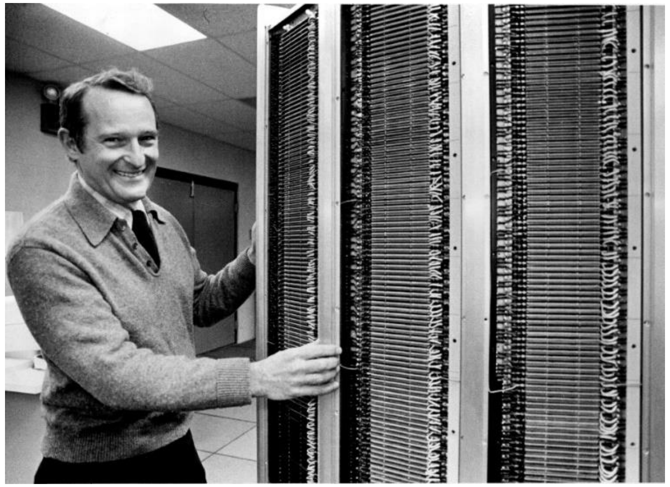
Shared-Nothing Parallelism & Distributed Programming

Lecture 4

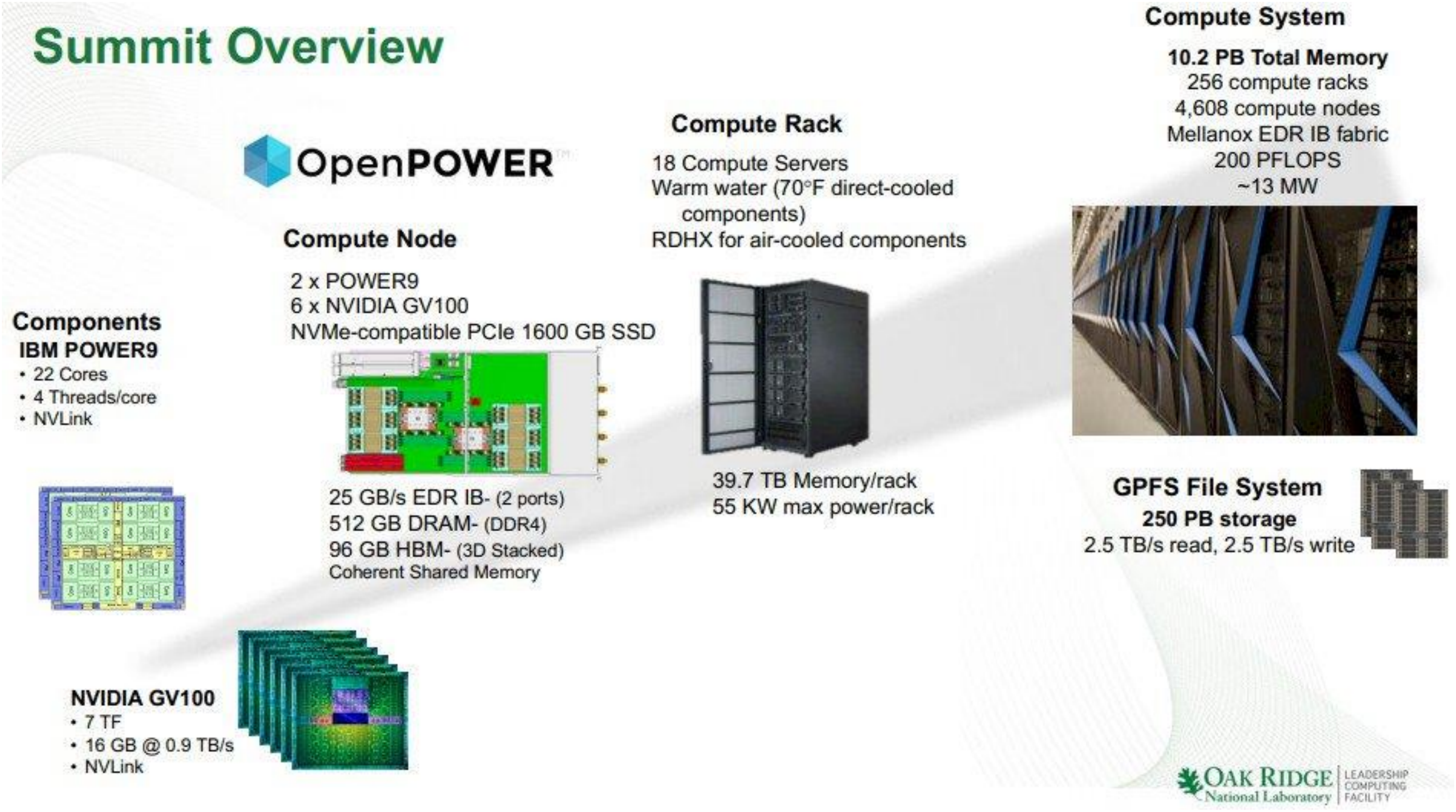
Single Node to Cluster

- Supercomputers the pinnacle of computation
  - Solve important science problems, e.g.,
    - Airplane simulations
    - Weather prediction
- Large national racing for most powerful computers
- In quest for increasing power, supercomputers are made of distributed/parallel computers
  - 1000s of processors
  - High-bandwidth low latency networking and storage



Seymour Cray, Cray-1

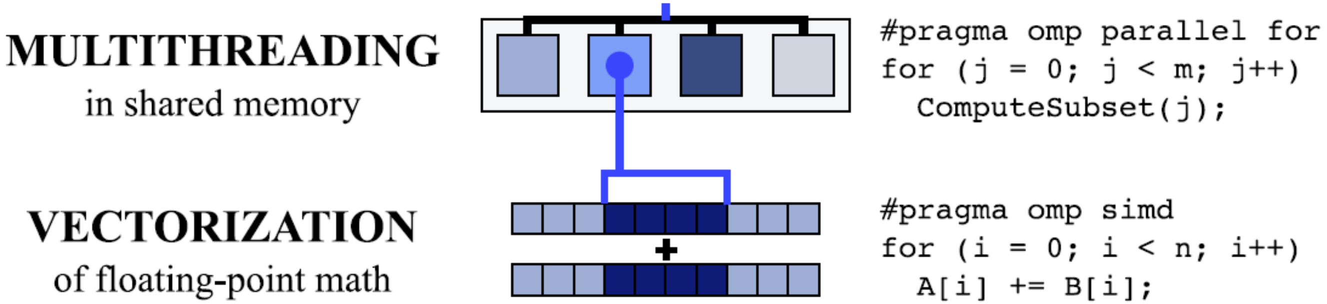
Summit

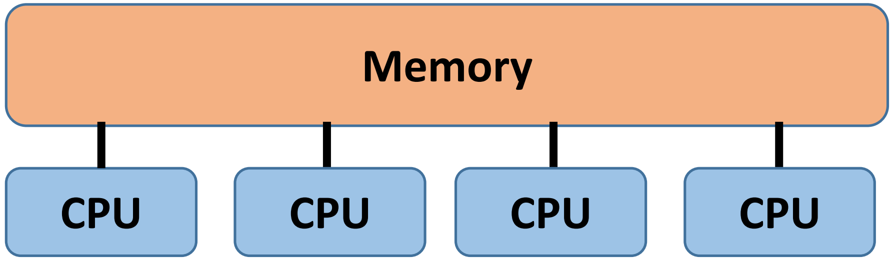

Summit Overview
OpenPOWER
Components
IBM POWER9
• 22 Cores
• 4 Threads/core
• NVLink
NVIDIA GV100
• 7 TF
• 16 GB @ 0.9 TB/s
• NVLink
Compute Node
2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD
25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory
Compute Rack
18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components
39.7 TB Memory/rack
55 KW max power/rack
Compute System
10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW
GPFS File System
250 PB storage
2.5 TB/s read, 2.5 TB/s write
OAK RIDGE National Laboratory
LEADERSHIP COMPUTING FACILITY

How do we program supercomputers?

