Lecture III

Message Passing Interface
and MapReduce

Message Passing Interface

- MPI is a **standard** for communication in super computing environments, which is
  - Simple
  - Network-Agnostic
  - Fast

Message Passing has a long tradition in computing, but many approaches finally converged to MPI.

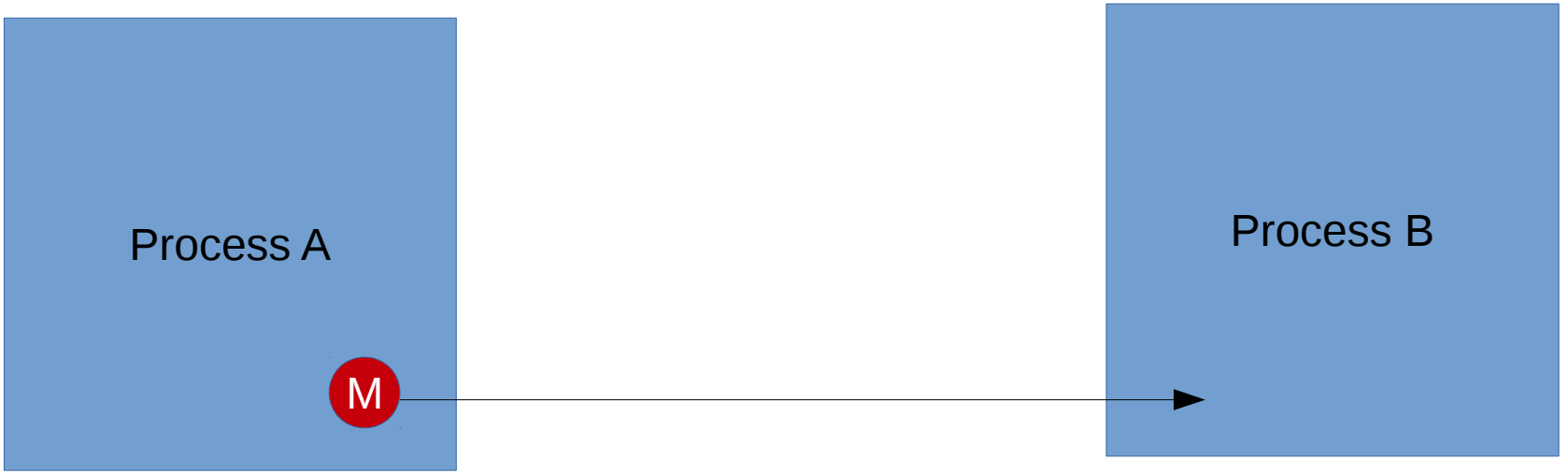
Implementations

- OpenMPI (https://www.open-mpi.org/)
  - Designed for the common case

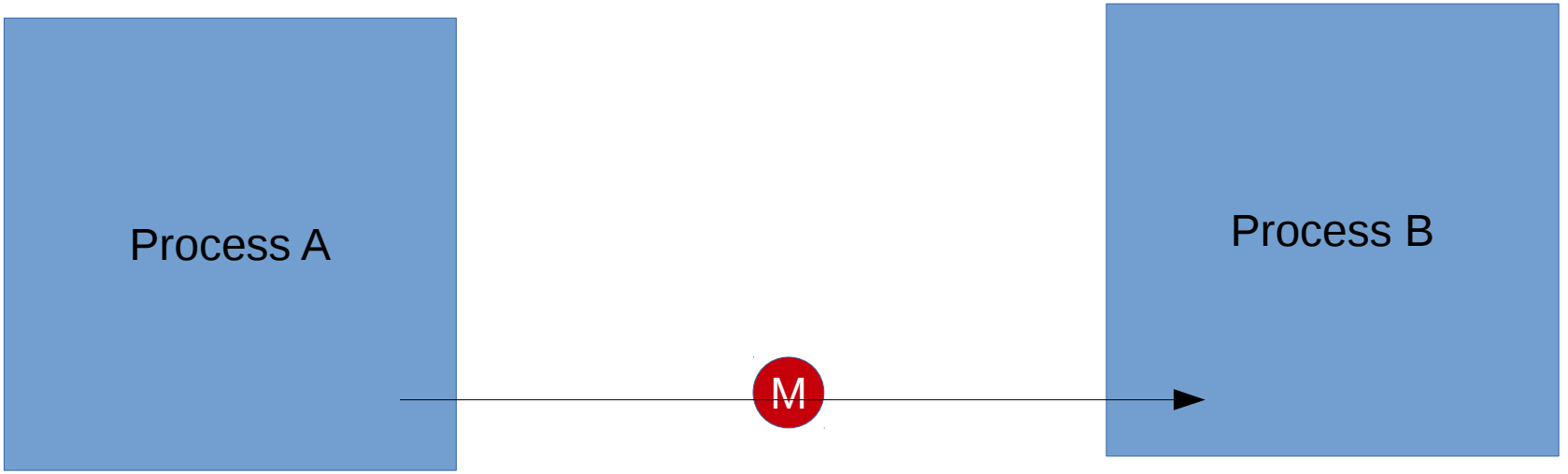
- MPICH
  - High performance
  - Widely portable
  - Derivates from Intel, Cray, and others

Running exclusively on the top 10 super computers including Taihu Light (June 2016)

