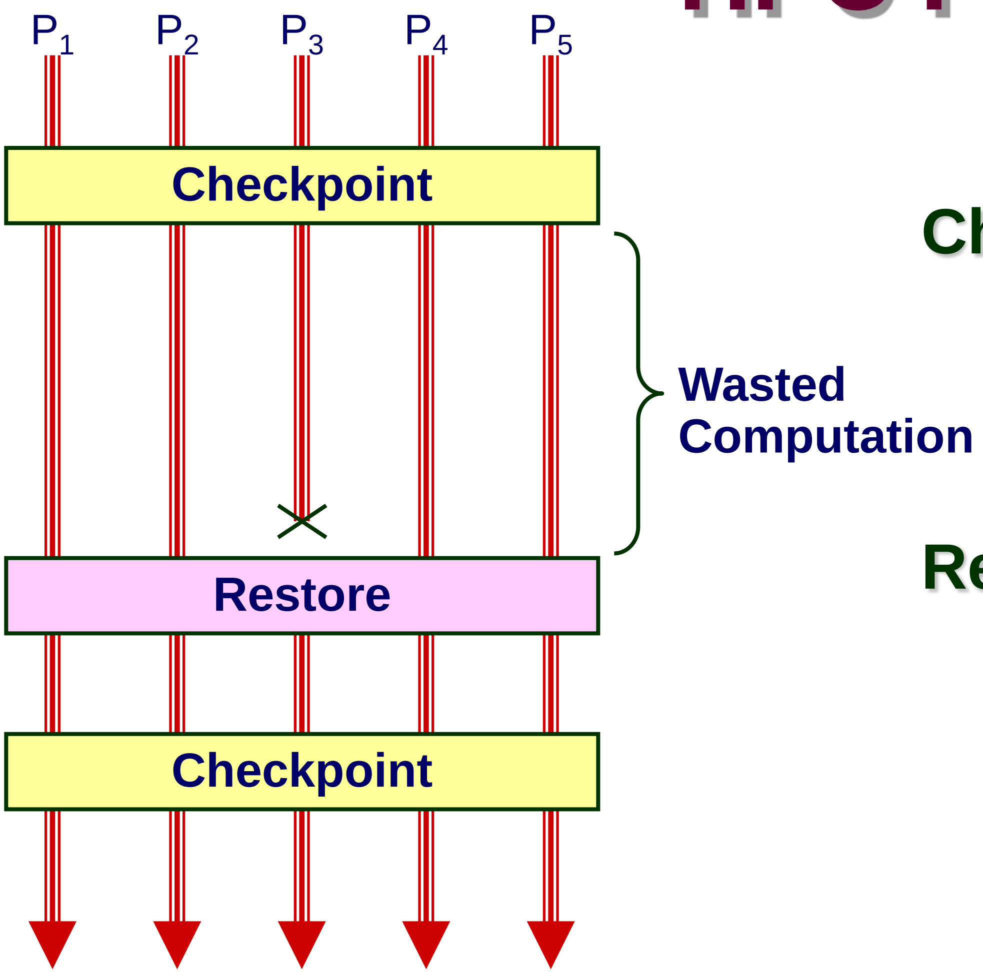
Strengths

- 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

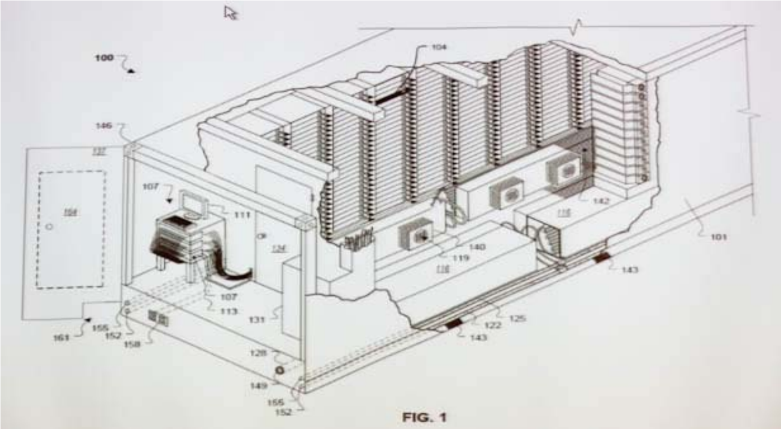
Restore

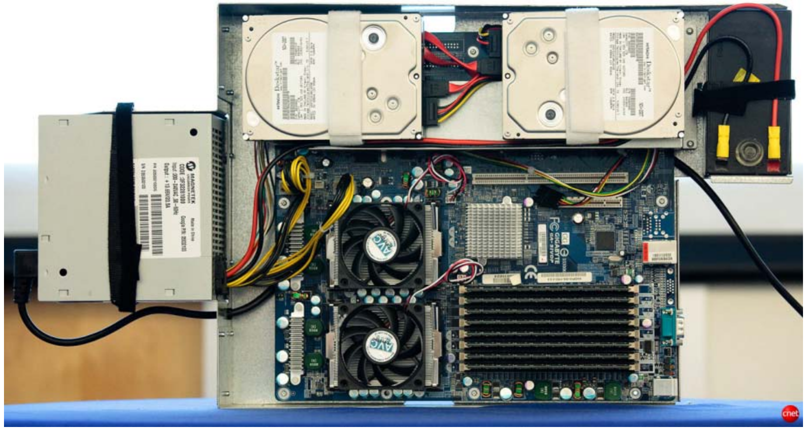
- When failure occurs
- Reset state to that of last checkpoint
- All intervening computation wasted