<u>Intra-Node: Shared memory parallelism</u>

- Within a single node
  - Hardware supports *cache coherent shared memory*
  - *Cache coherent*: store made by one cpu is visible to load by another cpu
  - *Shared memory*: any cpu can access any memory location

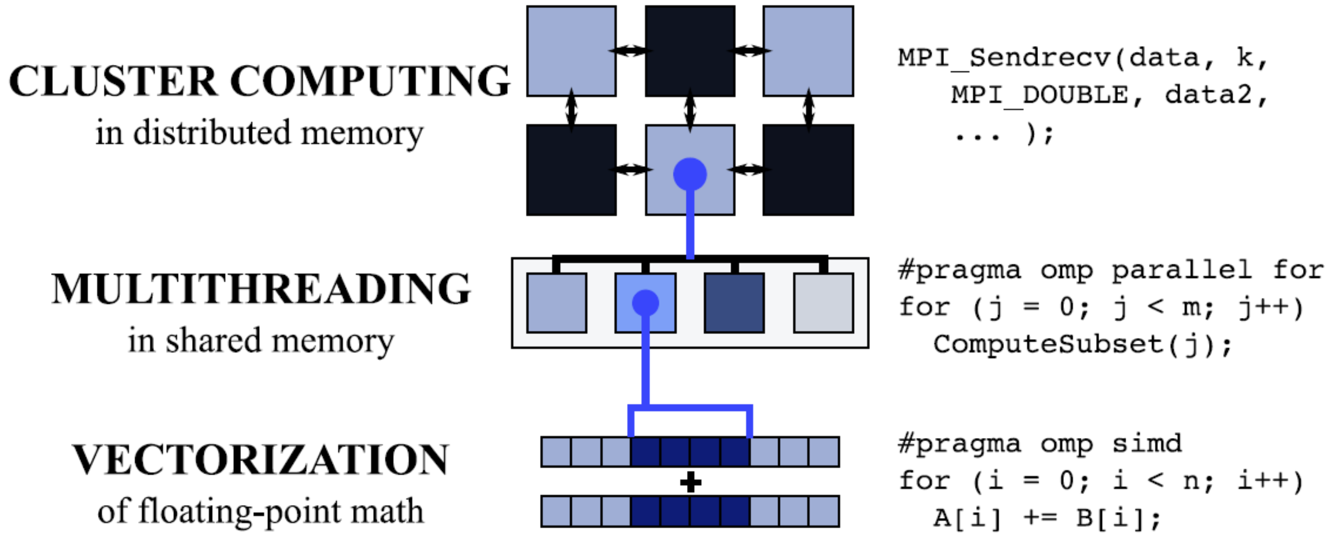


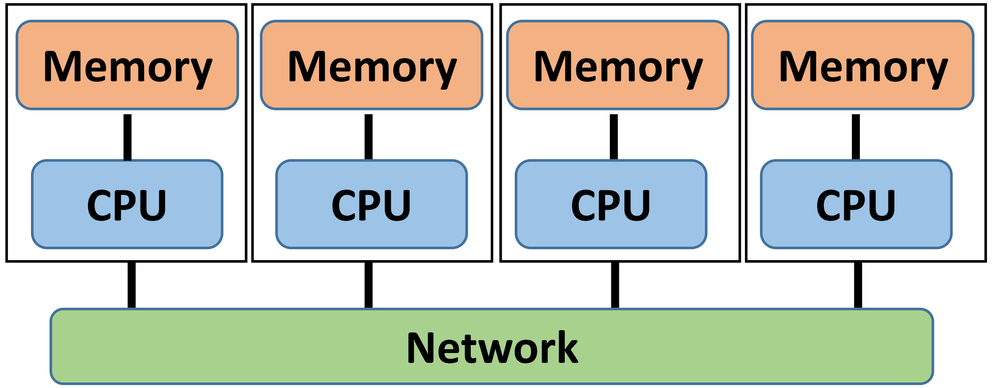


How do we program supercomputers?

Inter-node: Message Passing Parallelism

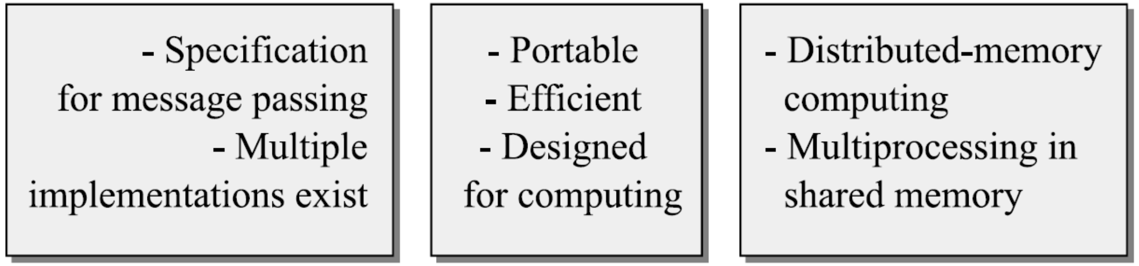
- No cache coherence, no shared memory
  - Need to explicitly communicate across machines by sending and receiving messages





Message Passing Interface

- Message Passing Interface (MPI)
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing
  - Portable: one standard, many implementations
  - De facto standard platform for the High Performance Computing (HPC) community



MPI Routines

- Many parallel programs can be written using just these six functions, only two of which are non-trivial

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of calling process.
MPI_Send	Sends a “unbuffered/blocking” message.
MPI_Recv	Receives a “unbuffered/blocking message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.

- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.

- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)

int MPI_Finalize()
```

Skeleton MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        MPI_Init(&argc, &argv);
        printf("Hello, world!\n");
        MPI_Finalize();
        return 0;
}
```

MPI Communicators

- A communicator defines a *communication domain*
  - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Communicator Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Example MPI Program

```
#include<stdio.h>
#include “mpi.h”

main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);

        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

        printf("From process %d out of %d, Hello
        World!\n", myrank, npes);

        MPI_Finalize();
}
```

Messaging MPI

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,int dest,
      int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- MPI_Datatype could be
  - MPI_CHAR (signed char)
  - MPI_SHORT (signed short int)
  - MPI_INT (singed int)
  - …

