Concurrent and Distributed Systems

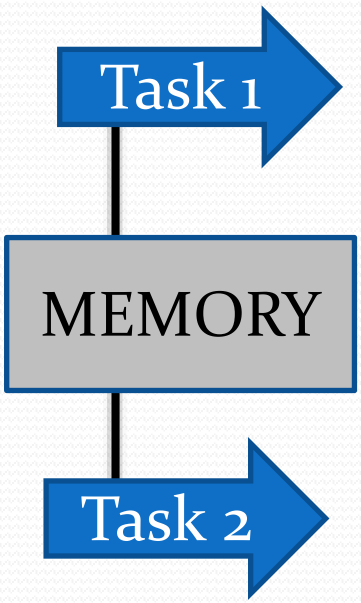
Talk Six

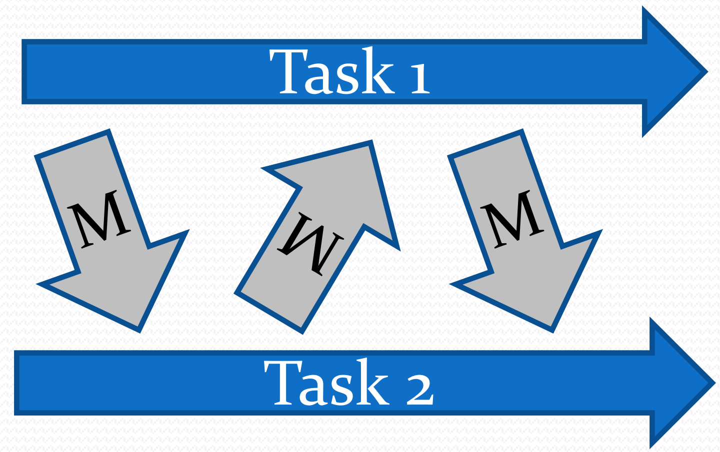
From Shared-Memory Model
to
Passing Messages

Maxim Mozgovoy

(Talk 2): Where is the Data?

Parallel processes can store data in the shared memory,
or in independent memory units and use message passing.





The choice often defined by equipment (multicore vs network)
But sometimes the developer can decide which model to use.

Introducing MPI

The most popular instrument for inter-process communication is MPI (message passing interface).

MPI is a protocol: it explains how processes can communicate, but it is language-independent and hardware-independent.

For any given programming language, a specialized MPI library is needed. For Java, we will use MPJ Express.

MPJ Express documentation:

http://mpj-express.org/documentation.html

Good MPI tutorial:

https://computing.llnl.gov/tutorials/mpi/

More on MPI

- An established solution
(ver. 1.0 = 1994, ver. 2.0 = 1997, ver. 3.0 = 2012, ver. 4.0 = *draft*)
- Supports point-to-point and collective communication.
- Goals: performance, scalability, portability.
- A de facto standard for fast distributed computations.
- Especially popular in scientific programming.
- Implemented for most popular programming languages, including C, Fortran, Java, Python, Perl, Ruby, .NET CLR.
- Can be run on multiprocessor computers and in networks (computational clusters).
- Will also work on single-processor machines
(useful for debugging).

MPI Philosophy

MPI program works as follows:

1. You write a procedure.
2. You ask MPI environment to execute N procedures in parallel.
3. MPI runs N copies of your code as independent processes.
4. Each process has a unique ID and can communicate with other processes.

Note: there is no shared memory!

So you don't have to care about technical details (how it works), you just concentrate on writing the algorithm!

Note that all processes in the program are identical;

the only difference is in their IDs.

"Hello, World!" Example (1)

```
import mpi.*;
public class w06_helloWorld {
  public static void main(String args[])
                                    throws Exception {
      MPI.Init(args);
      int ID = MPI.COMM_WORLD.Rank(); // proc ID
      System.out.println("I am " + ID);
      MPI.Finalize();
  }
}
```

```
C:\HelloMPI> mpjrun -np 3 w06_helloWorld
I am 0
I am 2
I am 1
```

“Hello, World!” Example (2)

`mpjrun -np <N> <MAIN-CLASS>`

Runs `N` copies of the program, defined in the `MAIN-CLASS`.

