OUTLINE

- Cloud computing
  - Introduction
  - Virtualization
  - Public/private/hybrid clouds
  - How to develop containerized applications
    - Docker
    - Kubernetes
- **Scientific computing**
  - Massive parallel processors (MPP)
  - Computing grids
  - Programming paradigms for parallel distributed computing
    - MPI
    - MapReduce
    - Apache Hadoop

SCIENTIFIC COMPUTING

- Until now, we have focused on **high throughput computing (HTC)**.
  - Applications in business and web services.
  - Focus on reliability and availability (*services* rather than *jobs*).
  - Typically loosely coupled components.
- Applications in science and engineering → **high performance computing (HPC)**
  - Focus on computing power for short amounts of time (*jobs* rather than *services*).
  - Typically highly parallel, tightly coupled jobs.
  - Low latency critical.
  - We use the term *scientific computing* with the same meaning.

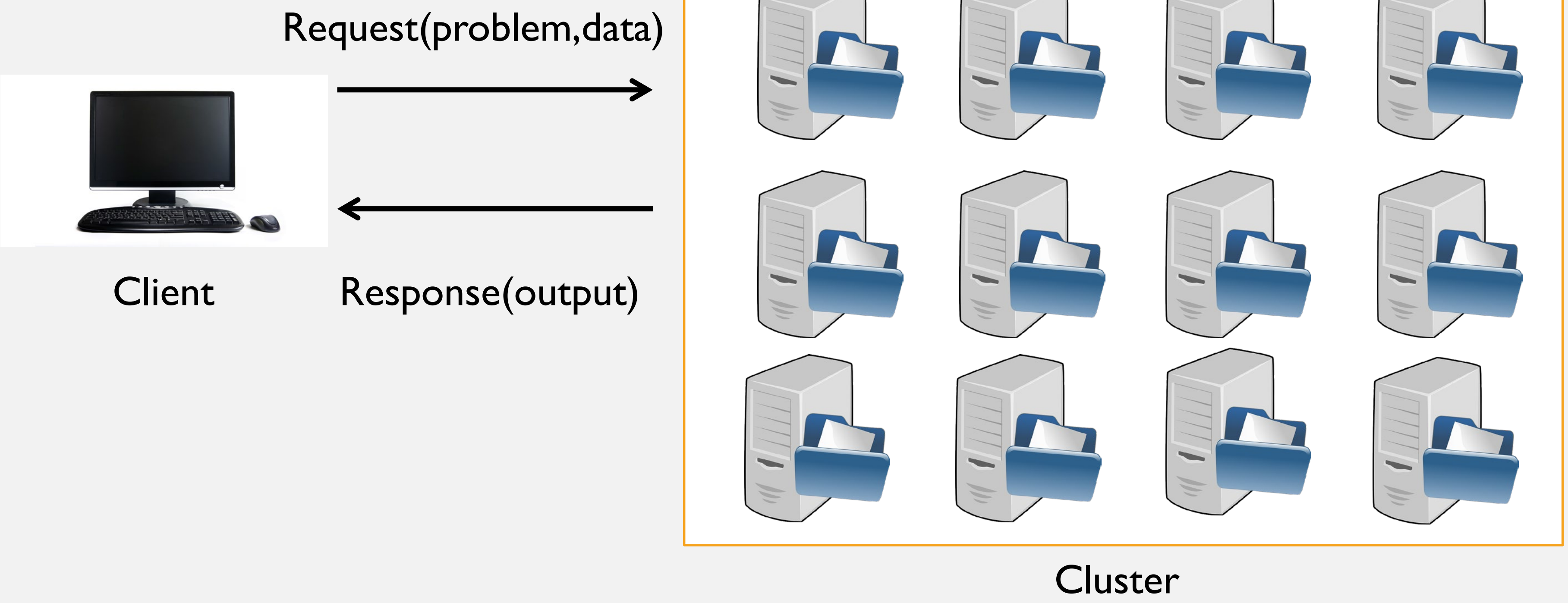
SCIENTIFIC COMPUTING

How to implement HPC?