Communication Example

Consider the following piece of code, in which process i sends a message to process *i* + 1 (modulo the number of processes) and receives a message from process *i* - 1 (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
```

What happens when MPI_Send is blocking? ***DEADLOCK***

- Blocking, means the program will not continue until the communication is completed (Synchronous communication)

Avoiding Deadlock

- Break circular wait

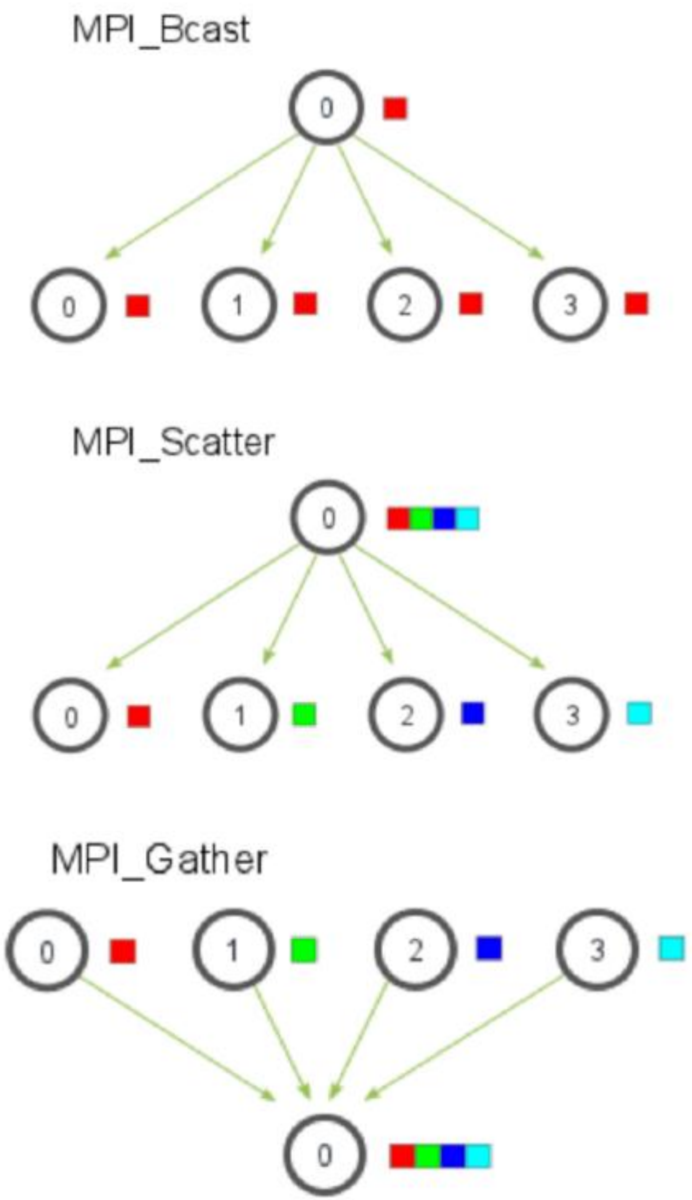
```
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
```

- Exchange (Send and receive) in one shot

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

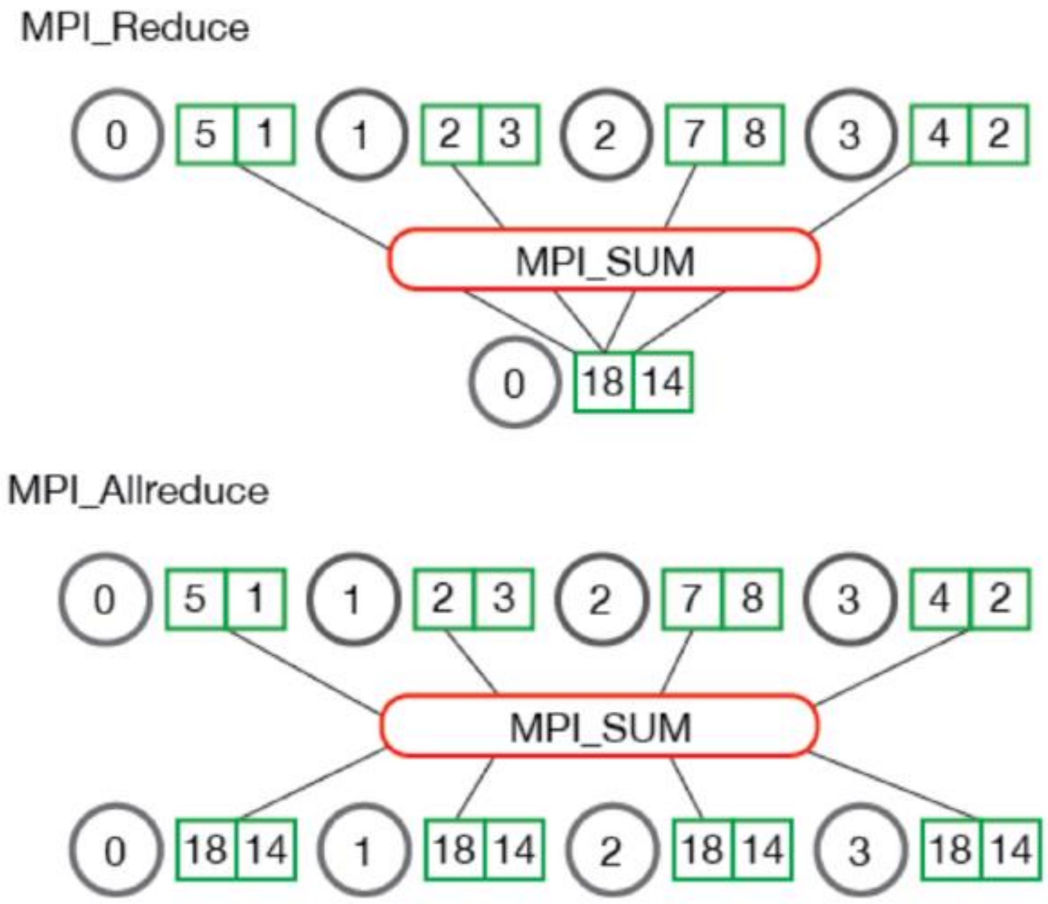
Many other functions

- MPI_Bcast
  - Broadcast same data to all processes in a group
- MPI_Scatter
  - send different pieces of an array to different processes
  - (i.e., partition an array across processes)
- MPI_Gather
  - take elements from many processes and gathers them to one single process



Many other functions

- MPI_Reduce
  - Takes an array of input elements on each process and returns an array of output elements to the root process given a specified operation
- MPI_Allreduce
  - Like MPI_Reduce but distribute results to all processes



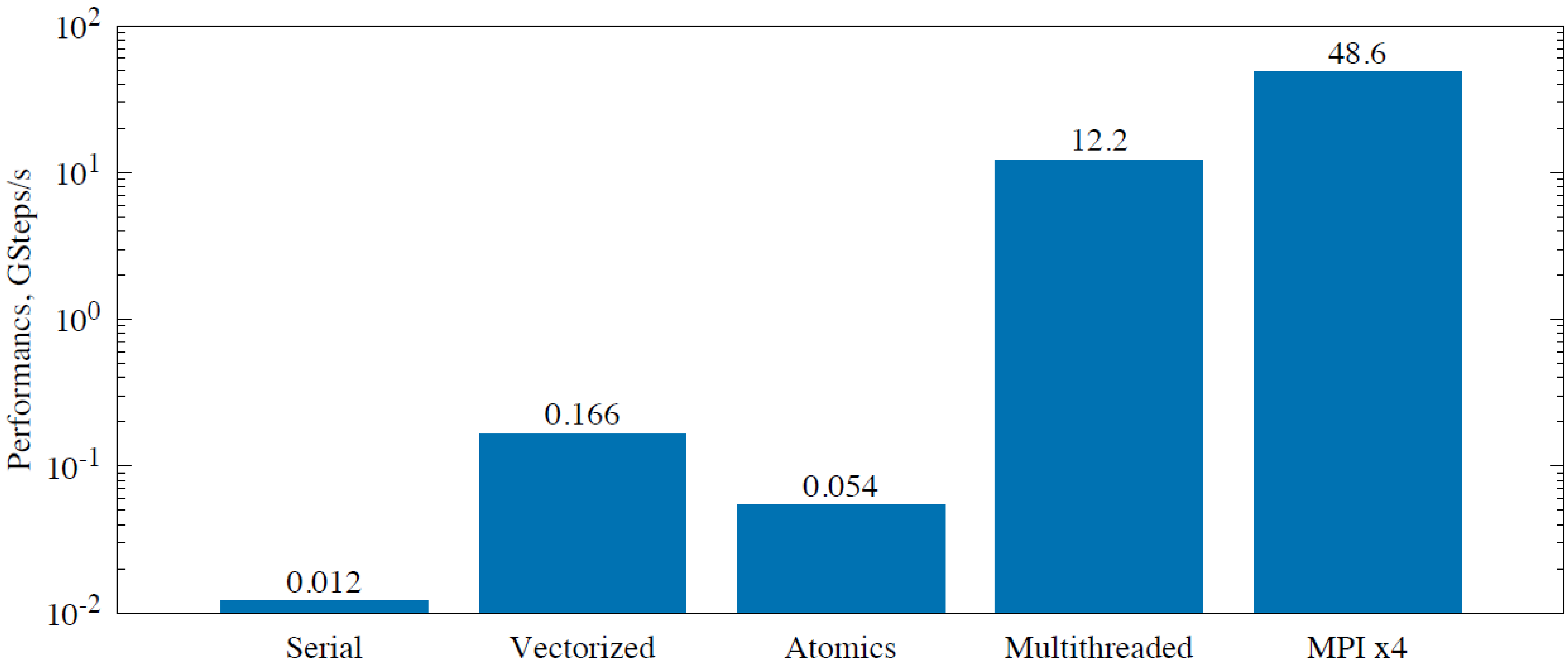
Numerical Integration (Recap): Single Node

```
const double dx = a/(double)n;
double integral = 0.0;
#pragma omp parallel for simd reduction(+: integral)
for (int i = 0; i < n; i++) {
  const double xip12 = dx*((double)i + 0.5);
  const double dI = BlackBoxFunction(xip12)*dx;

  integral += dI;
}
```

Numerical Integration with MPI

```
#include "library.h"
#include <mpi.h>

double ComputeIntegral(const int n, const double a, const double b, const int rank, const int nRanks)\
 {

  const int stepsPerProcess = double(n-1)/double(nRanks);
  const int iStart = int( stepsPerProcess*rank );
  const int iEnd = int( stepsPerProcess*(rank + 1) );

  const double dx = (b - a)/n;
  double I_partial = 0.0;

#pragma omp parallel for simd reduction(+: I_partial)
  for (int i = iStart; i < iEnd; i++) {

    const double xip12 = a + dx*(double(i) + 0.5);
    const double yip12 = BlackBoxFunction(xip12);
    const double dI = yip12*dx;
    I_partial += dI;

  }

  double I = 0.0;
  MPI_Allreduce(&I_partial, &I, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

MPI Benefit


Performancs, GSteps/s
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
0.012
0.166
0.054
12.2
48.6
Serial
Vectorized
Atomics
Multithreaded
MPI x4

MPI and High-performance computing

- Typically HPC application
  - Consists of several long-lived processes
  - Hold all program data in memory (no disk access)
  - High bandwidth communication
- MPI
  - Exposes number of processes
  - Communication is explicit, driven by the program
- Strengths
  - High utilization of resources
  - Effective for many scientific applications
- **Weaknesses**
  - **Requires careful tuning of application to resources**
  - **Intolerant of any variability**
  - **Dealing with failures is hard**