It is possible to supply command-line args to `MAIN-CLASS`:

`mpjrun -np <N> <MAIN-CLASS> arg1 arg2 …`

Note: inside `MAIN-CLASS` these arguments are stored as `args[3]`, `args[4]`,… (instead of `args[0]`, `args[1]`,…)

Setting Up MPI (1)

1. Download MPJ (mpj-v0_44.zip) and unzip it into your home dir (run java PrintUserHome to find it out if unsure).

2. Update environment variables MPJ_HOME and PATH (see the next slide).

3. Restart your session (logout, then login again).

Setting Up MPI (2)

Environment variables update:

For Solaris (default shell csh), run:

```
echo 'setenv MPJ_HOME $HOME/mpj-v0_44' >> ~/.cshrc
echo 'setenv PATH $MPJ_HOME/bin:{$PATH}' >> ~/.cshrc
```

For MS Windows, run:

```
SETX MPJ_HOME %USERPROFILE%\mpj-v0_44
SETX PATH "%PATH%";%USERPROFILE%\mpj-v0_44\bin
```

For Ubuntu and Mac (default shell bash) run:

```
echo 'export MPJ_HOME=$HOME/mpj-v0_44' >>
                                    ~/.bash_profile
echo 'export PATH=$PATH:$MPJ_HOME/bin' >>
                                    ~/.bash_profile
```

Setting Up MPI (3)

Try to compile and run “Hello, World!” example (prev. slides) with the mpj.jar package:

```
// *nix systems:
javac -cp .:$MPJ_HOME/lib/mpj.jar w06_helloWorld.java
mpjrun.sh -np 2 w06_helloWorld
```

```
// MS Windows
javac -cp .;%MPJ_HOME%/lib/mpj.jar w06_helloWorld.java
mpjrun -np 2 w06_helloWorld
```

Basic MPI Functions

```
MPI.Init(args); // to be called before any other MPI
                // function; pass it main() arguments

MPI.Finalize(); // to be called at the end

MPI.COMM_WORLD.Rank(); // get the ID of the
                       // current process (0…P-1)

MPI.COMM_WORLD.Size(); // get the total
                       // number of processes
```

Note: these are MPJ Express versions;
MPI implementations in different libraries
may slightly vary.

Send / Receive Functions

```
// send Nitems of type itemtype, taken from array arr
// at offset to the process destID;
// label message with the integer tag
MPI.COMM_WORLD.Send(arr, offset, Nitems,
                    itemtype, destID, tag);


// wait for the incoming message, then receive
// Nitems of type itemtype labeled with the
// integer tag from the process srcID; store them in
// the array arr at the given offset
MPI.COMM_WORLD.Recv(arr, offset, Nitems,
                    itemtype, srcID, tag);
```

Send / Receive Functions: Notes

```
MPI.COMM_WORLD.Send(arr, offset, Nitems,
                    itemtype, destID, tag);
MPI.COMM_WORLD.Recv(arr, offset, Nitems,
                    itemtype, srcID, tag);
```

Common values for the itemtype argument:

```
MPI.BOOLEAN, MPI.BYTE, MPI.DOUBLE, MPI.INT
```

Special value for the srcID argument:

```
MPI.ANY_SOURCE // Receive from any process ID
```

Special value for the tag argument of Recv():

```
MPI.ANY_TAG    // Receive messages with any tag
```

Send / Receive Example (1)

Pseudocode:

```
IF ProcID == 0
    String s = "Hello"
    SEND length(s) to the ProcID 1
    SEND s to the ProcID 1

ELSE // ProcID == 1
    RECEIVE integer N from the ProcID 0
    Allocate array of characters S[N]
    RECEIVE N chars from the ProcID 0, store in S
    Display S
```

Send / Receive Example (2)

```
import mpi.*;
public class w06_sendRec {
  public static void main(String args[])
                                        throws Exception {
    MPI.Init(args);
    if(MPI.COMM_WORLD.Rank() == 0) {
      String msg = "Hello!";
      MPI.COMM_WORLD.Send(new int[]{ msg.length() },
                                   0, 1, MPI.INT, 1, 0);
      MPI.COMM_WORLD.Send(msg.getBytes(), 0,
                           msg.length(), MPI.BYTE, 1, 0);
    }
    …
```

Send / Receive Example (3)

```
    else {
        int buff[] = new int[1];
        MPI.COMM_WORLD.Recv(buff, 0, 1, MPI.INT,
                                   MPI.ANY_SOURCE, MPI.ANY_TAG);
        int length = buff[0];
        byte msg[] = new byte[length];
        MPI.COMM_WORLD.Recv(msg, 0, length, MPI.BYTE,
                                   MPI.ANY_SOURCE, MPI.ANY_TAG);
        System.out.println(new String(msg));
    }
    MPI.Finalize();
  }
}
```

```
mpjrun -np 2 w06_sendRec
Hello!
```