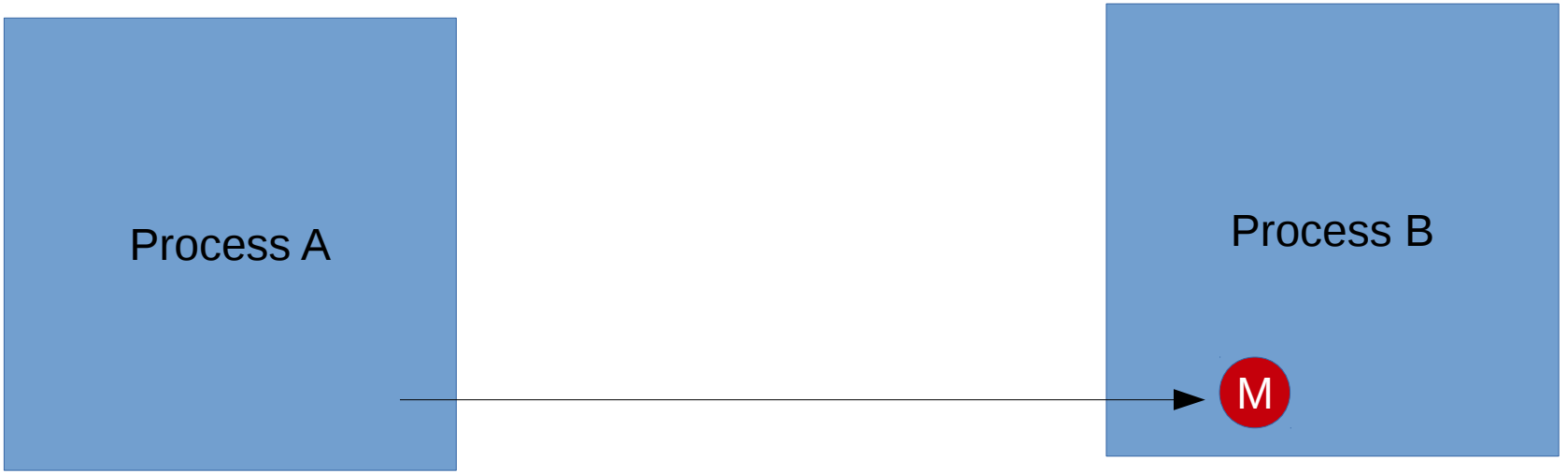
Message Passing



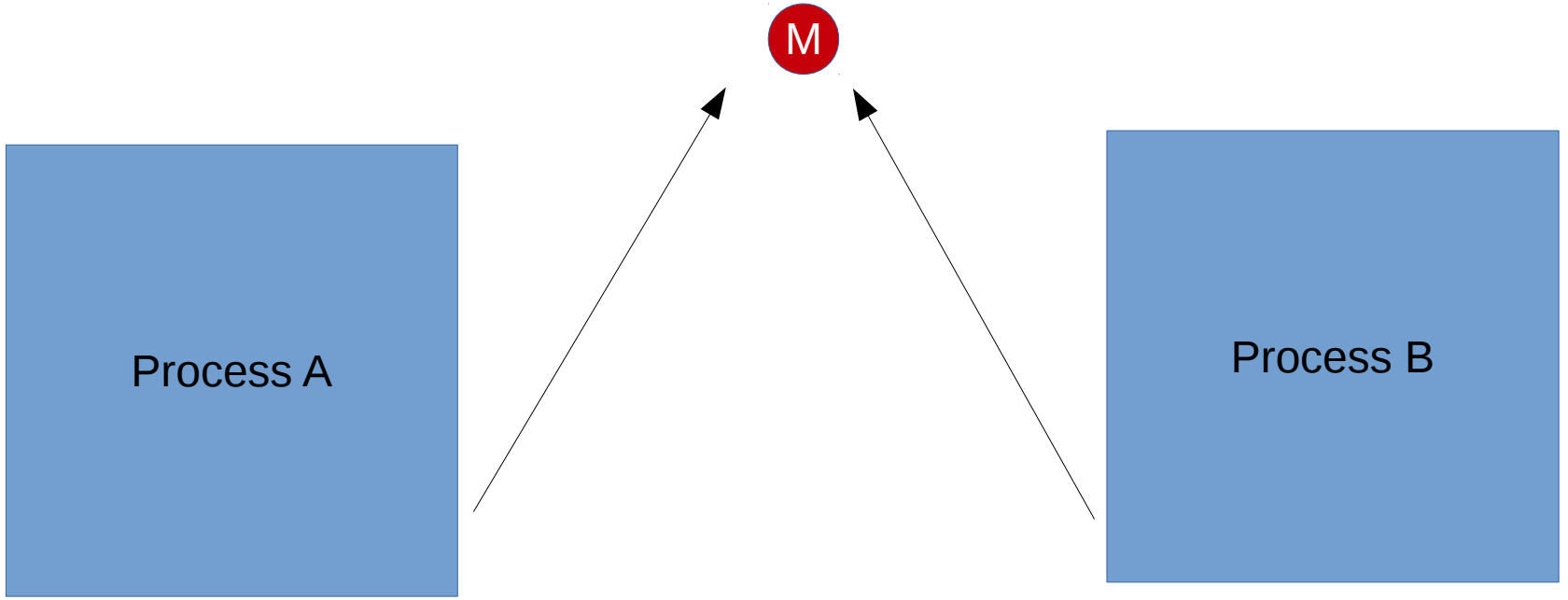
Message Passing



Message Passing



In contrast: Shared Variables



- Locking, Conflicts, Waiting, ...