- Massive parallel processing (MPP)
  - Computing clusters
  - Computing grids

SCIENTIFIC COMPUTING

Computing clusters



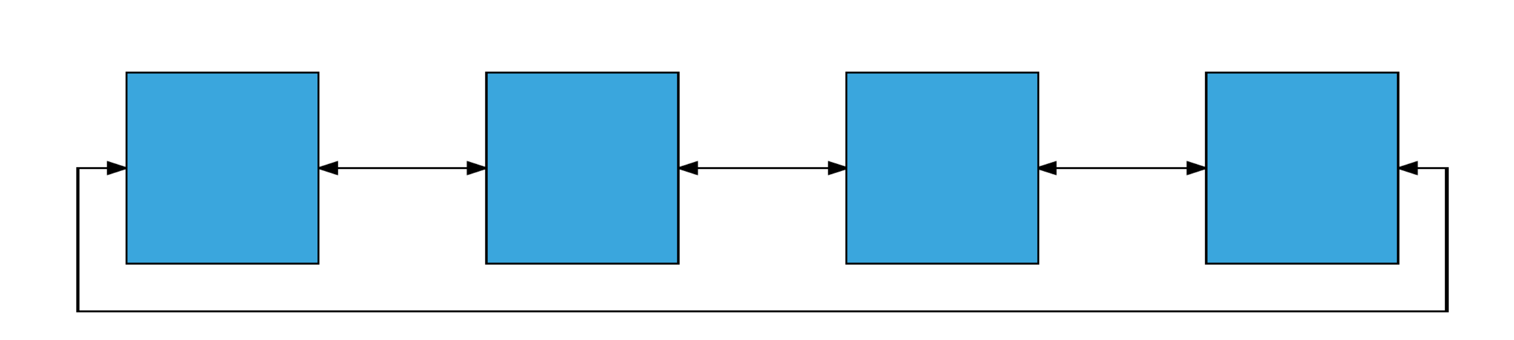
SCIENTIFIC COMPUTING

Computing clusters

- Stand-alone computers, collectively operating as a **single** resource pool.
- Achieves high availability (computing resources are redundant and interchargeable).
- Homogeneous clusters from commodity hardware more cost effective than mainframes/vector supercomputers.
- Today's supercomputers are mostly computing clusters with special hardware and network topology.
  - Special packaging.
  - Low latency.
  - Bandwith.
  - Reliability.

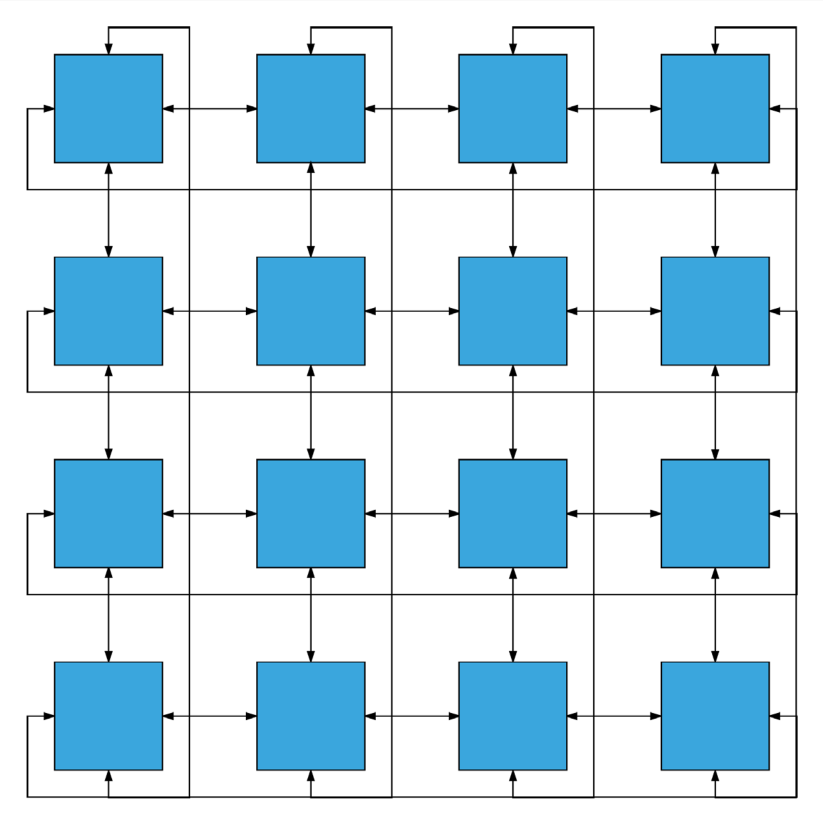
SCIENTIFIC COMPUTING

Example: torus network topology (1D)