Enter 1990s

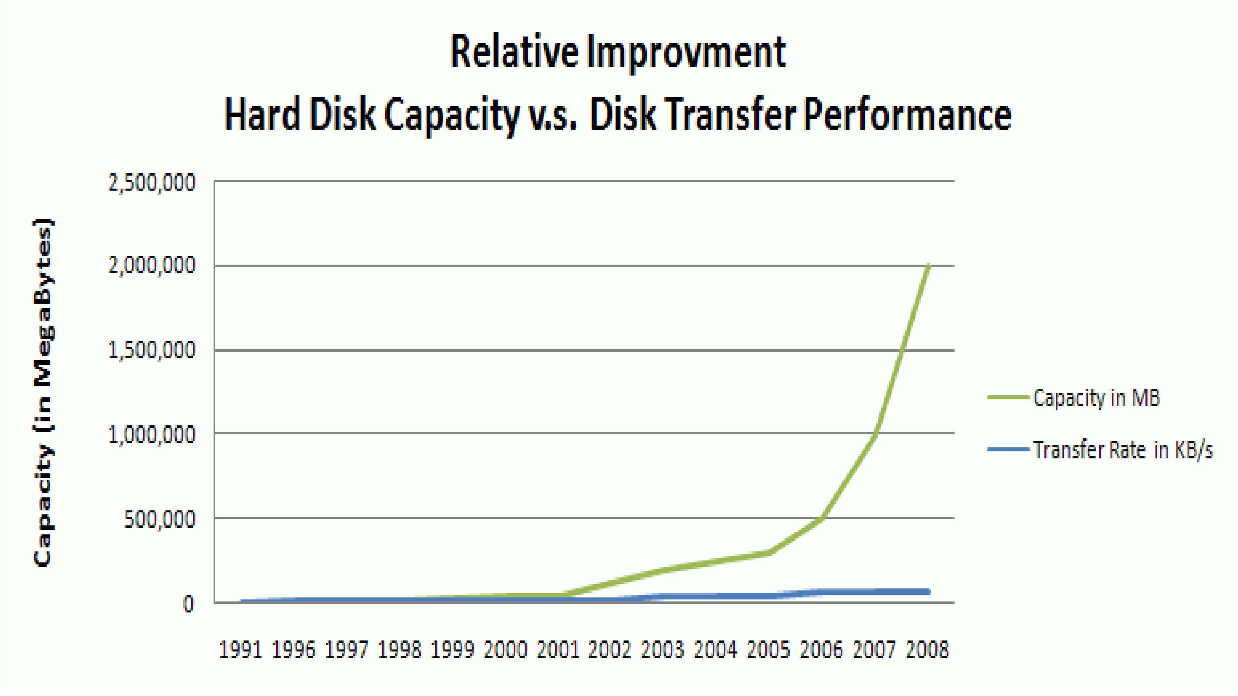
Larry Page, Sergey Brin

- Internet and World Wide Web taking off
- Search as a killer application
  - Need to index and process huge amounts of data
  - Data processing: highly parallel
  - Data too large to fit in memory, must be stored in disk
- Supercomputers are designed for computation intensive workloads
  - Search and similar workloads were data intensive
- Supercomputers are very expensive
  - Could build cluster of commodity servers with disks, CPU

Scale out with commodity servers instead of scaling up with a supercomputer

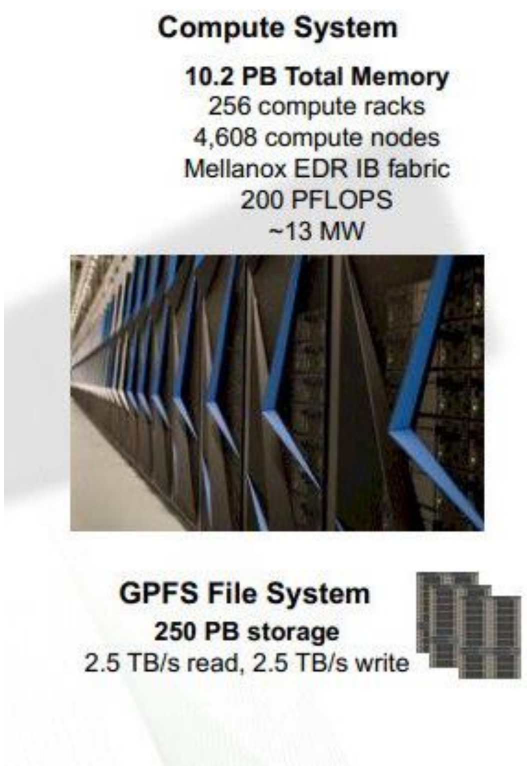
Big data, skinny pipe problem

- Hard disk capacity has been growing rapidly
  - Moore's law: Number of transistors doubles every 2 years
  - Kryder's rate: Hard disk density doubles every 13 months!
- But bandwidth improvements are not keeping pace
  - At 100MB/s, simply reading 100TB data will take ~11 days!



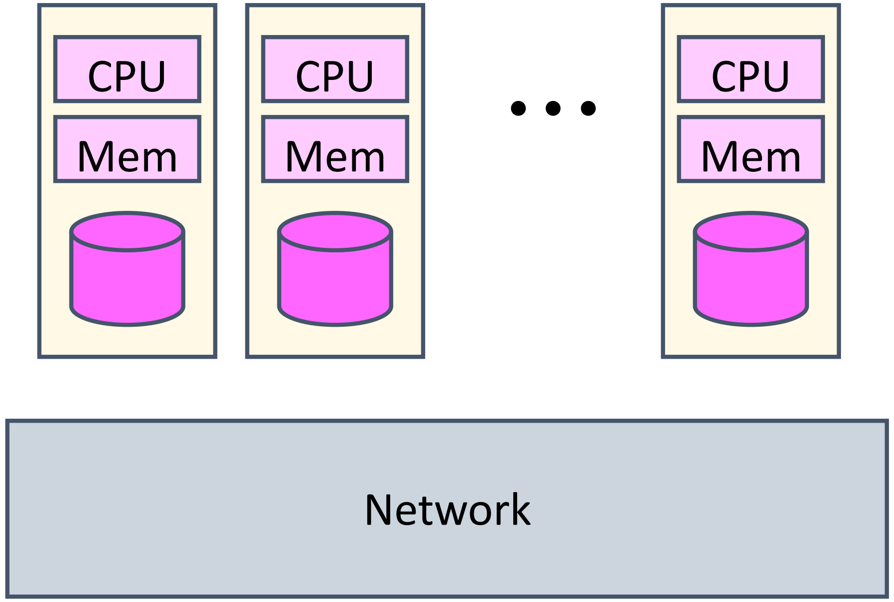
To share storage or not to share

- For Computation That Accesses 100 TB in 1 minutes
  - Data distributed over ~20,000 disks
    - Assuming uniform data partitioning
    - Aggregate bandwidth = 20k * 100 MB/s = ~2TB/s
- Supercomputers use Compute—storage separation
  - Storage servers shared data repository
  - Compute servers pull data across fast network
  - Easy to scale separately
- But in a cluster, network becomes bottleneck
  - 1 Gbit, 10 Gbit Ethernet cheap instead of high-performance network
  - Big data, skinny pipe problem again



Shared nothing architecture

- Collocate disk in each server
- Compute using thousands of processors: **Data locality principle**
  - Each processor processes data on local disk, minimizing network data transfer
  - Move processing to data instead of vice versa
- Use distributed file systems to manage data
  - Disk local to each server, but need shared global directory hierarchy

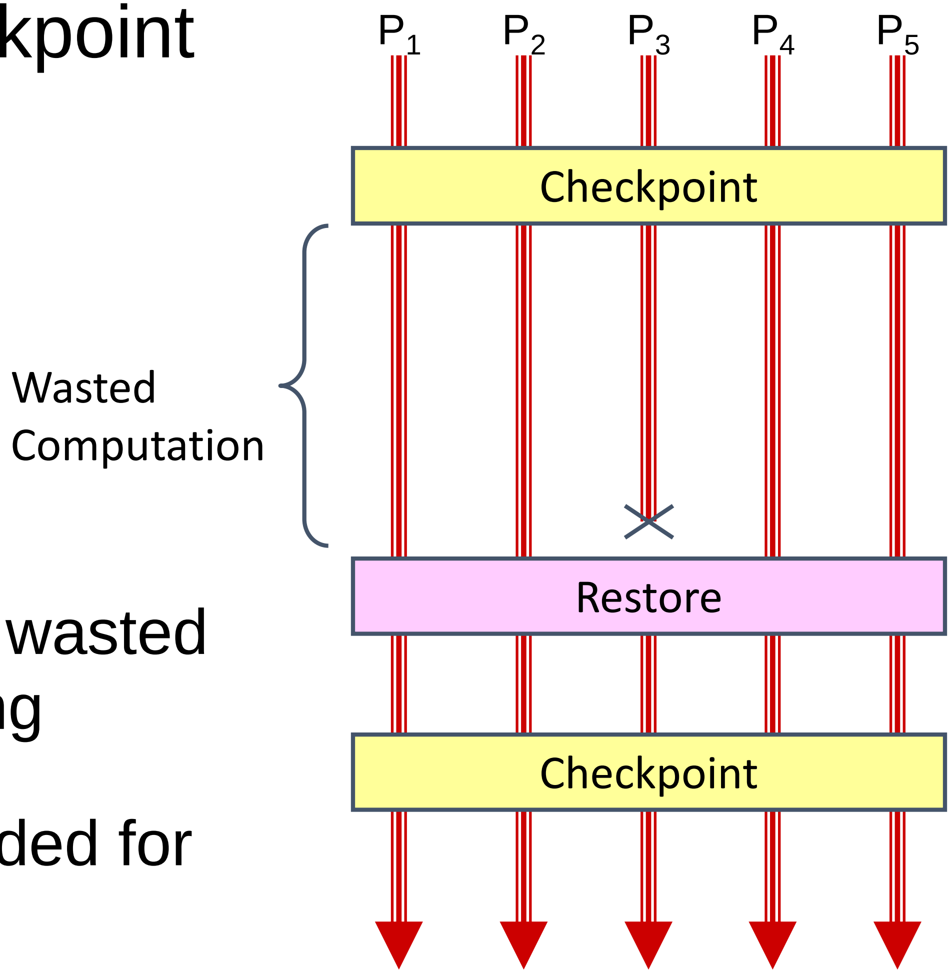


Back to the future: How do we program cluster?