Performance Scaling

- Very sensitive to number of failing components

Google Data Centers



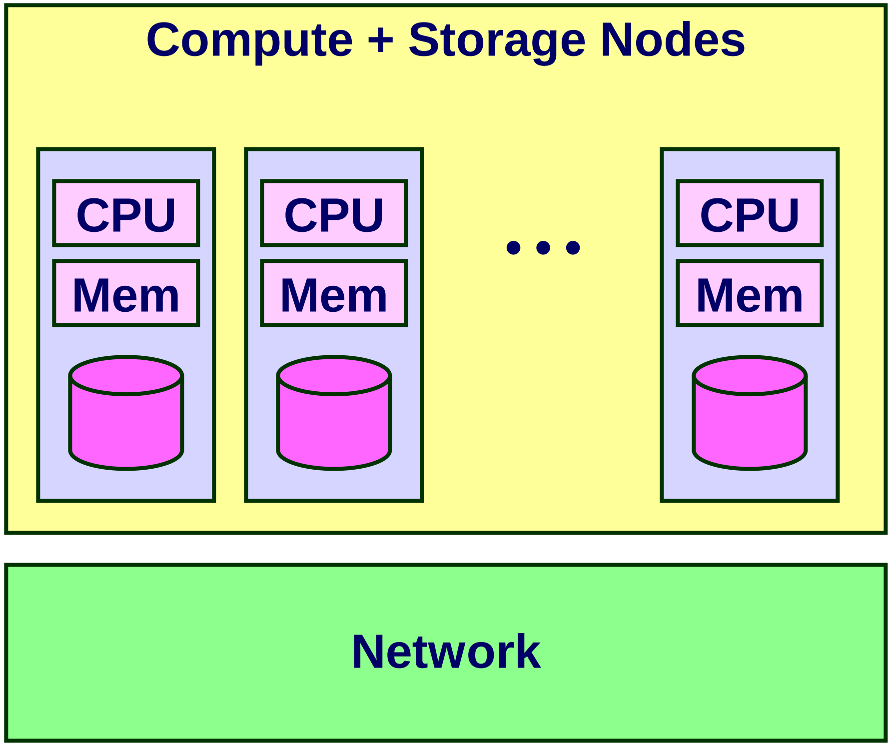


Dalles, Oregon

- **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**
- **2 TB @ $99**
  - 5¢ / GB
  - (vs. 40¢ in 2007)

Seagate Barracuda LP 2 TB 5900RPM SATA 3 GB/s 32 MB Cache 3.5-Inch Internal Hard Drive ST32000542AS-Bare Drive

by Seagate

(123 customer reviews) Like (23) |

List Price: ~~$202.99~~

Price: $99.45

You Save: $103.54 (51%)

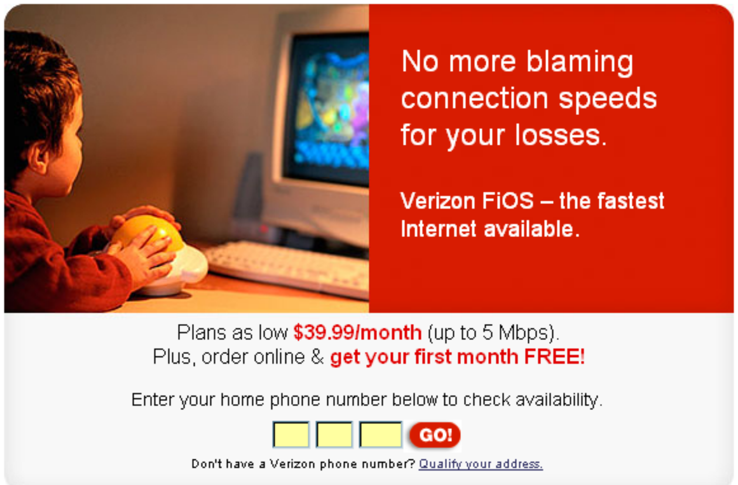
32 new from $74.99 1 refurbished from $99.00