Basic Operations (API)

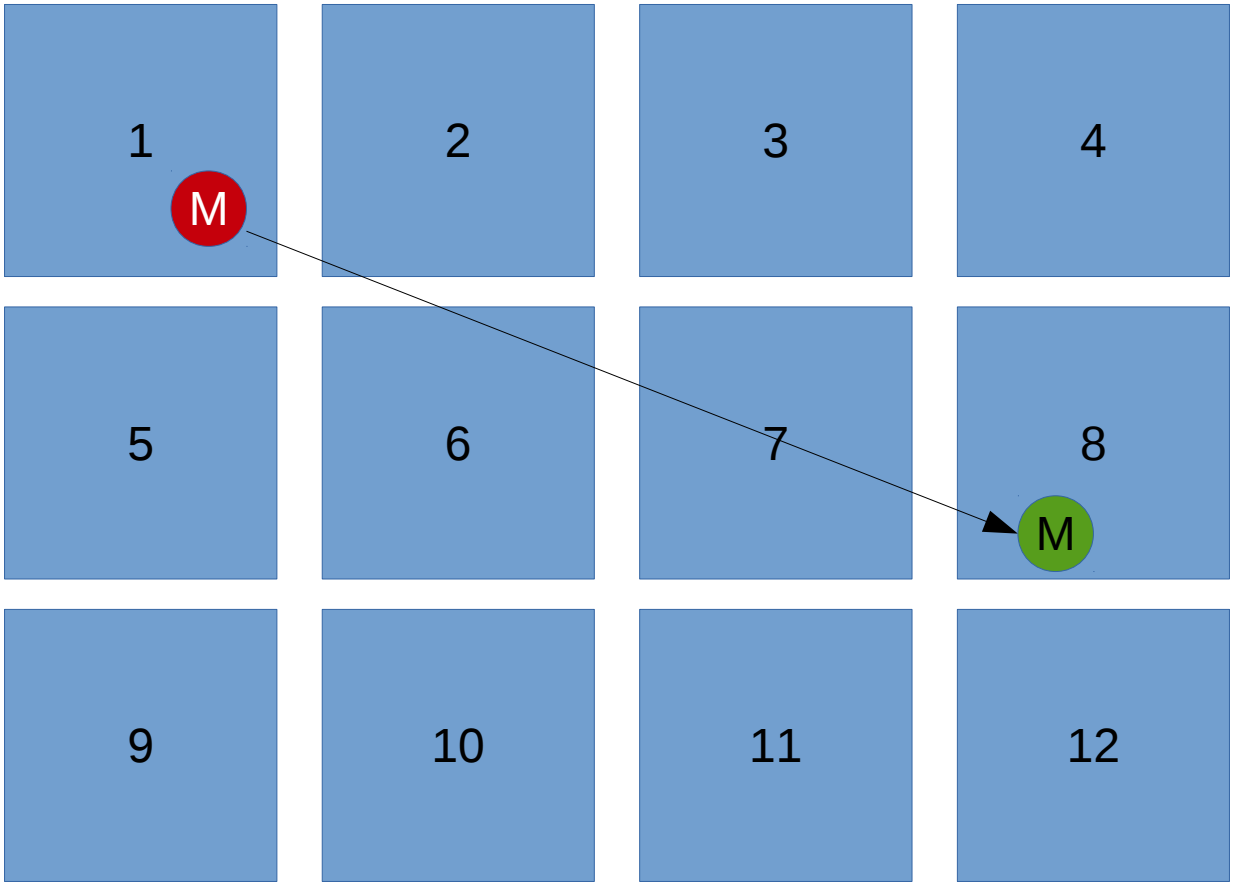
Some book-keeping (MPI_Init,...)

- **MPI_Send:** Send messages with different communication models
- **MPI_Recv:** Receive Messages

Why?