- MPI is hard
  - Too low level programming for building cloud scale applications
  - Programmers need to explicitly deal with communication
- How do we deal with failures?
  - Everything can and will fail at data center scale
    - Software and hardware can fail
    - Failures can be persistent or transient
  - All cloud vendors have suffered major outages
- Imagine you have to write a program across 15,000 servers where any server can die at any second
  - How do you recover from failures?

How do we deal with failures?

- HPC/supercomputing applications checkpoint
  - Periodically write out state of all processes
- Then restore when failure occurs
  - Reset state to that of last checkpoint
- Not a suitable model for the cluster
  - Significant I/O traffic during checkpointing
  - All computations between 2 checkpoints is wasted
  - Performance is sensitive to number of failing components
  - Worse, checkpointing needs to be hand coded for each application



Other parallelization challenges

- Load balancing
  - How do we efficiently split up data across workers so that we keep all machines busy
- Synchronization
  - How do workers access a shared resource? Say update a shared file?
- ….
- Higher-order question: *Are there salient features of a large class of parallel applications we can exploit to make life easier?*
- What is required
  - Hide system-level details from the developers
    - No explicit communication, synchronization, failure handling…
  - Separating the *what* from *how*
    - What: Developer specifies the computation that needs to be performed
    - How: Execution framework ("runtime") handles actual execution

Prior experience:
Parallelism & declarative programming

- If you can express a problem declaratively, it's easier to parallelize
- Example: SQL
  - SELECT * from students where id='yourname';
- Databases do this in parallel
  - checking every record in the database against id 'yourname'
  - returning a list of the matching ones
  - …

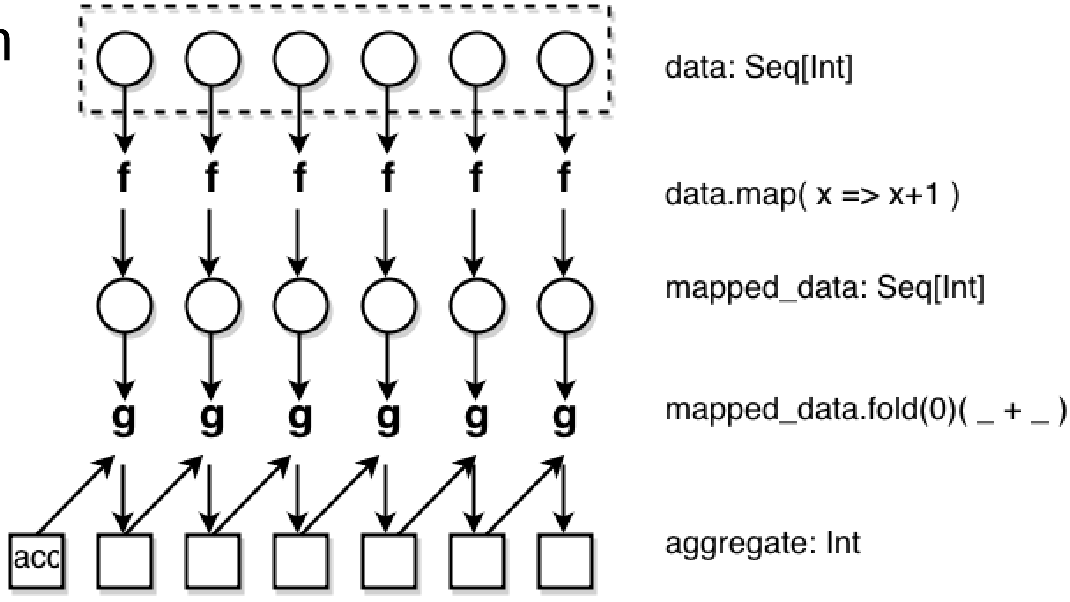
Prior Experience: Parallelization and functional programming

- If problem is expressed functionally, it’s often easier to parallelize

- **Map**
  - map takes as an argument a function f (that takes a single argument) and applies it to all element in a list
  - Common in ML: *List.map timestwo [1; 2; 3;]; ----> int list [2; 4; 6;]*

Parallelization and functional programming

- **Fold**
  - fold takes as arguments a function g   (that takes two arguments) and an initial value (an accumulator)
  - g is first applied to the initial value and the first item in the list
  - The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g
  - The process is repeated until all items in the list have been consumed
- Map and fold are higher order functions
  - Functions that take functions as arguments

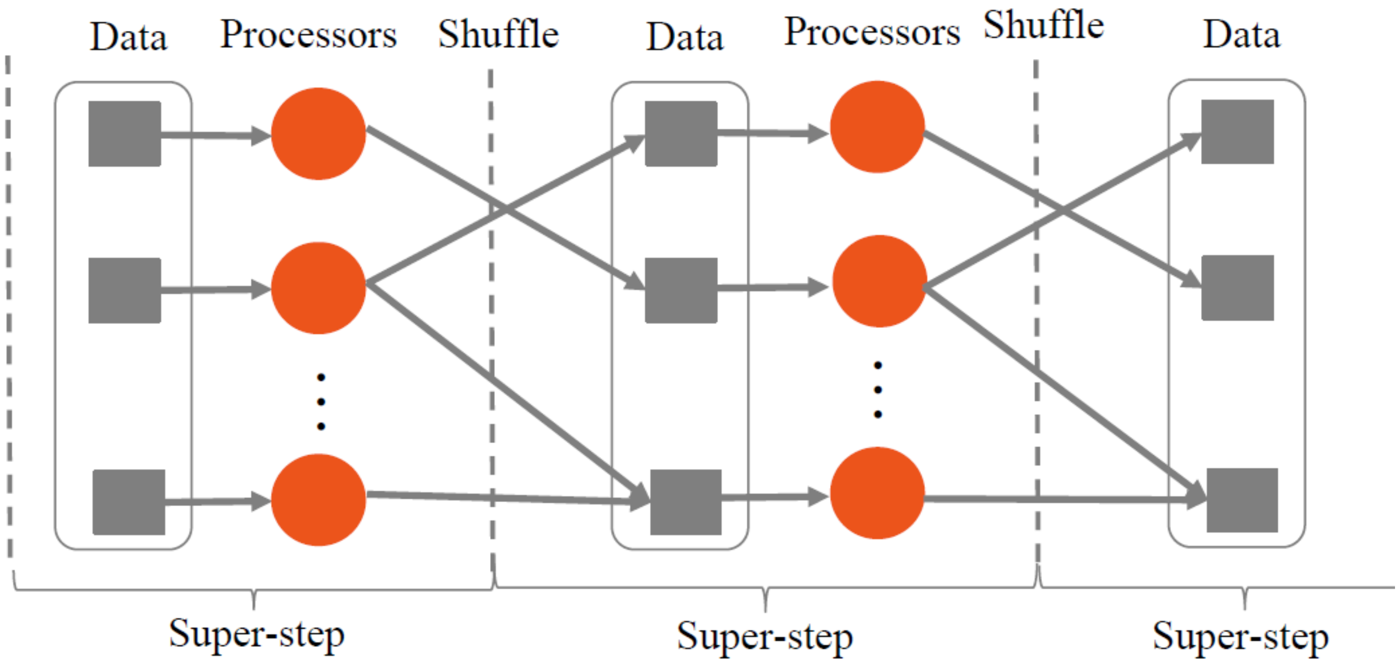


Parallelization and functional programming

- Let's Look closely at that Map function
  - map takes as an argument a function f (that takes a single argument) and applies it to all element in a list
  - Common in ML: *List.map timestwo [1; 2; 3;]; ----> int list [2; 4; 6;]*
- In what order do we have to apply the function to the elements?
  - Any one we want, even in parallel.
  - The function has no side effects - the applications are independent
- What happens if we apply the function to the same element twice?
  - Nothing, it's safe to re-do it and recompute the value - no side effects!
- Suggests a nice basis for both parallelization and fault tolerance...