Reduce() Operation

```
// Reduce() is a collective operation that involves
// ALL the processes in the program

// combine data, taken from each process,
// using operation op,
// then send it to process rootID
MPI.COMM_WORLD.Reduce(sendbuf, sendoffset,
                      recvbuf, recvoffset,
                      count, datatype, op, rootID)


// Operation can be programmed by the user, but there
// are some built-in operations, too
// (e.g., MPI.MAX, MPI.MIN, MPI.SUM, MPI.PROD…)
```

Count $1^2 + 2^2 + \ldots + P^2$ Example (1)

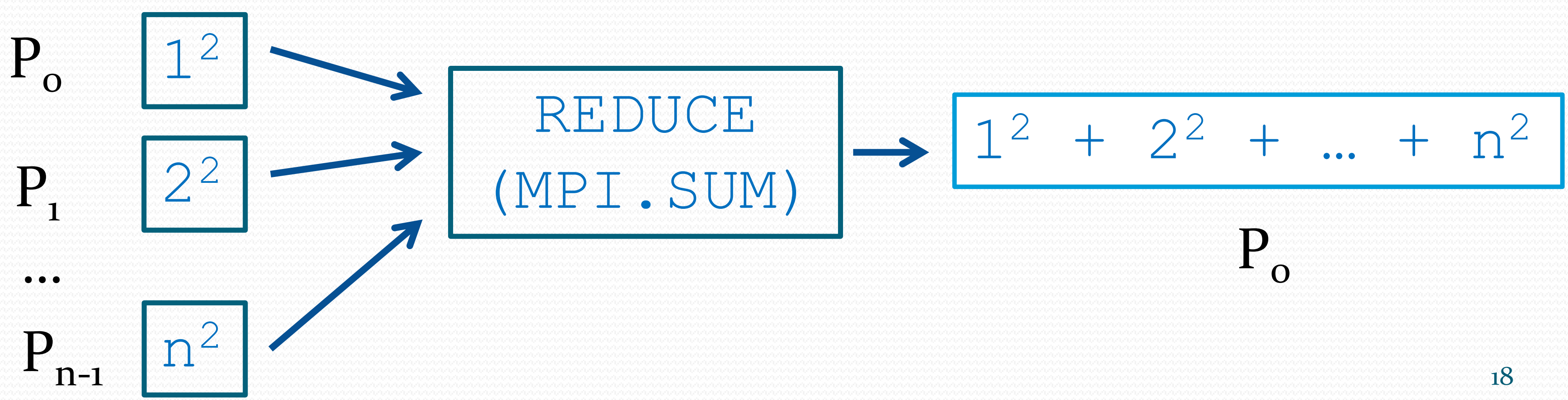
Suppose that each process has:

```
int recv[1];
int send[] { (ProcID + 1) * (ProcID + 1) };
```

And each process executes:

```
Reduce(send, 0, recv, 0, 1, MPI.INT, MPI.SUM, 0);
```

After this operation the root process (ID = 0) will receive the sum (`MPI.SUM`) of `send[0]` elements:



Count $1^2 + 2^2 + \ldots + P^2$ Example (2)

```
import mpi.*;
public class w06_count {
  public static void main(String args[])
                                  throws Exception{
      MPI.Init(args);
      int recv[] = new int[1];
      int N = MPI.COMM_WORLD.Rank() + 1;
      MPI.COMM_WORLD.Reduce(new int[] { N * N }, 0,
                  recv, 0, 1, MPI.INT, MPI.SUM, 0);
      if(MPI.COMM_WORLD.Rank() == 0)
            System.out.println("Sum: " + recv[0]);
      MPI.Finalize();
  }
}
```

```
mpjrun -np 3 w06_count
Sum: 14
```

Other Collective Operations (1)