Drawbacks

- **Long and highly variable delays**
- **Not very reliable**

Not included in HPC Nodes

Oceans of Data, Skinny Pipes

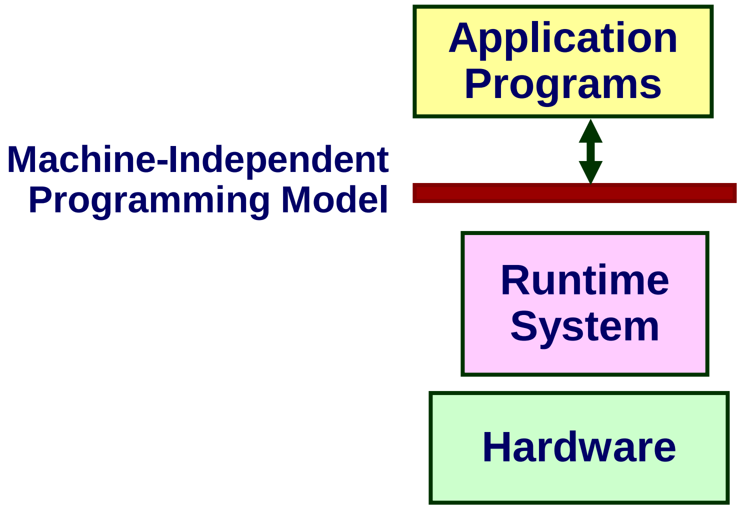


1 Terabyte

- Easy to store
- Hard to move

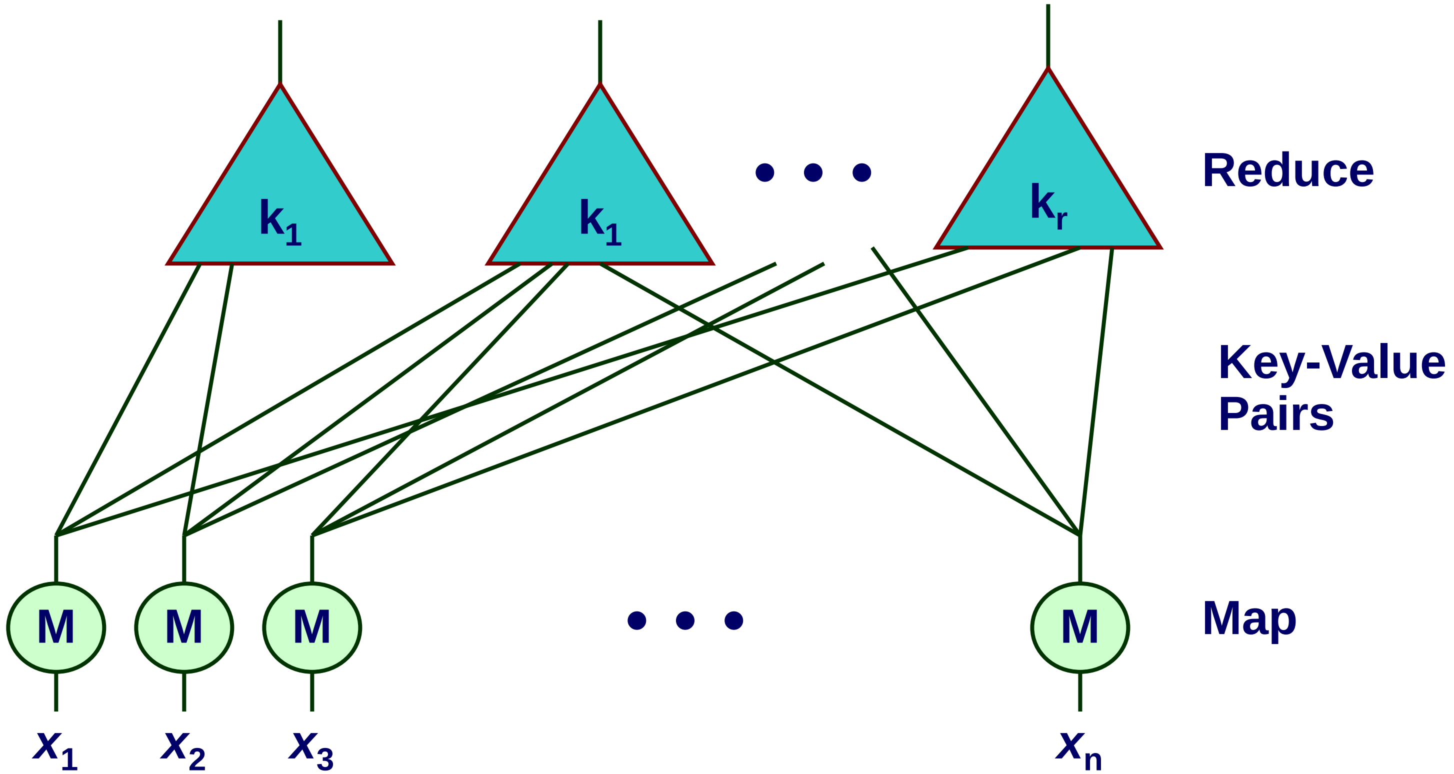
Disks	MB / s	Time
Seagate Barracuda	115	2.3 hours
Seagate Cheetah	125	2.2 hours
Networks	**MB / s**	**Time**
Home Internet	< 0.625	> 18.5 days
Gigabit Ethernet	< 125	> 2.2 hours
PSC Teragrid Connection	< 3,750	> 4.4 minutes