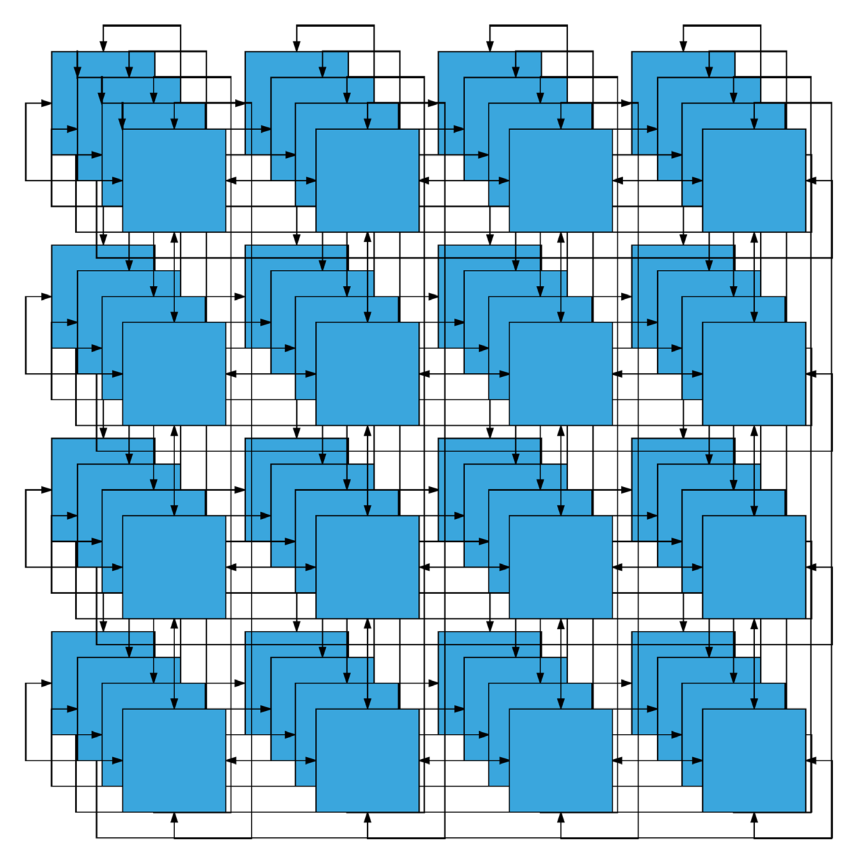
SCIENTIFIC COMPUTING

Example: torus network topology (2D)



SCIENTIFIC COMPUTING

Example: torus network topology (3D)



SCIENTIFIC COMPUTING

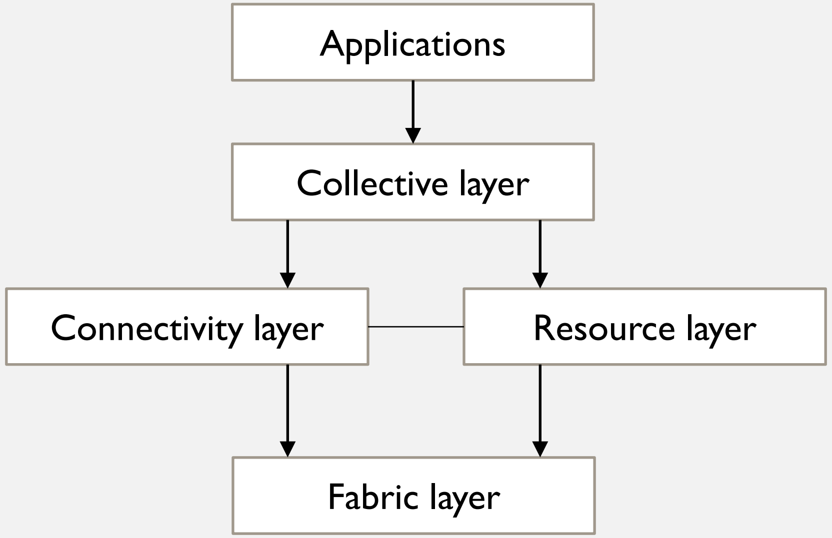
Computing grids

- Heterogeneous and independent computing units.
  - Also across organization boundaries.
  - Computers, laptops, clusters, databases, etc.
- Oportunistic collaboration
  - Use of idle resources.
- Communication via WAN/LAN.
- Examples:
  - Volunteer grids: seti@home, folding@home.
  - European Grid Infrastructure (EGI): links computing resources to support international research projects.