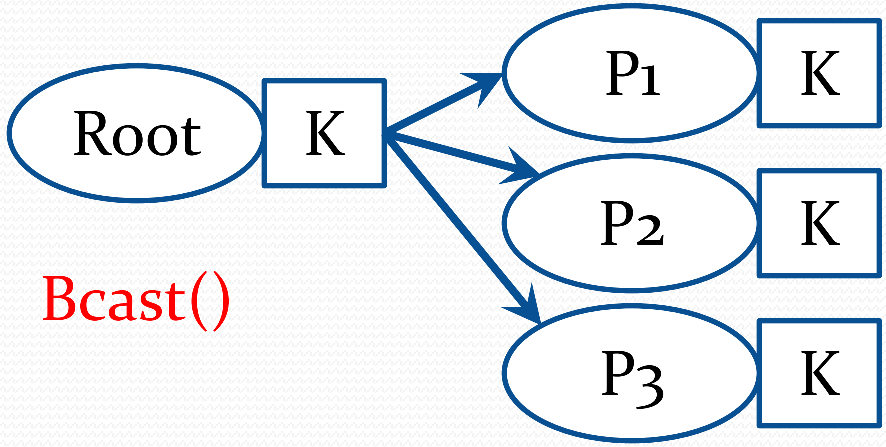
```
// send count elements of arr (beginning at offset)
// from rootID to all other processes
// (send operation for rootID, receive for others)
// after Bcast() everybody has the same elements
// in the arr
MPI.COMM_WORLD.Bcast(arr, offset, count,
                                datatype, rootID)
```

Other Collective Operations (2)

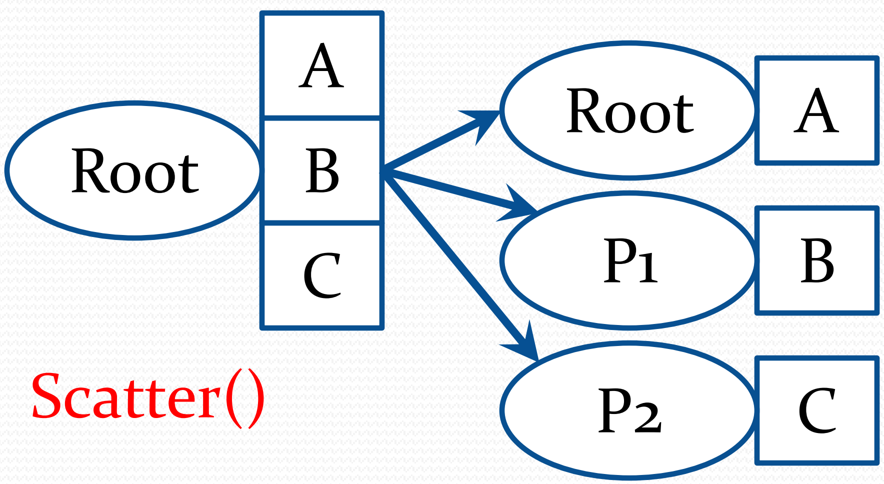
```
// send a data block from each process to rootID
MPI.COMM_WORLD.Gather(sndbuf, sndoffs, sndcount,
                      sndtype, rcvbuf, rcvoffs,
                      rcvcount, rcvtype, rootID)

// send a data block to each process from rootID
MPI.COMM_WORLD.Scatter(sndbuf, sndoffs, sndcount,
                       sndtype, rcvbuf, rcvoffs,
                       rcvcount, rcvtype, rootID)
```

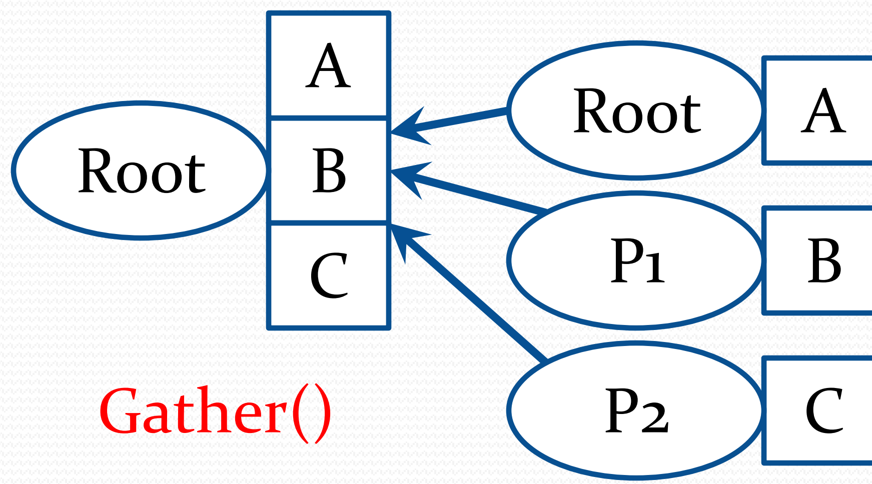
Bcast() / Gather() / Scatter()



Bcast()



Scatter()



Gather()