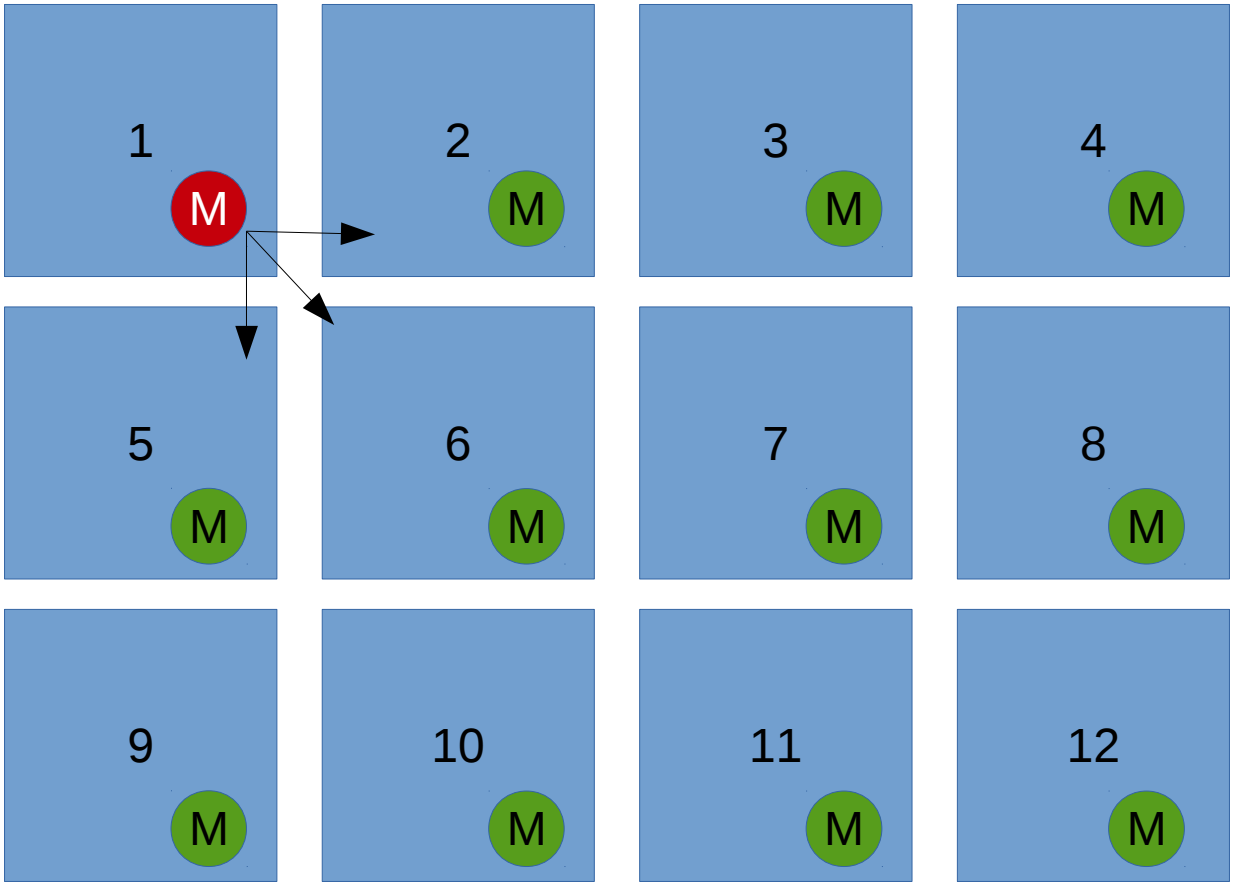
- Can't I just do this with sockets?
  - Yes, technically, one would be able
- Why then, MPI?
  - Because it provides higher layer **communication patterns**, which can be optimized by the MPI library
  - Because **integer rank** is simpler than IP adresses
  - Because it would **automatically use shared memory** between processes on the same node.

Communication Models
Point-To-Point



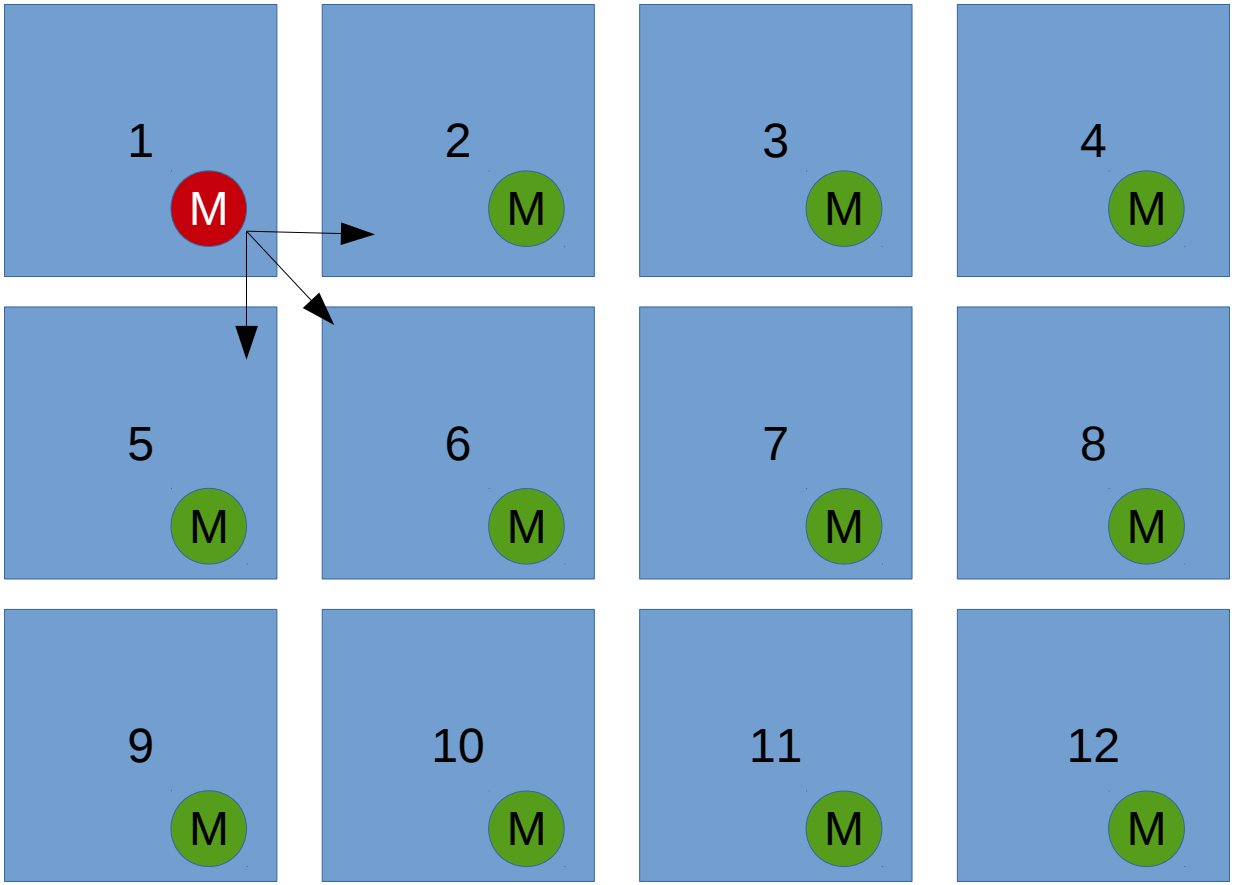
Messages can be sent **Point-to-Point** by specifying the Message Destination by its rank.