SCIENTIFIC COMPUTING

Grid computing

- Example architecture (Forster, Kesselman, Tuecke, 2001)



Discovery, allocation, scheduling…

Resource management

Hardware level

SCIENTIFIC COMPUTING

Grid computing

- **Application layer:** applications that make use of the grid environment
- **Collective layer:** handles access to multiple resources (discovery, allocation, scheduling)
- **Resource layer:** manages sharing of single resources. Implements access control, configuration information, specific operations on the resource
- **Connectivity layer:** supports transactions using multiple resources (i.e. transfer data between resources
- **Fabric layer:** interfaces to specific resources at a specific site. Includes all computational resources

OUTLINE

- Cloud computing
  - Introduction
  - Virtualization
  - Public/private/hybrid clouds
  - How to develop containerized applications
    - Docker
    - Kubernetes
- Scientific computing
  - Massive parallel processors (MPP)
  - Computing grids
  - **Programming paradigms for parallel distributed computing**
    - MPI
    - MapReduce
    - Apache Hadoop

PROGRAMMING PARADIGMS

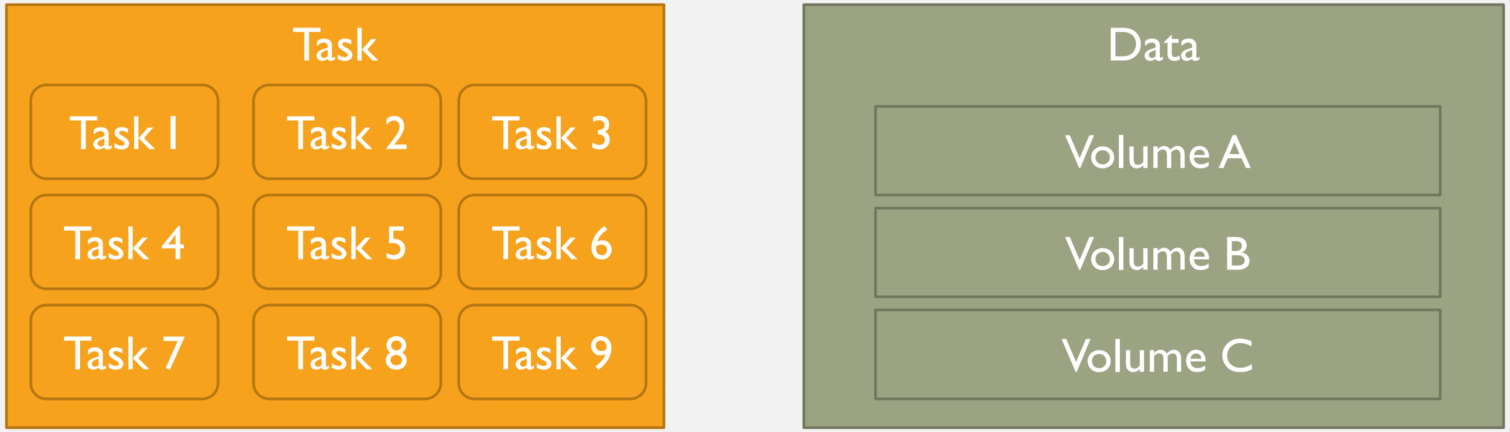
Parallel distributed computing

- Simultaneous use of more than one computational engine to run a job or application.
- Computational engines: distributed systems (clusters, clouds, grids, etc).
- What are the advantages and disadvantages of running a program in parallel?

PROGRAMMING PARADIGMS

Basic concepts

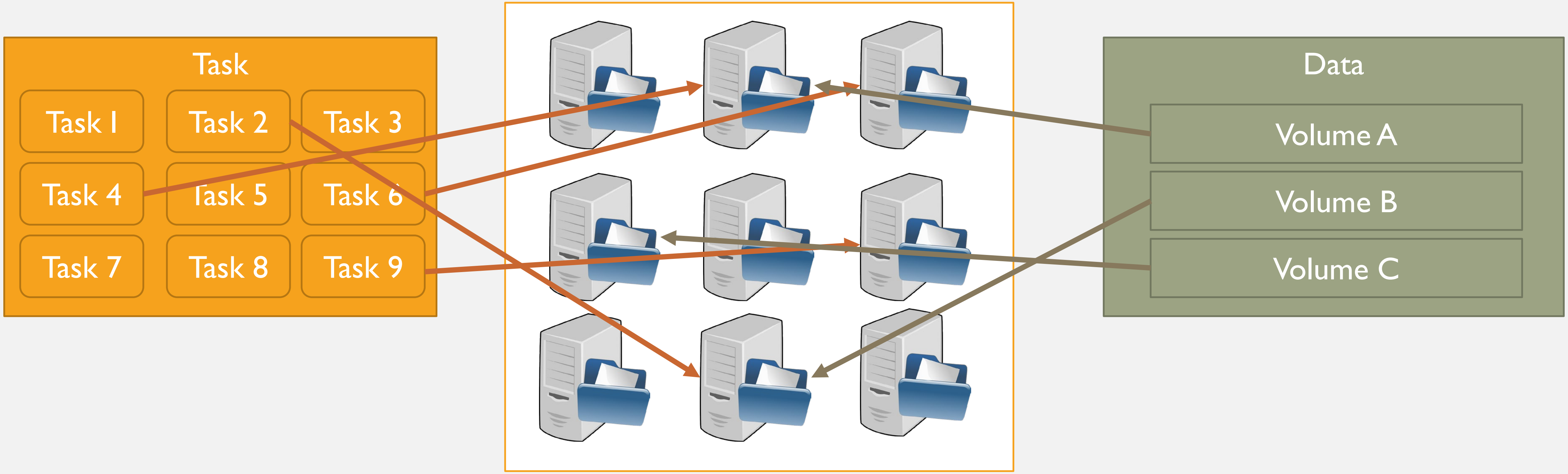
- **Partitioning:** can be done in two levels
  - *Job partitioning*: which tasks can be computed in parallel.
  - *Data partitioning*: how to divide data that can be processed by different workers.