Ideal Cluster Programming Model



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

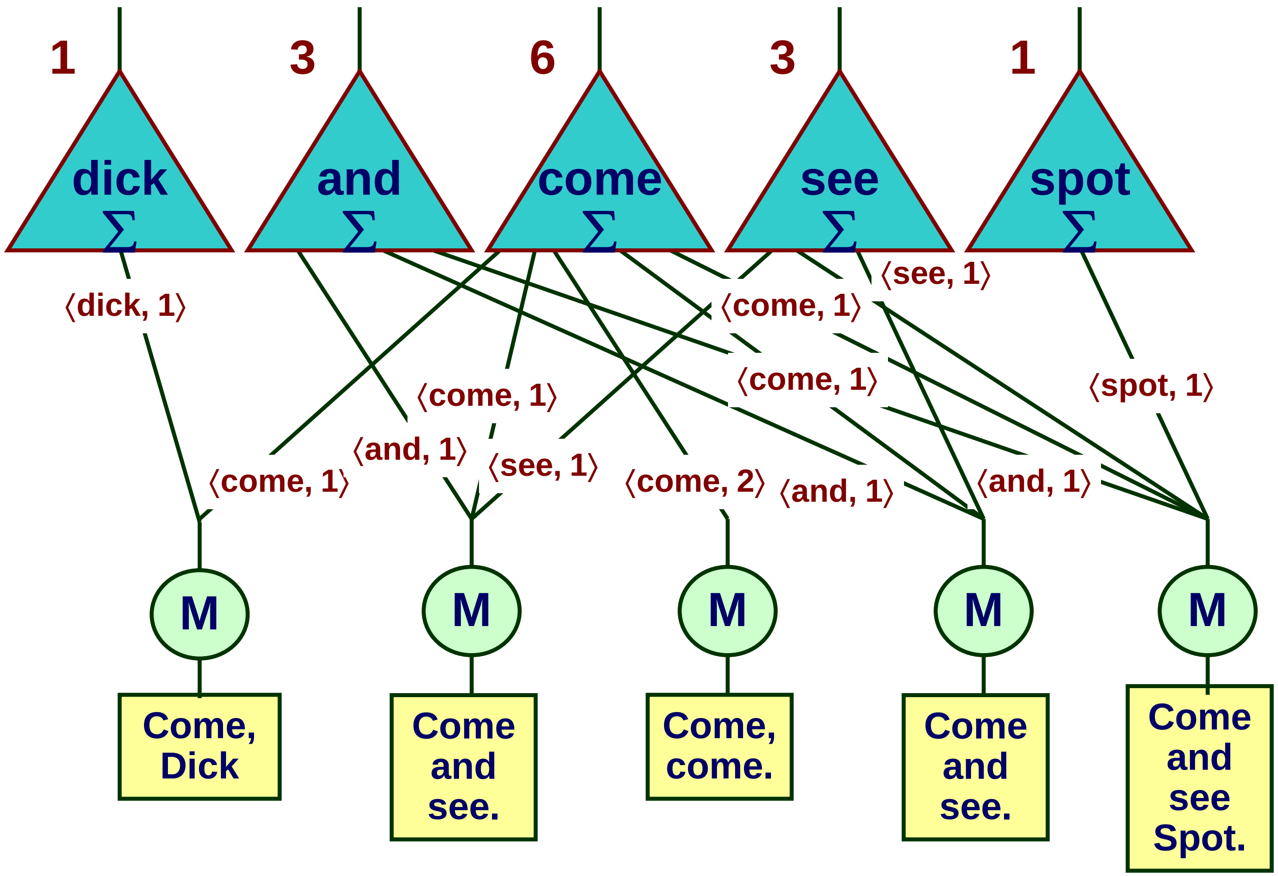
Map/Reduce Programming Model



- **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

Map/Reduce Example



Extract

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

Getting Started

Goal

- **Provide access to MapReduce framework**

Software

- **Hadoop Project**
  - Open source project providing file system and Map/Reduce
  - Supported and used by Yahoo
  - Rapidly expanding user/developer base
  - Prototype on single machine, map onto cluster

Hadoop API

Requirements

- **Programmer must supply Mapper & Reducer classes**

Mapper

- **Steps through file one line at a time**
- **Code generates sequence of <key, value>**
  - 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);
       }
    }