Bulk Synchronous Processing

- BSP is a programming model for parallel computation introduced by Leslie Valiant
  - Used in many HPC applications
  - Inspired the way Google MapReduce was designed



Google’s MapReduce

- Introduced in 2004 by Jeff Dean and Sanjay Ghemawat
  - Used in Google for processing tens of EBs of data per day
  - Read “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004

- Users specify the computation in terms of a map and a reduce function

- Underlying runtime system automatically handles everything
  - parallelizes the computation across large-scale clusters of machines
  - handles machine failures, efficient communications, and performance issues

MapReduce data structures

- **Key-value pairs are the basic data structure in “Map Reduce”**
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

- **The design of “Map Reduce” algorithms involves**:
  - Imposing the key-value structure on arbitrary datasets
  - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content

MapReduce generic algorithm

- **The programmer defines a mapper and a reducer as follows**:
  - The mapper is applied to every input key-value pair to generate a intermediate key-value pairs

    map: (k1; v1) -> [(k2; v2)]

  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

    reduce: (k2; [v2]) -> [(k3; v3)]

- Map and Reduce are pure functions
  - They cannot keep state across calls
  - They cannot read or write files other than expected inputs/outputs
  - There's no hidden communication among tasks
  - Crucial for simplicity

Word Count in MapReduce

- Input:
  - Key-value pairs: (offset, line) of a file
  - *a*: unique identifier of a line offset
  - *l*: is the text of the line itself
- Mapper
  - Takes an input key-value pair, tokenize line
  - Emits intermediate key-value pairs: the word is the key and the integer 1 is the value
- The reducer
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences

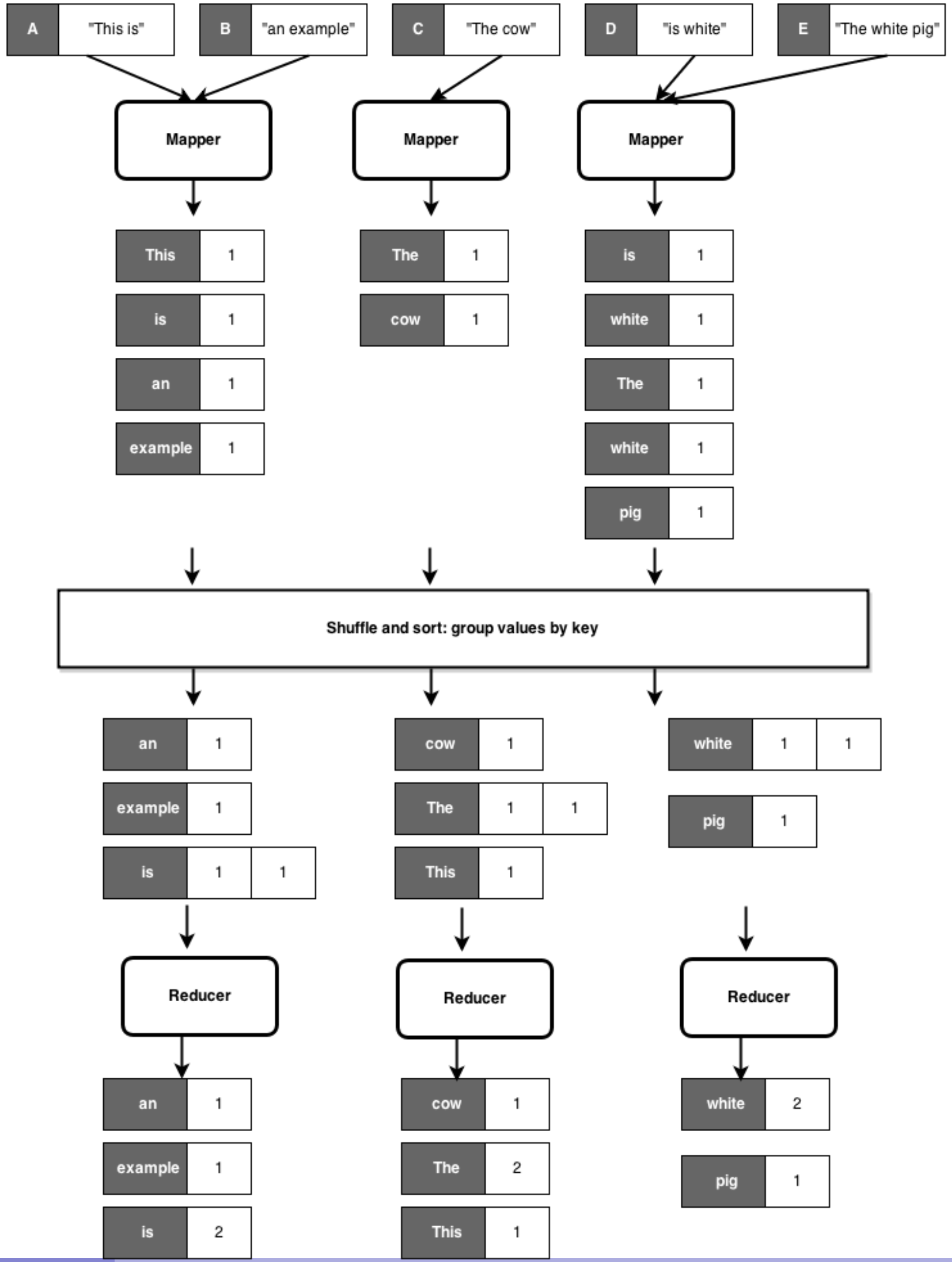
class MAPPER
    method MAP(offset a, line l)
        for all term $t \in$ line l **do**
            EMIT(term t, count 1)

class REDUCER
    method REDUCE(term t, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
            $sum \leftarrow sum + c$
        EMIT(term t, count sum)

Word Count in MapReduce

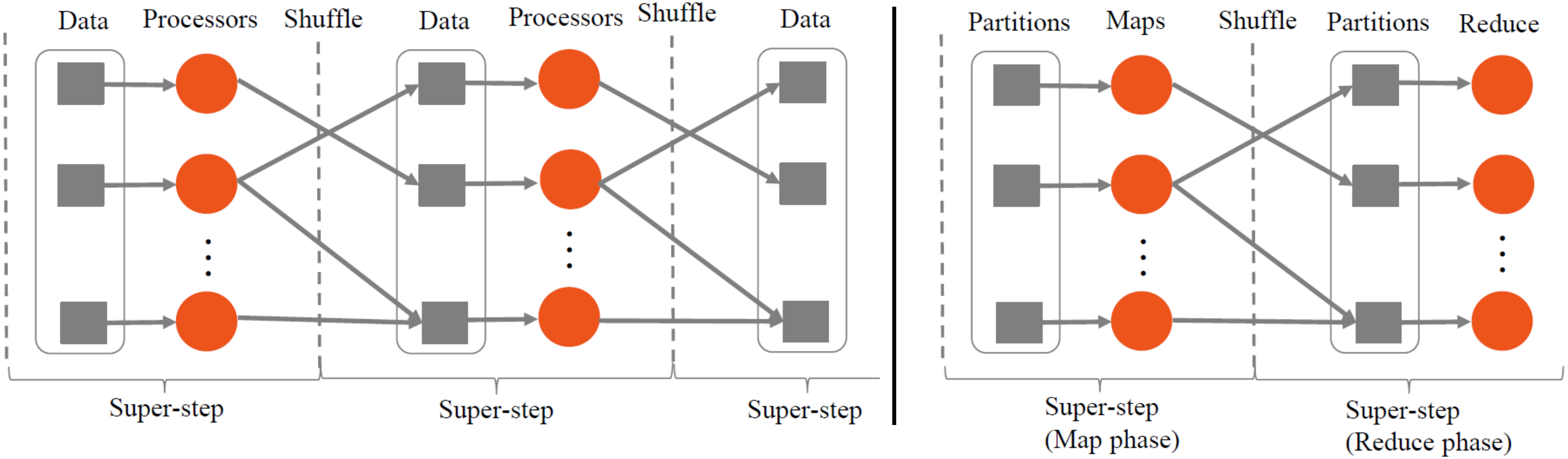
- Implicit between the map and reduce phases is a parallel “group by” operation, also called **shuffle**, on intermediate keys

- The framework guarantees all values associated with the same key (the word) are brought to the same reducer