PROGRAMMING PARADIGMS

Basic concepts

- **Mapping:** assigns partitions to resources. Usually done by a *resource allocator*.



PROGRAMMING PARADIGMS

Basic concepts

- **Synchronization:** prevent race conditions and manage dependencies.
- **Communication:** send data/programs to workers and manage communication between workers.
- **Scheduling:** manage which tasks to schedule next/to which worker(s).

PROGRAMMING PARADIGMS

Why do we need programming paradigms?

1. Improve productivity.
2. Decrease time-to-market.
3. Use resources more efficiently.
4. Increase system throughput.
5. Higher level of abstraction.

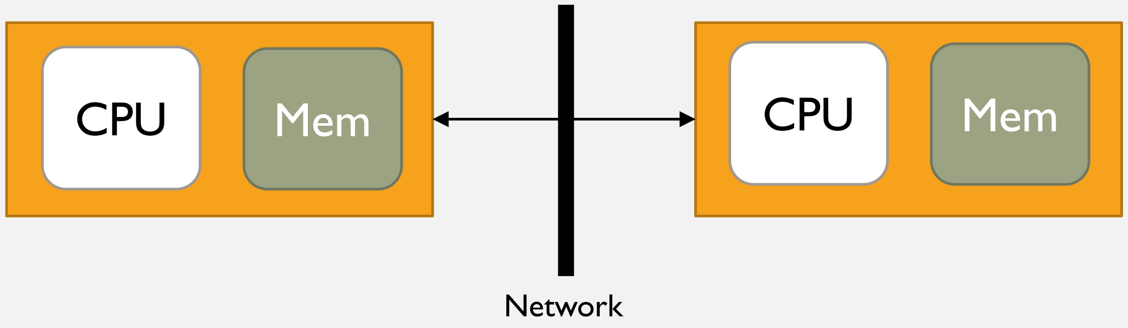
Some examples

- Message-Passing Interface (MPI)
- MapReduce
- Apache Hadoop

PROGRAMMING PARADIGMS

Message Passing Interface (MPI)

- Message-passing parallel programming model: data is moved from one process to another by means of cooperative operations (messages).
- MPI standard (currently version 4) adopted by many vendors and implementations.
- Distributed memory model.



PROGRAMMING PARADIGMS

Message Passing Interface (MPI)

- Defines *communicators* and *groups* to define which processes communicates with each other.
- Each process has an unique *rank* (integer), which identifies it.
- Processes communicate via point-to-point message passing calls.
- Send and receive can be synchronous or asynchronous calls.
- System buffer holds data in transit
  - Receiver not ready.
  - Many messages are received at once.

PROGRAMMING PARADIGMS

Message Passing Interface (MPI)

- Collective communication primitives (scope: communicator)
  - Broadcast: all data is sent to all processes.
  - Scatter: data is distributed among processes.
  - Gather: data is collected from other processes.
  - Reduce: a computation is performed on data gathered from other processes.
- Communication patterns between processes can be modeled by *virtual topologies*.
  - Example: torus topology.
- Example: CERN has a MPI cluster.