}
```

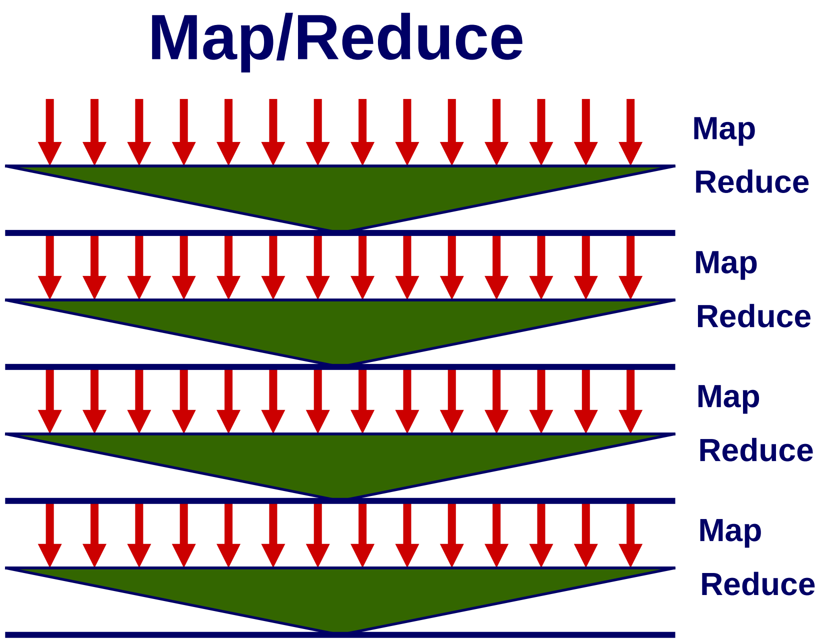
Hadoop Word Count Reducer

```
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));
        }

}
```

Map/Reduce Operation



Characteristics

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

Strengths

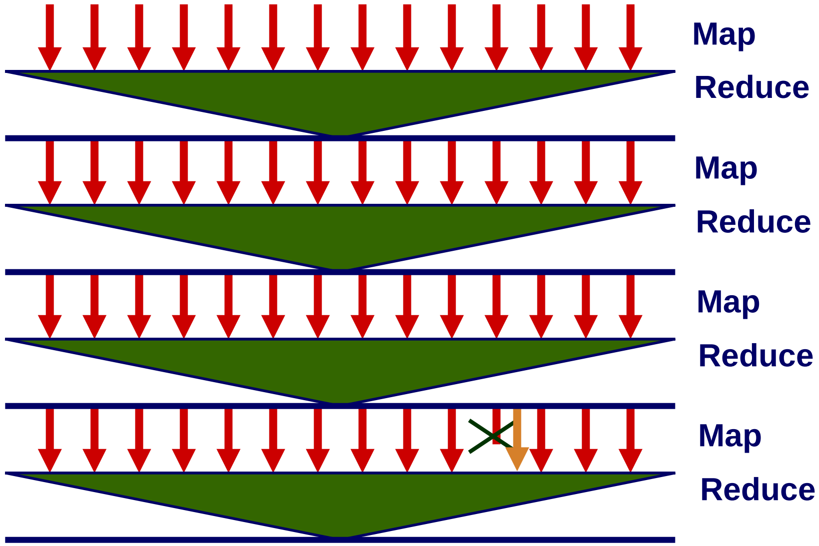
- **Great flexibility in placement, scheduling, and load balancing**
- **Can access large data sets**

Weaknesses

- **Higher overhead**
- **Lower raw performance**

Map/Reduce Fault Tolerance

Map/Reduce



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**

Cluster Scalability Advantages

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

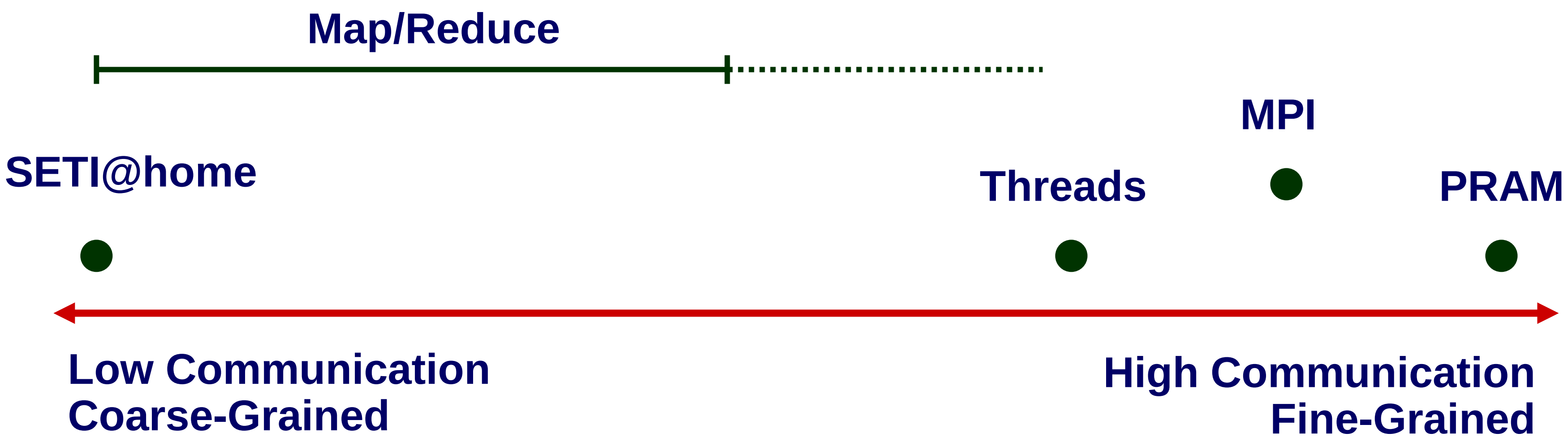
Provisioning Advantages

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

Operational Advantages