Communication Models
Broadcast



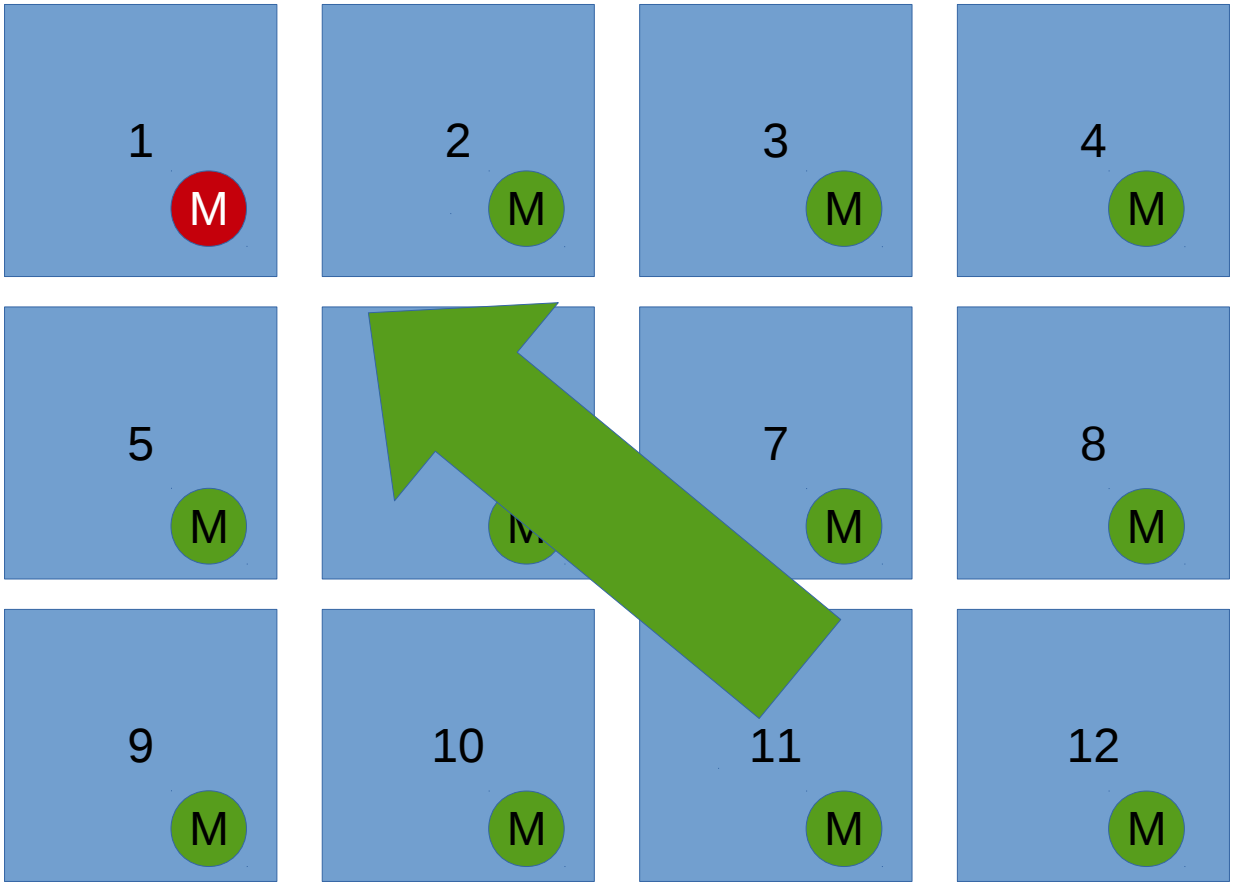
Messages can be **broadcast** to all other computers.