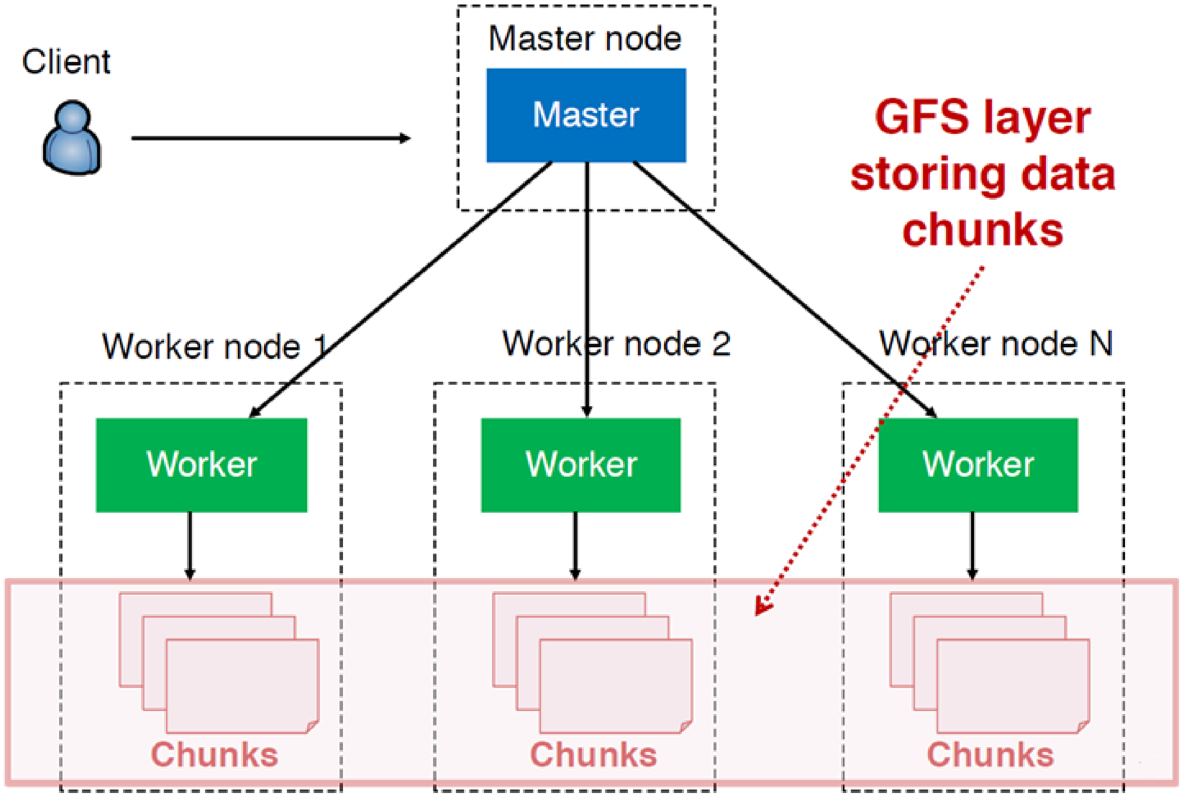
MapReduce, functional programming, BSP

- Equivalence of "Map Reduce" and Functional Programming
  - The map of Hadoop MapReduce corresponds to the map operation
  - The reduce of Hadoop MapReduce corresponds to the fold operation
  - Unlike the fold we saw, their "fold" a.k.a Reduce is partitioned by key
- Mapreduce as restricted BSP



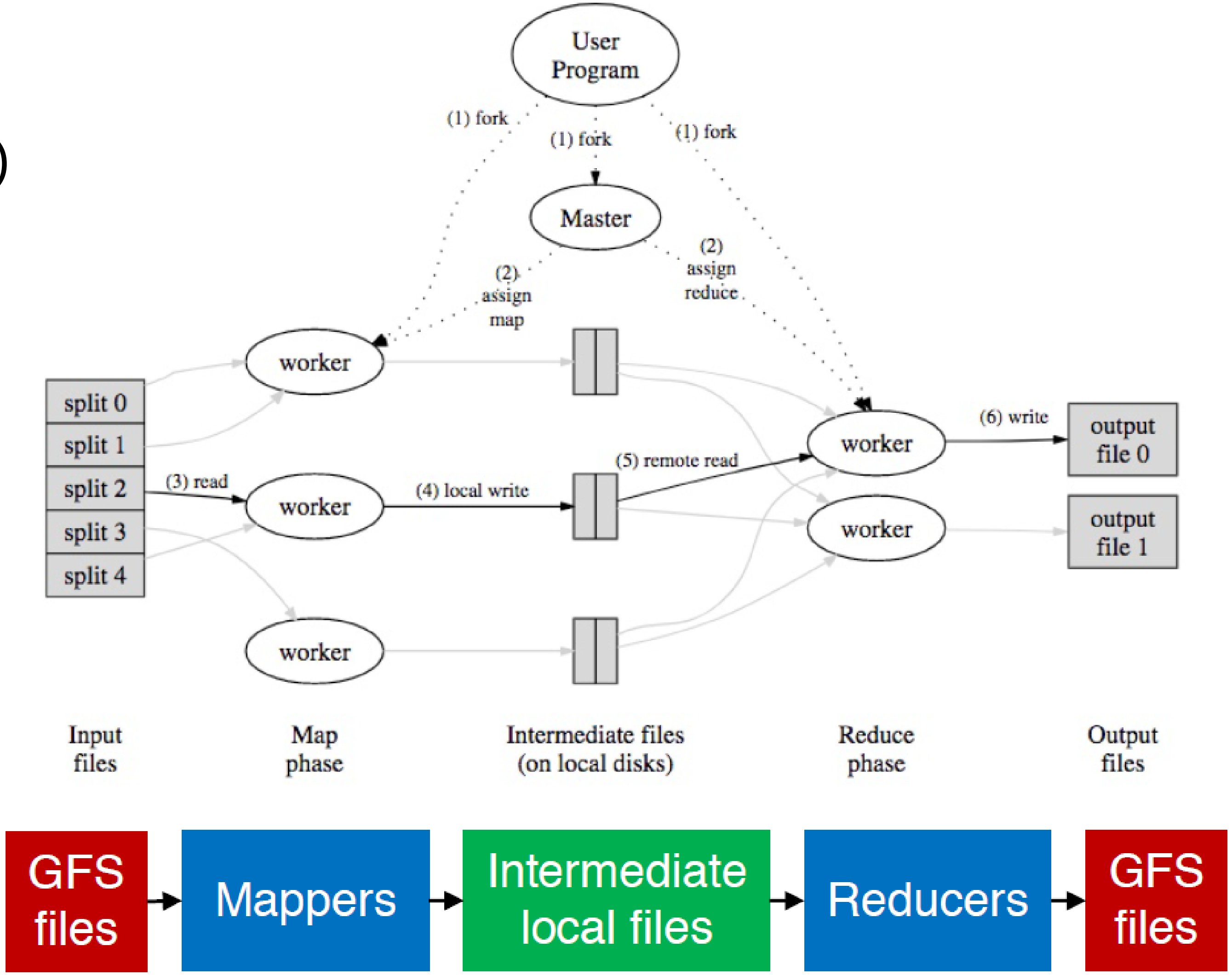
MapReduce Architecture

- Runtime reads and writes data from/to Google File System
  - Input & Output: Set of files in reliable file system
- Master breaks work into *tasks*
  - Master schedules tasks on workers dynamically
- MapReduce workers run on same machines as GFS server daemons
  - Remember data locality principle



MapReduce data flow (from the paper)

1. Library splits files into 16-64MB pieces
2. Master picks workers and assigns map or reduce task (M map, R reduce tasks)
3. Map worker reads input split, calls map function, buffers map output in memory
4. Periodically, in-memory data flushed to disk & master is informed of disk location
5. Master notifies reduce worker of location, reduce worker reads map output files, sorts data
6. Reduce worker iterates over sorted data, passes each unique key, list of values to reduce function. Output of reduce function written out.



MapReduce: Fault tolerance

- Server Crashes are detected with ***heartbeats***
- Map worker crashes:
  - Intermediate Map output is lost. It will be needed by every Reduce task!
  - Master re-runs map, spreads tasks over other GFS replicas of input.
    - Replication in GFS ensures data access inspite of server failures
  - All reducer tasks are notified of new execution. Reduce tasks that have not read intermediate data from failed task read from new task
- Reduce worker crashes.
  - Finished tasks are OK -- stored in GFS, with replicas.
  - Master re-starts worker's unfinished tasks on other workers.
  - How do we deal with 2 reduce workers writing to same file? (hint: Atomic rename)

MapReduce: Fault tolerance

- Master stores several data structures
  - For each map/reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine
  - For each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task

- Master crash
  - Could checkpoint master data structure.
  - But rare enough that they simply aborted computations.