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

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

Example: Sparse Matrices with Map/Reduce

A

10		20
	30	40
50	60	70

X

B

-1	
-2	-3
	-4

=

C

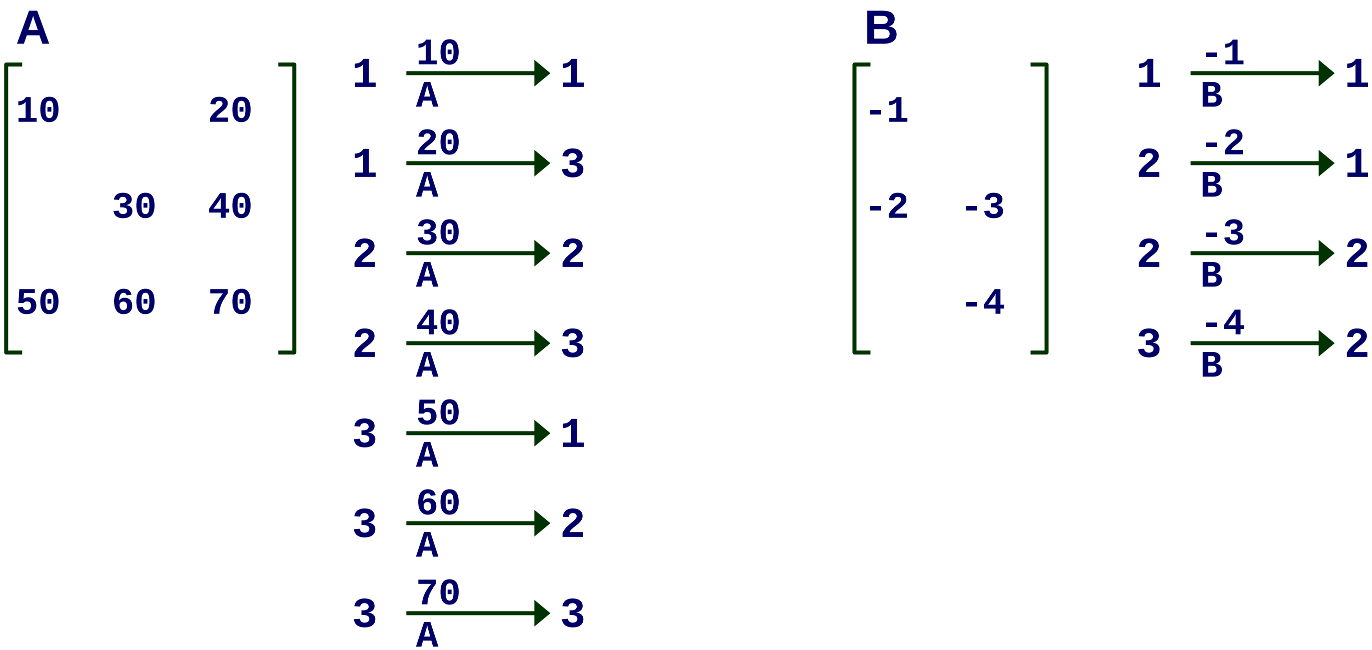
-10	-80
-60	-250
-170	-460

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

Motivation

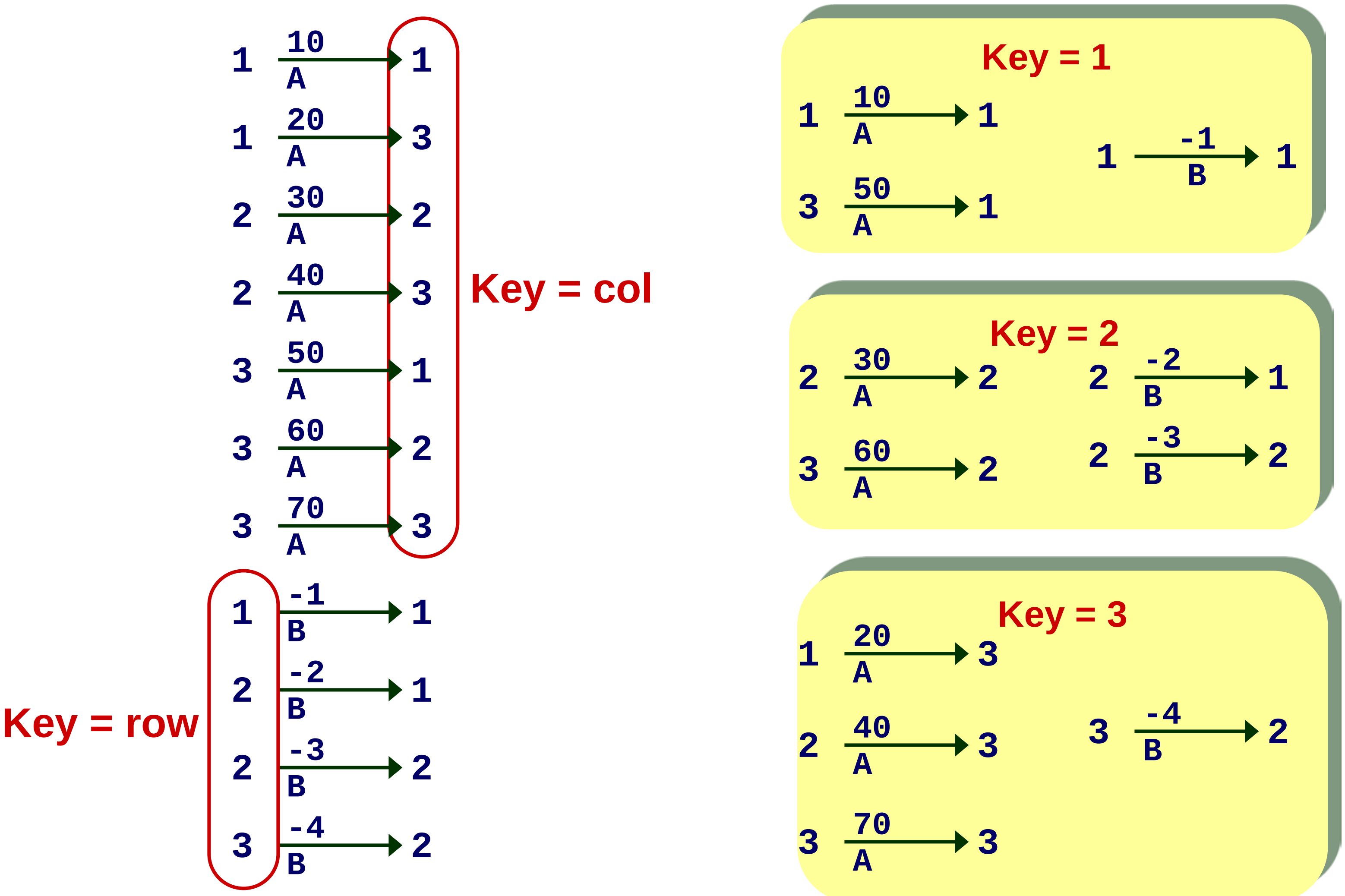
- Core problem in scientific computing
- Challenging for parallel execution
- Demonstrate expressiveness of Map/Reduce

Computing Sparse Matrix Product



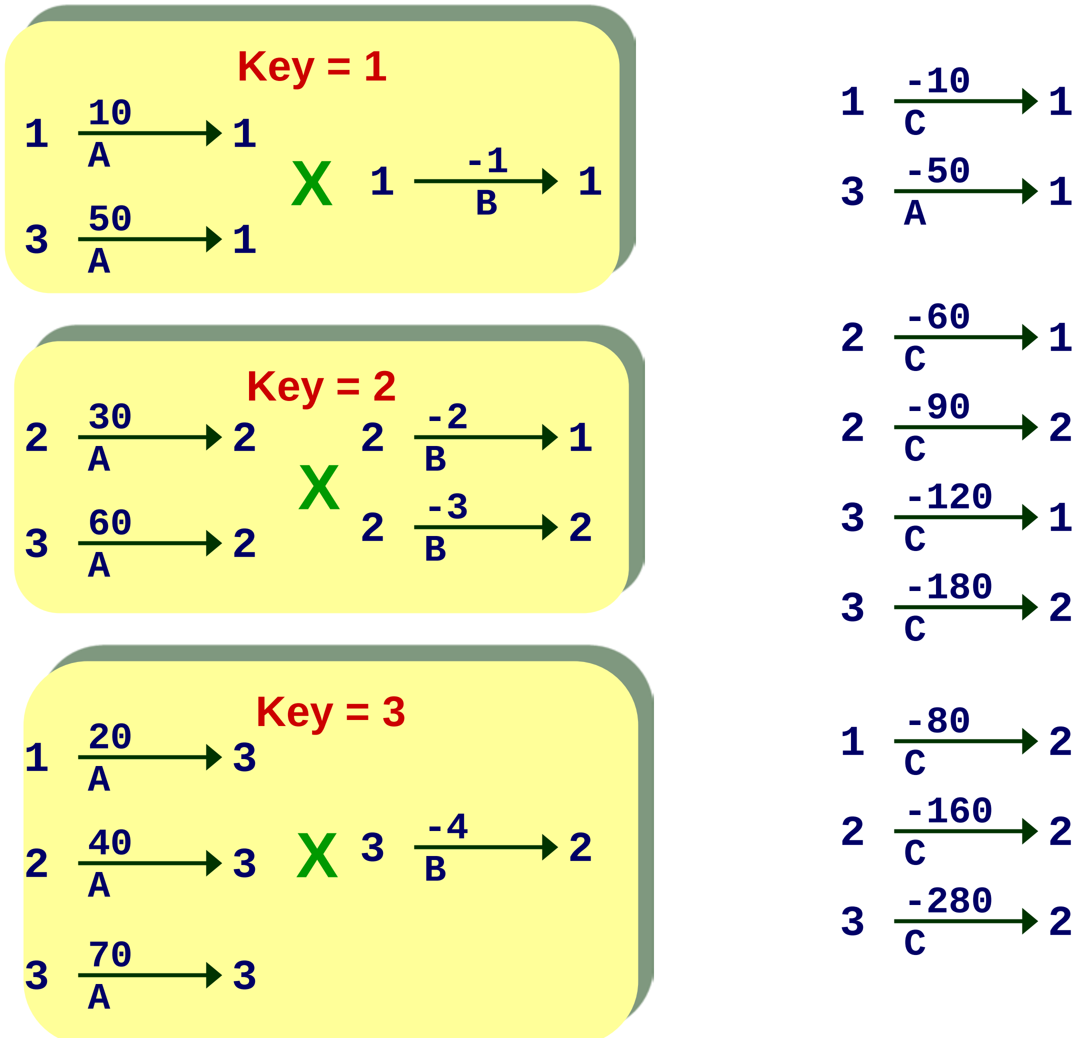
- Represent matrix as list of nonzero entries
  ⟨row, col, value, matrixID⟩
- Strategy
  - Phase 1: Compute all products $a_{i,k} \cdot b_{k,j}$
  - Phase 2: Sum products for each entry i,j
  - Each phase involves a Map/Reduce

Phase 1 Map of Matrix Multiply