Communication Models
Scatter



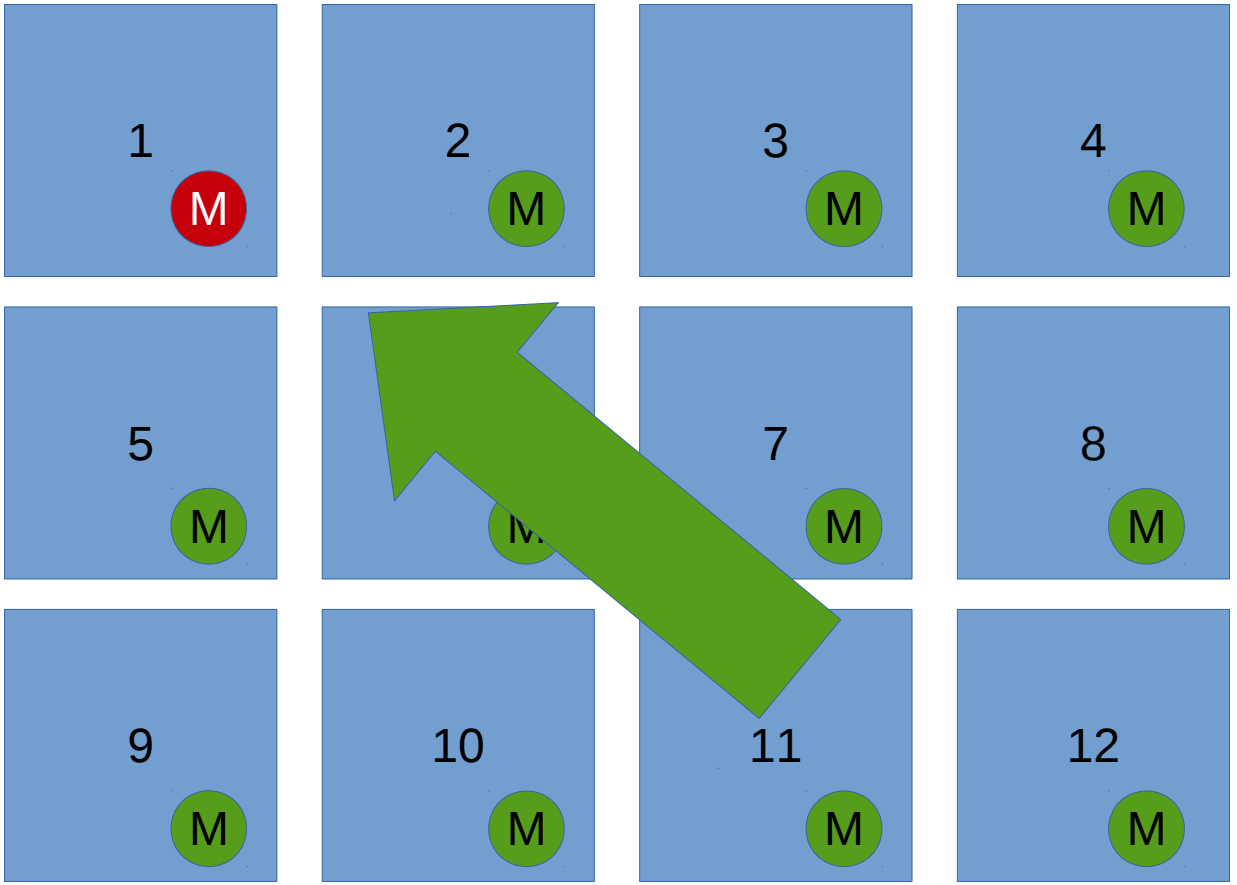
A large array can be **scattered as chunks**, which are distributed across nodes

Communication Models
Gather



One node can **gather** values from all other nodes.

Communication Models
Reduce



MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_LAND
MPI_LOR -
MPI_BAND
MPI_BOR
MPI_MAXLOC
MPI_MINLOC

One node can **reduce** values from all other nodes with builtin operations.

MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

Some Details

- 6 functions suffice for MPI:
  - MPI_Init: Initialize the libraries
  - MPI_Comm_Size: How many processes
  - MPI_Comm_Rank: My number
  - MPI_Send() Send information
  - MPI_Recv() Receive information
  - MPI_Finalize() Cleanup

Summary

- The most important communication patterns supported by MPI are:
  - Point-To-Point
  - Broadcast
  - Scatter
  - Gather
  - Reduce

  Directly supported is also the combination of **gather** and **broadcast** under the name
  - AllGather

Hello World

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

int main( int argc, char *argv[] )
{
    int rank;
    int size;

    MPI_Init( 0, 0 );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Passing a Message

```
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

Passing a Message

```
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

Additional Functions to Know

- **MPI_Probe:** Get the size of a message before receiving it
- **MPI_IO**: Extension for reading and writing files, especially important, when complex storage area networks are being used.
- **MPI_Barrier:** Global synchronization: All programs wait until all other reach this point.

MPI Interoperation

- MPI works well with
  - Threads
  - OpenMP
  - Debuggers

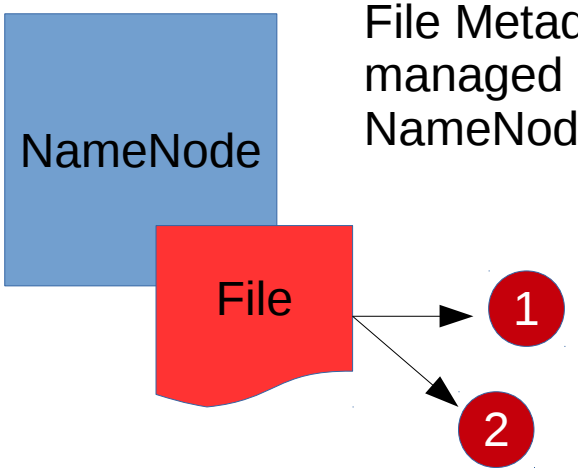
Map Reduce – A framework for big data

Distributed File Systems

Modern Cloud Architectures are largely based on distributed file systems.