PROGRAMMING PARADIGMS

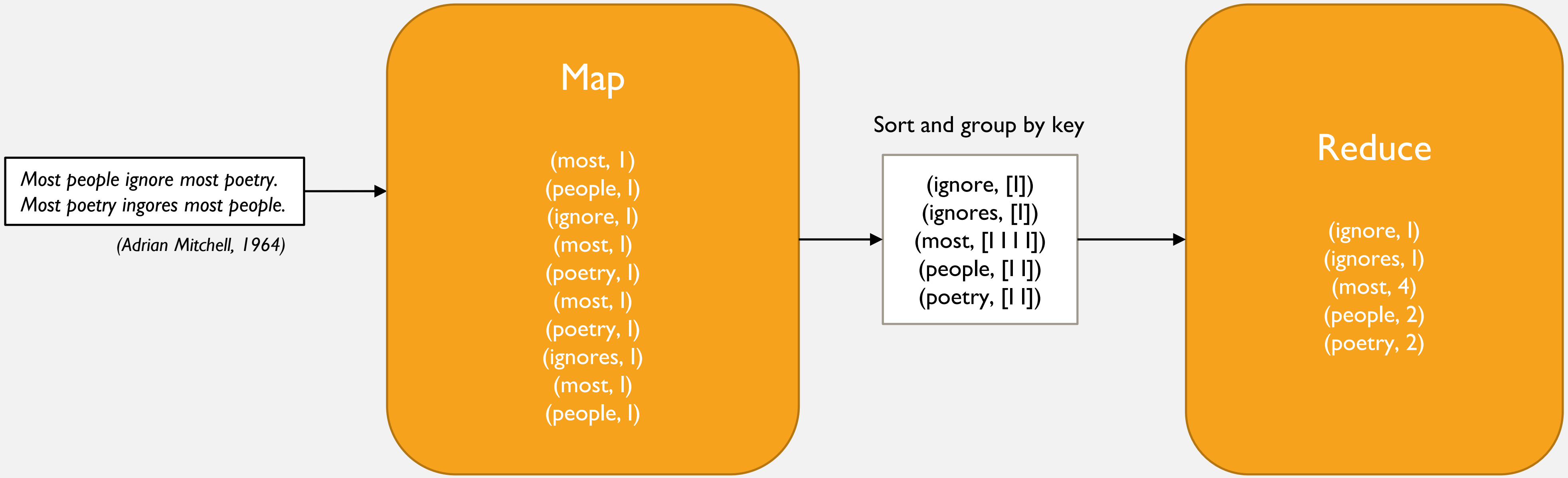
MapReduce

- Focus on parallel and distributed computing on large data sets.
- Proposed by Google in combination with the Google File System (GFS).
- User abstractions in the form of *Map* and *Reduce* functions.
  - Developer needs to implement both functions.
- Data in *(key, value)* form.
- **Map:** parallel pre-processing of input data.
  - Produces intermediate (key, value) pairs.
- **Reduce:** processes data with same key.
  - Combines (key, [values]).

PROGRAMMING PARADIGMS

MapReduce classical example: word count

- Input: a number of text lines.
- Output: word count for each word.



PROGRAMMING PARADIGMS

MapReduce

- Map processes each input pair in parallel producing intermediate (key, value) pairs)
  - Example: 2 parallel executions of Map, one for each text line.
- Framework sorts intermediate pairs and groups them by key.
- Grouped keys are passed as argument to the Reduce function, one parallel execution for each group.
  - Example: 5 parallel executions of Reduce.

PROGRAMMING PARADIGMS

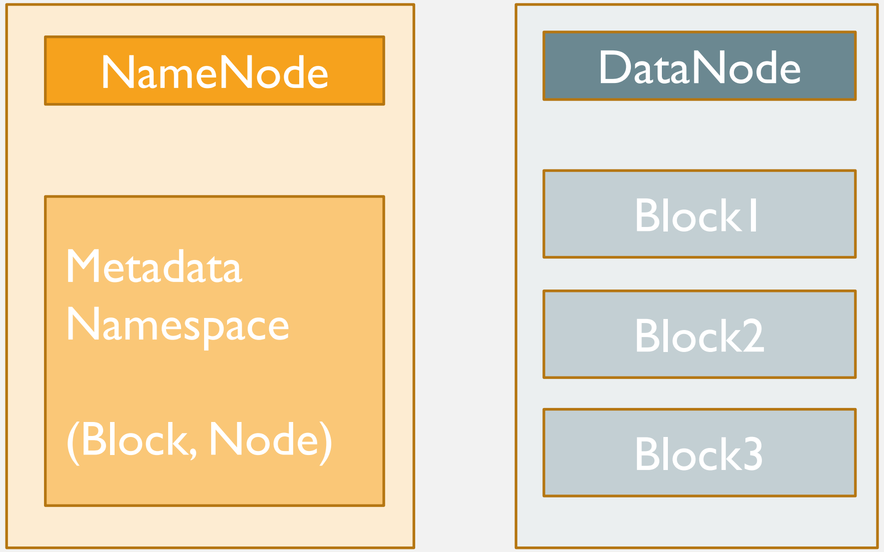
MapReduce

- GFS: data is split into blocks of fixed size (i.e. 64 KB).
- Master/worker architecture: Map and Reduce programs are sent to worker nodes.
- Each block is an input to a Map function and processed in parallel.
- Blocks are stored redundantly on worker nodes. *Programs are sent where the input data are.*
    - Avoid communication costs.
- Data have to be sent to Reduce nodes → potential bottleneck.
- Output file is written to GFS.