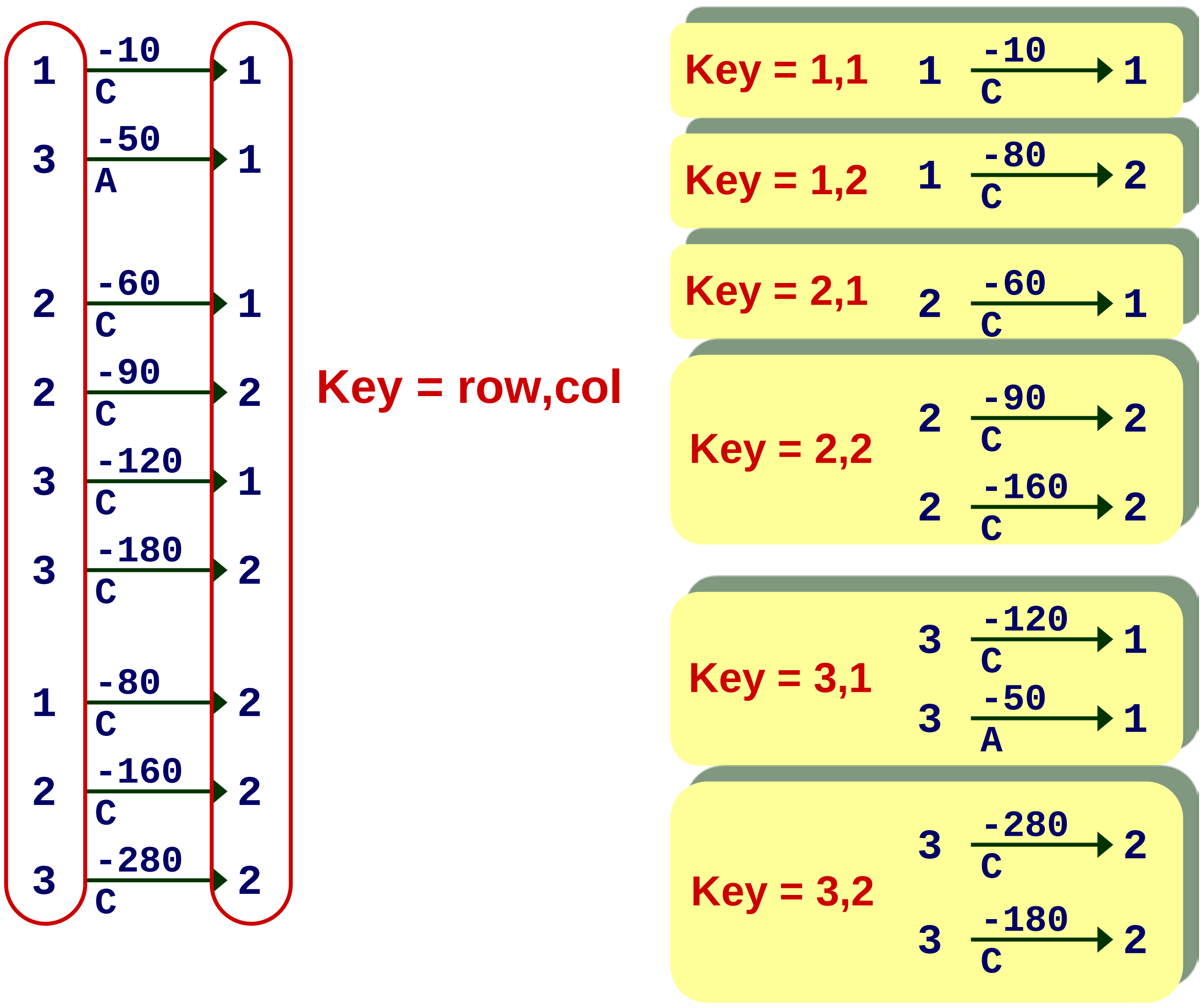
- Group values $a_{i,k}$ and $b_{k,j}$ according to key k

Phase 1 "Reduce" of Matrix Multiply



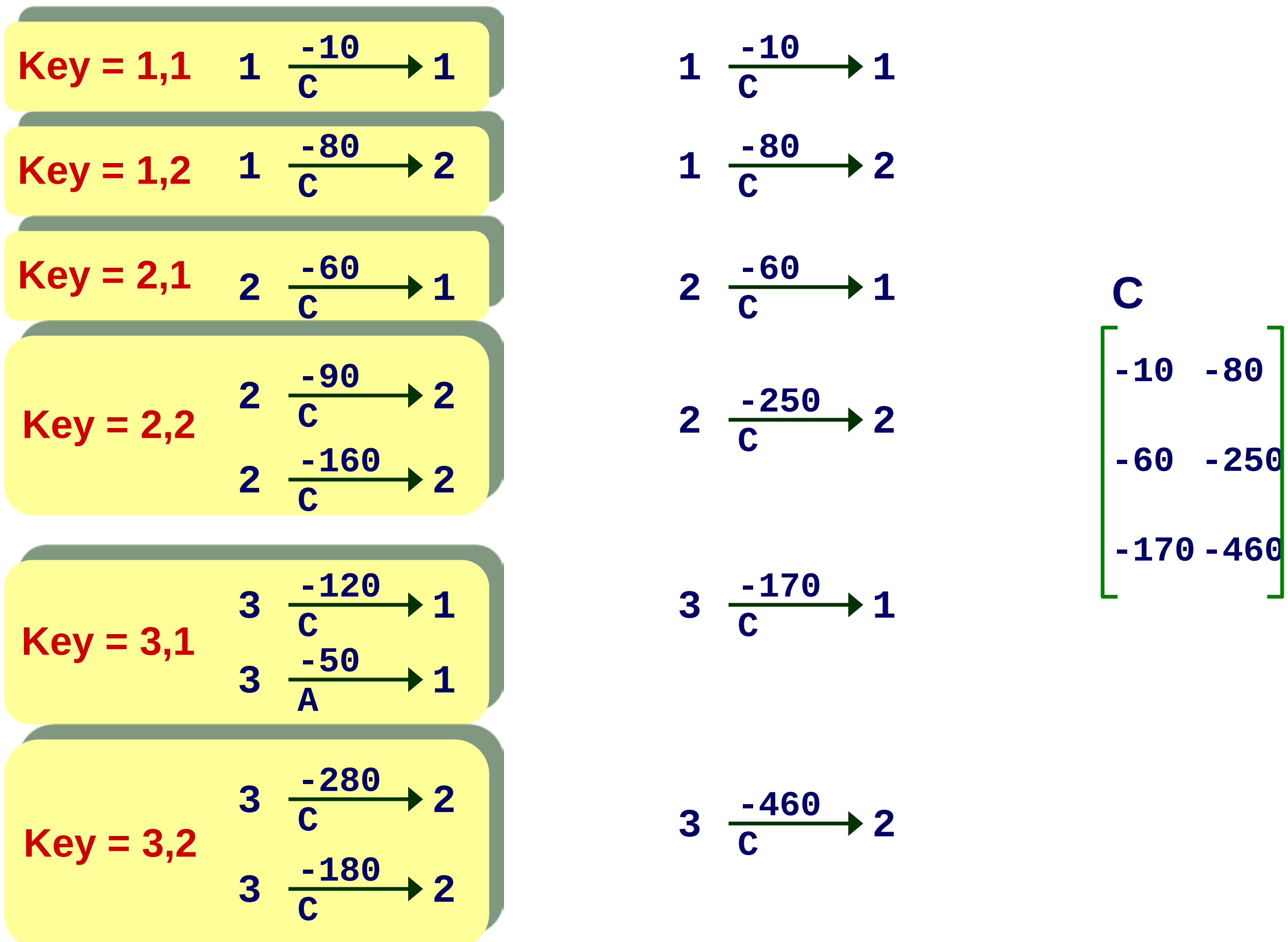
- Generate all products $a_{i,k} \cdot b_{k,j}$

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



- 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
- Can write program as series of MapReduce steps

Mapping

Parameters

- **M: Number of mappers**
  - Each gets ~1/M of the input data
- **R: Number of reducers**
  - Each reducer i gets keys k such that hash(k) = i

Tasks

- **Split input files into M pieces, 16—64 MB each**
- **Scheduler dynamically assigns worker for each "split"**

Task operation

- **Parse "split"**
- **Generate key, value pairs & write R different local disk files**
  - Based on hash of keys