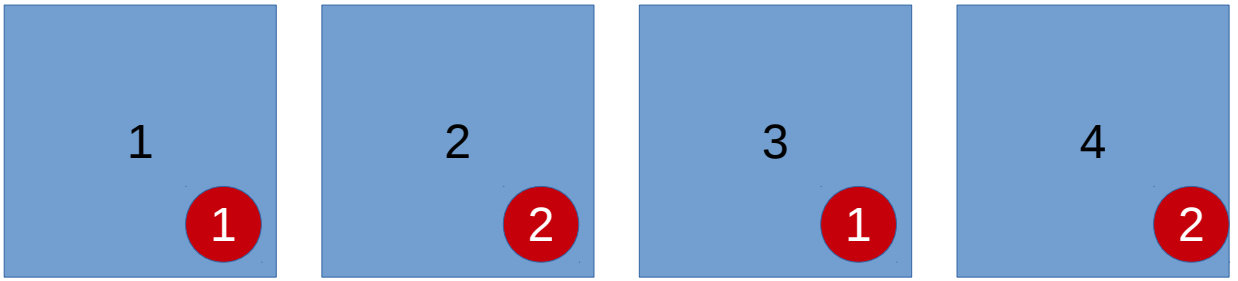
- HDFS (Hadoop File System) is the de-facto default for open source big data applications
- Distributed File Systems are based on splitting files into **chunks** and distributing these chunks **between servers** with **replication**.

HDFS Overview



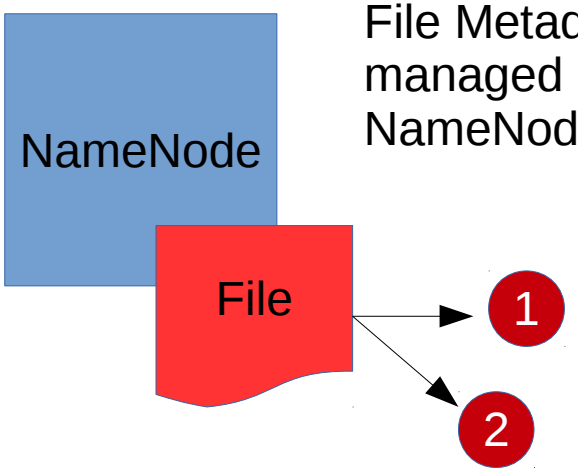
File Metadata and Access is managed by a dedicated NameNode

File contents is split into pieces Smaller than a predefined constant, e.g., 128 MB



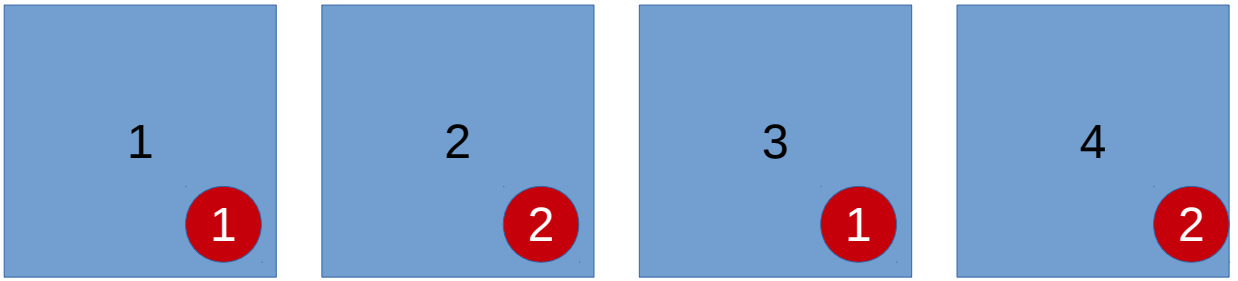
Chunks are stored on a distributed system of data nodes (worker nodes, processing nodes) with replication protecting from data loss due to expected node failures.

HDFS Overview



File Metadata and Access is managed by a dedicated NameNode

File contents is split into pieces Smaller than a predefined constant, e.g., 128 MB



Chunks are stored on a distributed system of data nodes (worker nodes, processing nodes) with replication protecting from data loss due to expected node failures.

HDFS Management

- HDFS is managed via shell commands or a web interface, for example

  hadoop fs -put /<local machine path> /<hdfs path>
  hadoop fs -get /<hdfs path> /<local machime path>

  where hdfs-path is a URL to the name node for the cluster you want to work with.
- HDFS blocks will be used in MapReduce (later) in a way such that Map tasks are assigned to worker nodes, where the data is **already** stored.
- This avoids **delay** from distributing data across the cluster as for example with MPI_Scatter()
- Additionally, it avoids a global storage area network (SAN), which is **difficult for administration** and **expensive**.