PROGRAMMING PARADIGMS

Apache Hadoop

- Open source Java implementation of MapReduce (Cutting and Cafarella, 2006).
- Instead of GFS, own implementation: Hadoop Distributed File System (HDFS).
  - Inspired by GFS.
- HDFS master/slave architecture
  - Master: NameNode.
  - Slave: DataNode.
- HDFS design goals:
  - Fault tolerance (through replicas)
  - High throughput



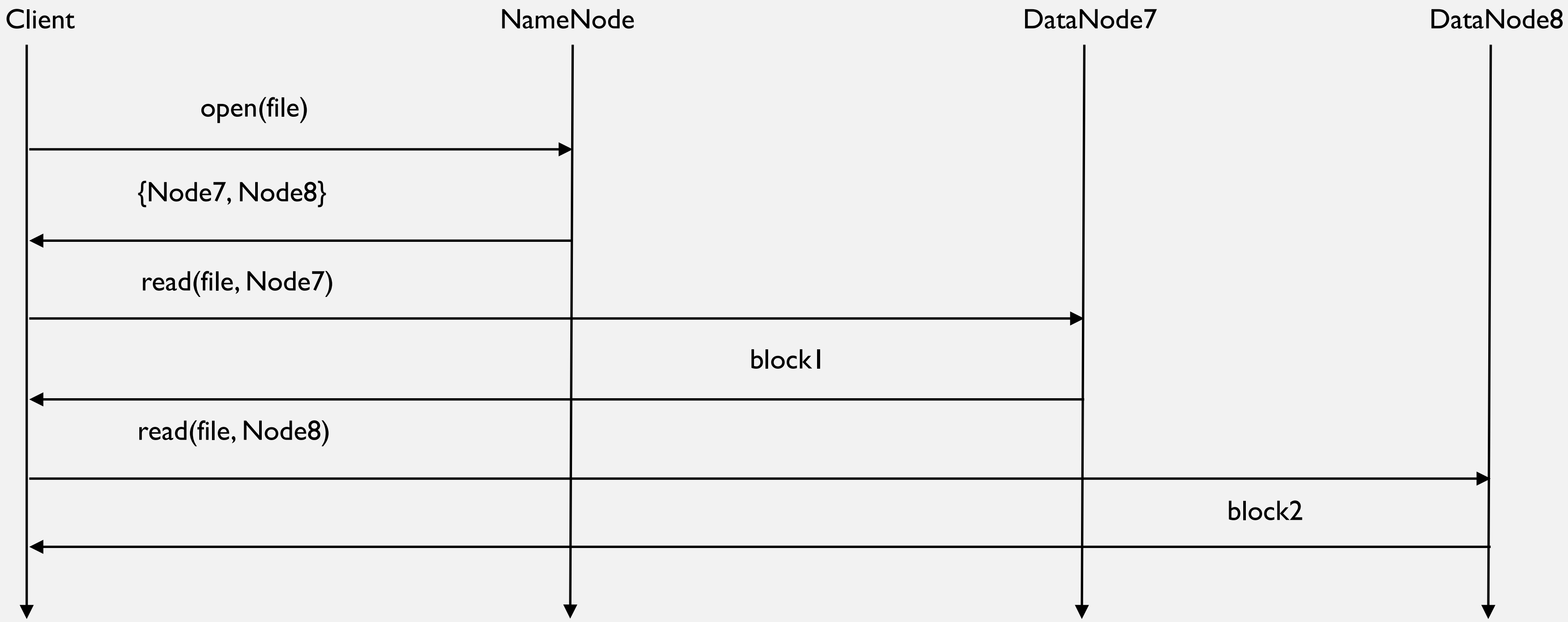
PROGRAMMING PARADIGMS

Apache Hadoop

- HDFS Fault tolerance: replication factor defaults to 3 (3 copies of each block).
  - One copy in the same node (reduce communication costs).
  - One copy in the same rack (reduce communication costs).
  - One copy in another rack (improve reliability).
- Large block size (64 MB, 128 MB) to allow for high throughput.