- **Notify master of worker of output file locations**

Reducing

Shuffle

- Each reducer fetches its share of key, value pairs from each mapper using RPC
- Sort data according to keys
  - Use disk-based ("external") sort if too much data for memory

Reduce Operation

- Step through key-value pairs in sorted order
- For each unique key, call reduce function for all values
- Append result to output file

Result

- R output files
- Typically supply to next round of MapReduce

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
- **Rescheduled 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

Generalizing Map/Reduce

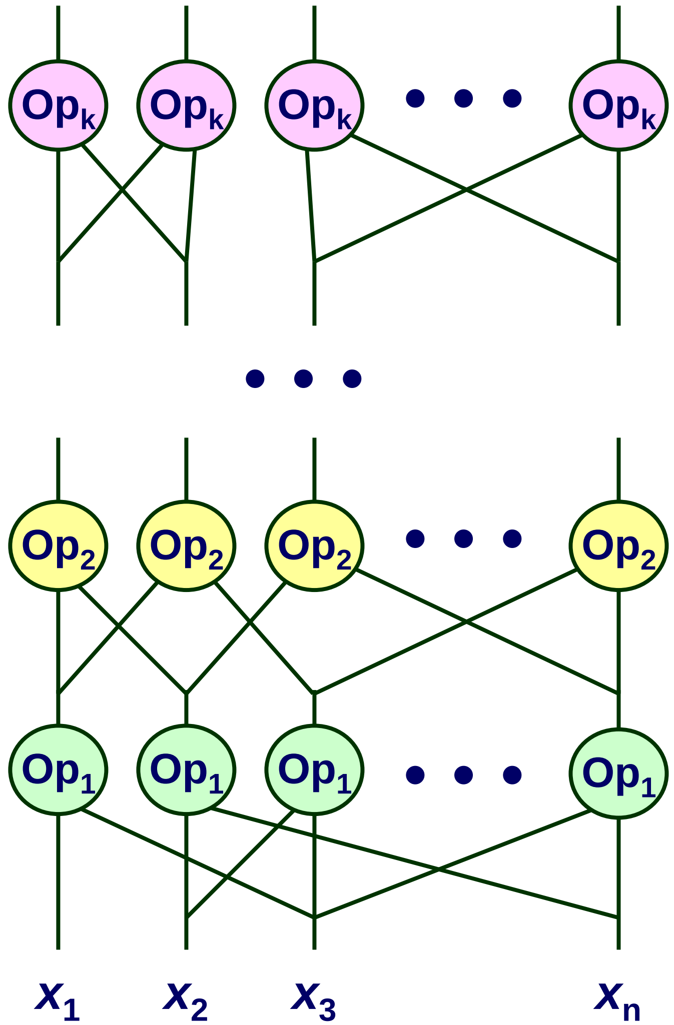
- Microsoft Dryad Project

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



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?