MapReduce

MapReduce is a programming abstraction for parallel programs rootet in

- Functional Programming (Map and Reduce)
- Distributed Systems (Avoid Data Distribution before Computation)

based on distributed file systems (Google GFS, Hadoop File System HDFS)

Illustrative Example

- A global search engine:
  - **20+ billion web pages, each about 20k** of interesting data (keywords without stop words)
  - Roughly **400 TB** of data (not that much, right?)
  - But, one computer can read roughly 35 MB /s, that is it takes nearly 4 months to read this data with a single computer
  - With 1.000 hard drives, we have a capacity of roughly 35 GB/s, that is need some more than 10 seconds for reading the data

Illustrative Example (ctd.)

- The problem with 1,000 machines is solvable in small time, but, as already seen, we need
  - Communication and coordination (Naming, Messaging, Routing, …)
  - Recovery from Failures
  - Dashboard / Monitoring / Status
  - Debugging

  The efforts put into these topics are usually done once per project and, therefore, in bad status.

At Google (in 2008)

- A lot of computers
  - 2 CPUs (hyper-threaded)
  - 1-6 locally attached disks (200GB – 2 TB)
  - 4 GB – 16 GB RAM
  - Software:
    - GFS daemon (the distributed file system)
    - Scheduler daemon (for starting and stopping jobs)
    - User tasks (run through the scheduler with data on GFS)

Implications / Assumptions

- Peak Performance less important than Throughput / $
- Errors are normal (especially, as nobody has capacity to take care of them
  - If you have one server, it might work 3 years (1,000)
  - If you hold 1,000 servers, one fails every day.

  Central Question: How can we **make it easy** to write programs for such situations?

MapReduce

In this context, MapReduce is a **quite inflexible** programming model, which, however, allows to move **most of the headache into a library**. As a side effect, all optimizations in this library affect all problems solved using MapReduce.

MapReduce

Usually, data processing for named entities must follow a simple pattern:

- Read an **Entity** into RAM
- **Map:** extract information from each **Entity**
- Shuffle and Sort: Map results will need to be assembled in a globally consistent way implying sort and shuffling.
- **Reduce:** Aggregate, Summarize or Filter results
- Write Results.

MapReduce

Usually, data processing for named entities must follow a simple pattern:

- Read an **Entity** into RAM
- **Map:** extract information from each **Entity**
- Shuffle and Sort: Map results will need to be assembled in a globally consistent way implying sort and shuffling.
- **Reduce:** Aggregate, Summarize or Filter results
- Write Results.

Programming

- Two essential methods:
  - **Map** processes an entity, which is a pair of a key and a value (k,v), and is able to **emit** new key value pairs <k',v'>
  - Shuffle sorts all previously emitted pairs by key and assigns each emitted key to the reduce method:
  - **Reduce** takes a key-multivalue structure and „reduces“ the multiple values to a single value.

  Though these signatures are a restriction, it is surprising, how many problems have a natural formulation in this setting.

WordCount: Hello MapReduce World

A typical example for MapReduce is WordCount.

- Input Entities are files, for example, web pages
- **Map** will read each file word by word and emit a pair ***(W,1)*** for each word ***W***.
- The shuffle phase will sort all words onto the same machine (e.g., into a new entity named by the word).
- **Reduce** will just take all these entries and add them.

Additional Examples

- **Buidling Statistical Language Models**
  - Map: Read each file and emit each 5-word sequence
  - Reduce: Sum up counts.

Additional Examples

- **Reproject Spatial Datasets**
  - Map: For each file, read it and emit geometries after reprojecting to a different spatial reference system
  - Reduce: Identity

A trivial reduce step is a standard technique applicable if each map emits a single result.

Distributed File Systems

- GFS and HDFS are distributed file systems and organized by storing blocks of data on a cluster of computers such that each and every piece of data (e.g., 64 MB) is stored on several different computers.
- HDFS and GFS have files, which are automatically split into pieces and stored over the cluster with replication.
  - **Map** can be assigned to computers, where the data is already on local disk
  - The results of the Map phase go to the **local file system** of a worker and not to the distributed file system

Execution

- One **master** assigns map tasks to many workers
- At time of assignment, data locality is taken into account: Try to assign map task to a location, where the data chunk is already in available.
- Try to have many more map tasks than machines: Better utilization, error recovery, load balancing.

  Typical values (Google 2008: 200,000 map tasks, 5,000 reduce tasks, with 2,000 machines)

Error Tolerance

- Check health of machines via periodic **heartbeats**
- Re-Execute failing Map tasks
- Re-Execute failing Reduce tasks
- Task completion commited through master
- Master employs **checkpointing**

Error Tolerance Refined

- Slow Machines slow down processing
  - Bad disks
  - Bad settings (e.g., disabled CPU cache)
- Idea: As soon as machines get idle, issue already scheduled tasks to them.
- **Dramatically shortens completion time in practice**

Skipping Bad Records

- At least for C++ at Google:
  - If a program exits with an exception such as SEGMENTATION FAULT, the job sequence number is sent to the master via UDP
  - If the master sees a job failing for **several** times on **different** machines, it marks the job as impossible and continues.

A twist on Reduce

Usually, Map tasks emit a lot of key-value pairs, the shuffle phase sorts them and communicates them over the cluster such that **each key goes to a single machine** for reduction.

It is, however, possible to Reduce already before the shuffle phase, if the Reduce is

- Commutative and
- Associative

- **Combiner:** Call Reduce already on each node before shuffle to reduce network consumption