- Net-net: Mapreduce is highly resilient
  - 80 node outage for several minutes during MR workflow

MapReduce benefits

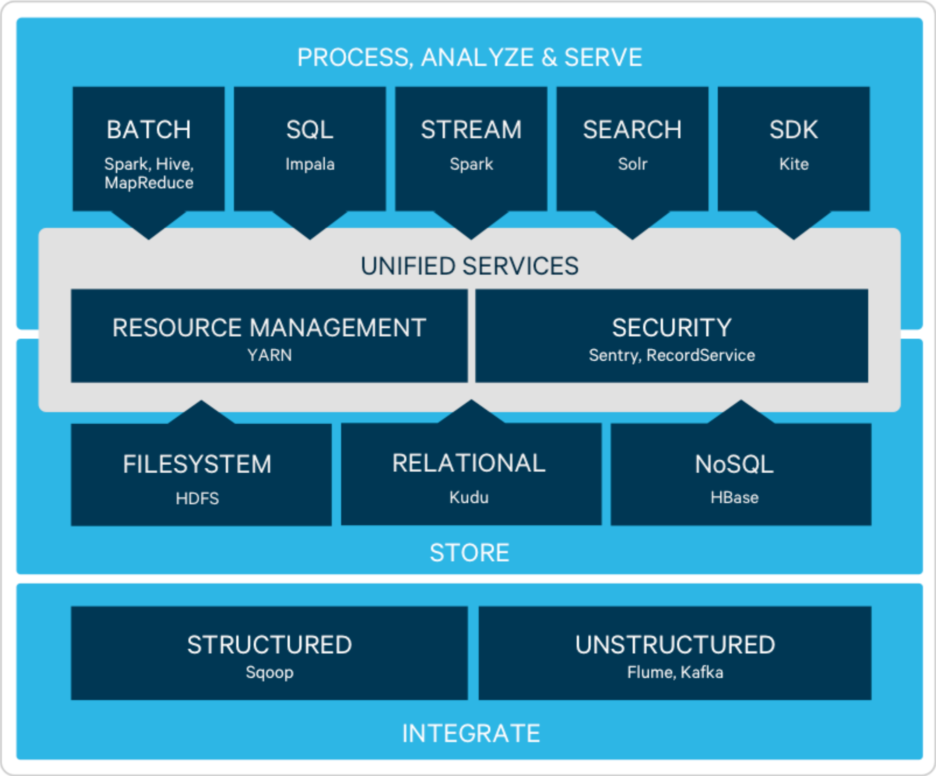
- Widely adopted within Google after 2003
  - large-scale machine learning problems
  - clustering problems for the Google News and Froogle products
  - large-scale graph computations
  - Index system that produces data structured used by Google Search
    - 20TB of files stored in GFS
    - The indexing process runs as a sequence of five to ten MapReduce operations.
- MapReduce made big cluster computation popular
  - Designed to run large batch jobs over Big Data
  - Scales well
  - Easy to program -- failures and data movement are hidden.

From MapReduce to Hadoop

- Hadoop was created by Doug Cutting and Mike Cafarella
- Apache Nutch was a open source web search engine
  - Part of the Apache Lucene text search engine project
  - Web crawl and indexing generated huge data repositories
  - Several algorithms needed to run at scale
- Google publishes GFS and MapReduce papers in 2004
  - Nutch Distributed File System and Nutch MR implementation in 2005
  - Moved out of Nutch into a project called Hadoop
  - Doug Cutting joins Yahoo!, Hadoop becomes web-scale project
  - February 2008 Yahoo! Announces that production search index was being generated by a 10,000-core Hadoop cluster
  - 2008, Hadoop becomes top-level Apache project

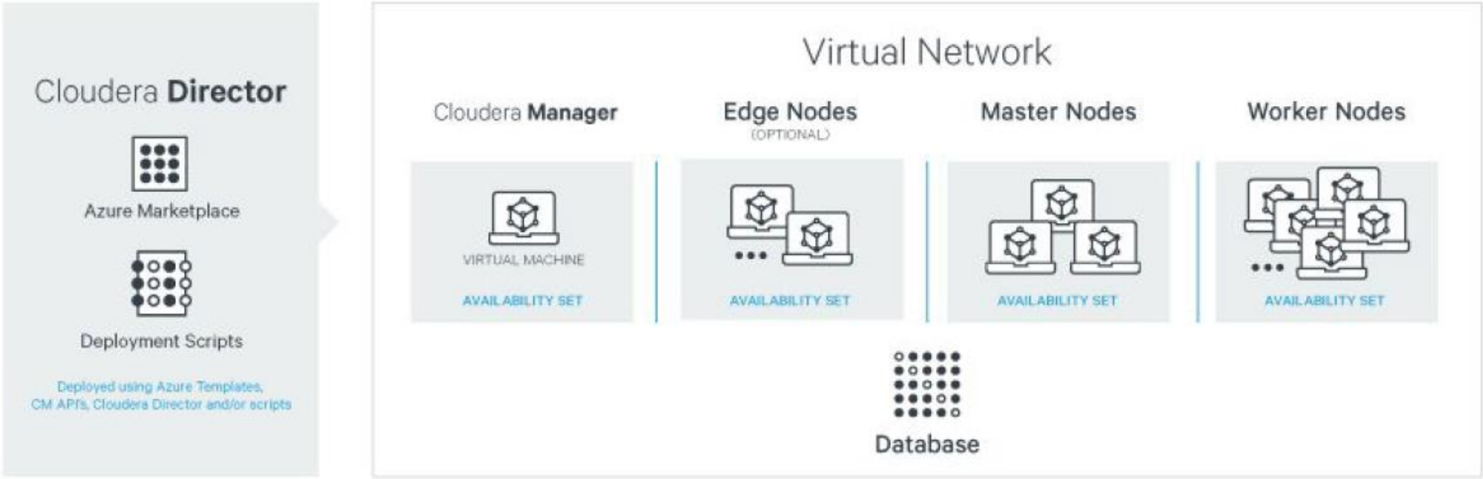
Hadoop Ecosystem today: The Cloudera Enterprise Data Hub

- Cloudera founded in 2008, the lead Hadoop flag bearer today
- Today Hadoop ecosystem is a rich, ever-evolving collection of open-source projects for ingesting, storing, and processing data



Hadoop in the Cloud: IaaS

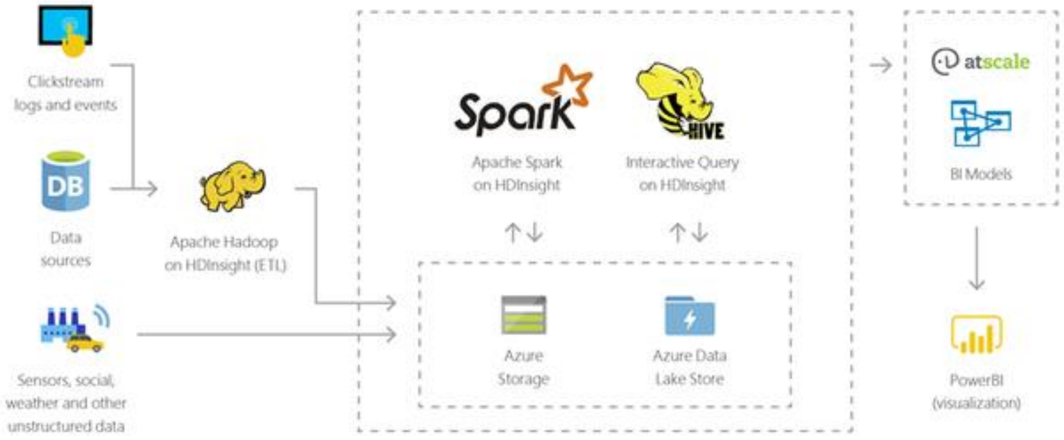
- Use VMs to run Cloudera EDH from Azure Marketplace
  - Use recommended instance types for various Hadoop nodes
  - Use preconfigured Cloudera CentOS image as OS
  - Cloudera Director for deploying, monitoring and elastic scaling
  - You pay license to Cloudera + as-per-use price for VMs/storage/net
  - You administer your VMs, Hadoop cluster



See [Cloudera Enterprise Reference Architecture for Azure Deployments]

Hadoop in the Cloud: PaaS

- Azure HDInsight offering from Microsoft on Azure cloud
  - Internally based on HortonWorks Data Platform
  - Spin up Hadoop clusters on demand, scale them up or down based on your usage needs, and pay only for what you use.
  - Integrated with Data Factory (Pipeline automation) and Data Lake Storage service
  - Pay on use, no VM/cluster/Hadoop administration



Hadoop in the Cloud: FaaS

- You could build a serverless MapReduce framework
  - Write Map & Reduce as functions
  - Write an *Orchestrator* function to execute Map & reduce fns
- Problem: Orchestrator needs state
  - Need to track which mappers ran/finished, …
  - Remember normal functions are stateless
- Solution: Special ***Durable*** functions in Azure
  - Can maintain state
  - Can call other functions
  - Automatically checkpoint progress to save state
- Serverless Mapreduce
  - Build orchestrator as a durable function
    - distributes the workload across multiple mappers
    - Coordinate outputs to the Reducer
    - Return back the computed values