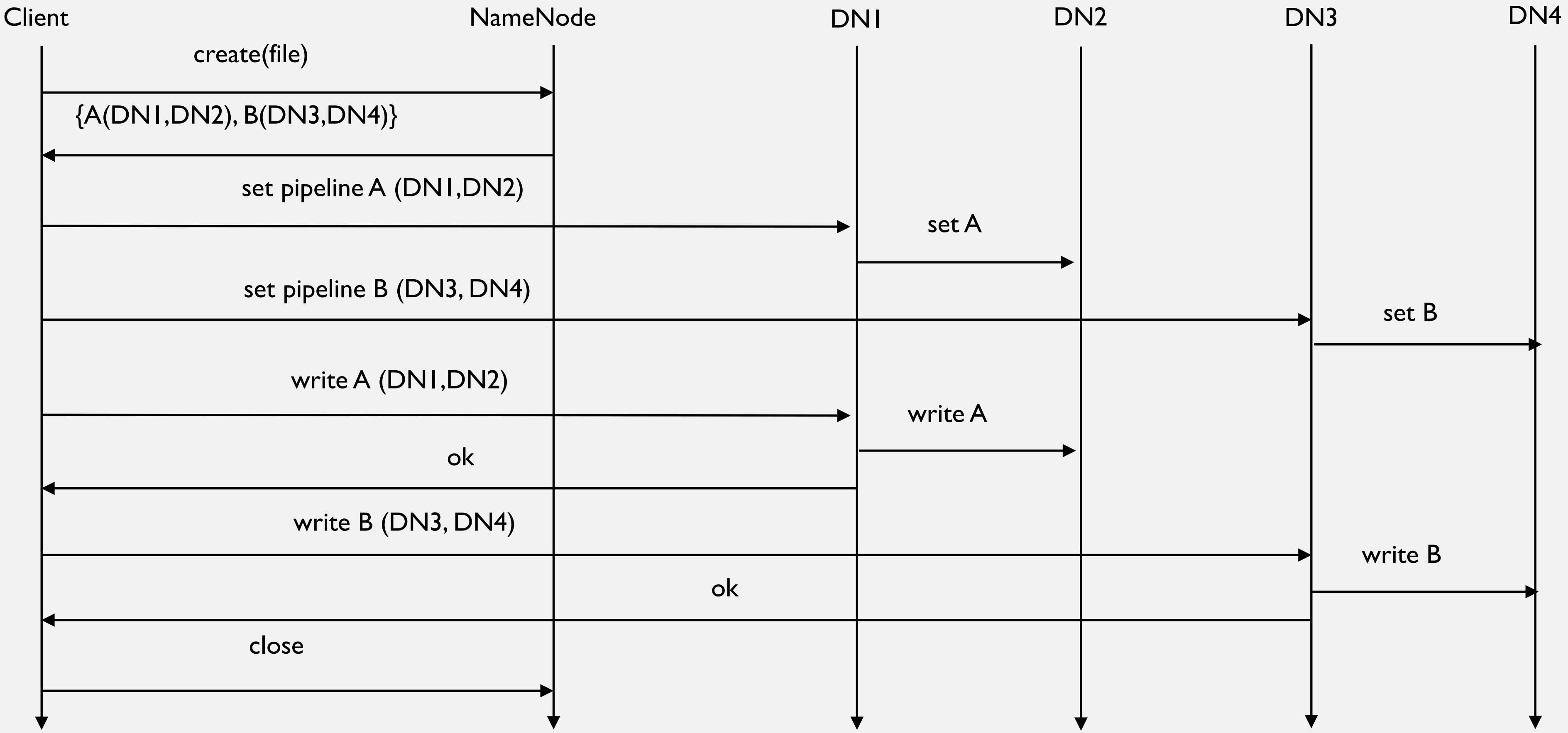
PROGRAMMING PARADIGMS

HDFS: Read file



PROGRAMMING PARADIGMS

HDFS: Write file



PROGRAMMING PARADIGMS

Apache Hadoop

- MapReduce engine for computing Map and Reduce tasks.
- Master/worker architecture: JobTracker, TaskTracker.
- Worker nodes typically both DataNode and TaskTracker.
- Workflow:
  - User submits job to JobTracker (JAR file + metadata) + input data splits.
  - Map tasks are assigned by JobTracker to TaskTrackers (considering data localization).
  - Number of reduce tasks specified by user.

PROGRAMMING PARADIGMS

Apache Hadoop