Bcast() and Gather() Example

```
// Send a random seed from root (0) to each process.
// each process makes a new seed (seed + rank_id)
// then returns a random number

long seed[] = new long[]{System.currentTimeMillis()};
MPI.COMM_WORLD.Bcast(seed, 0, 1, MPI.LONG, 0);
long nseed = seed[0] + MPI.COMM_WORLD.Rank();
java.util.Random r = new java.util.Random(nseed);
long rnd[] = new long[] { r.nextInt(100) };
long rcvbuf[] = new long[MPI.COMM_WORLD.Size()];
MPI.COMM_WORLD.Gather(rnd, 0, 1, MPI.LONG,
                      rcvbuf, 0, 1, MPI.LONG, 0);
if(MPI.COMM_WORLD.Rank() == 0)
    for(int i = 0; i < rcvbuf.length; ++i)
        System.out.println(rcvbuf[i]);
```

Collective Operations: Notes

Note that all the processes should execute the same collective operation. For example, consider a call:

```
MPI.COMM_WORLD.Bcast(arr, 0, 1, MPI.INT, root);
```

The MPI library itself performs different actions for the `root` process and all other processes.

When `Bcast()` is executed, the `root` process actually sends the data, while all other processes receive it.

New Tools, Old Problems

Note that our old problems with synchronization are still relevant for MPI.

In particular, note that all the message-passing operations postpone the processes until the data has been transferred.

So instead of semaphores, we now use Send / Receive.

This means, we still should synchronize processes.

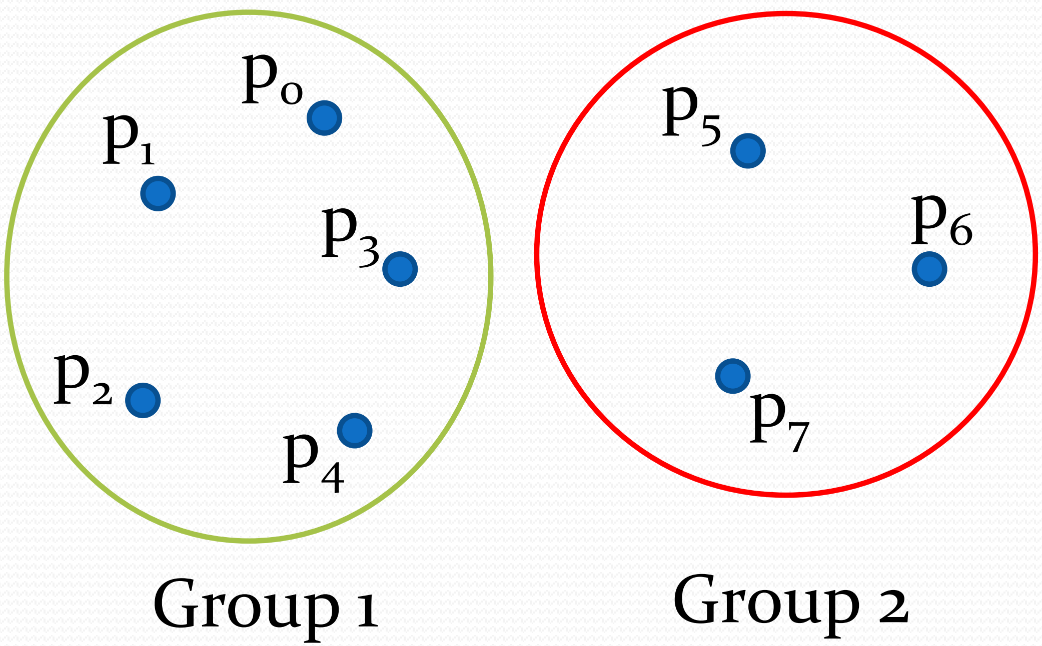
And we still should avoid deadlocks:

- P1 waits for a message from P2
- P2 waits for a message from P1

Communicators (1)

Collective operations such as Bcast() or Reduce() presume that all the processes in our program participate in communication.

For more complex applications, we might need to separate processes into groups. Groups can communicate with each other, and each group can run their own collective operations.



Communicators (2)

Communication is performed by using communicator objects. So far, we've used one such communicator: `MPI.COMM_WORLD`. It includes all the processes in the system and always can be used to send a message from any process to any other process.

We can also create local communicators attached to process groups. One possible way to do it is to use `Split()` function:

```
class Intracomm {
    …
    public Intracomm Split(int color, int key);
}
```

Communicators (3)

`Split()` should be called by each process independently. By calling `Split()` a process tells the system to which group it wants to belong:

```
Intracomm Split(int color, int key);
```

A `color` is an arbitrary integer ID of a new group where a process wants to belong (it can be `MPI.UNDEFINED` if it does not want to belong anywhere).

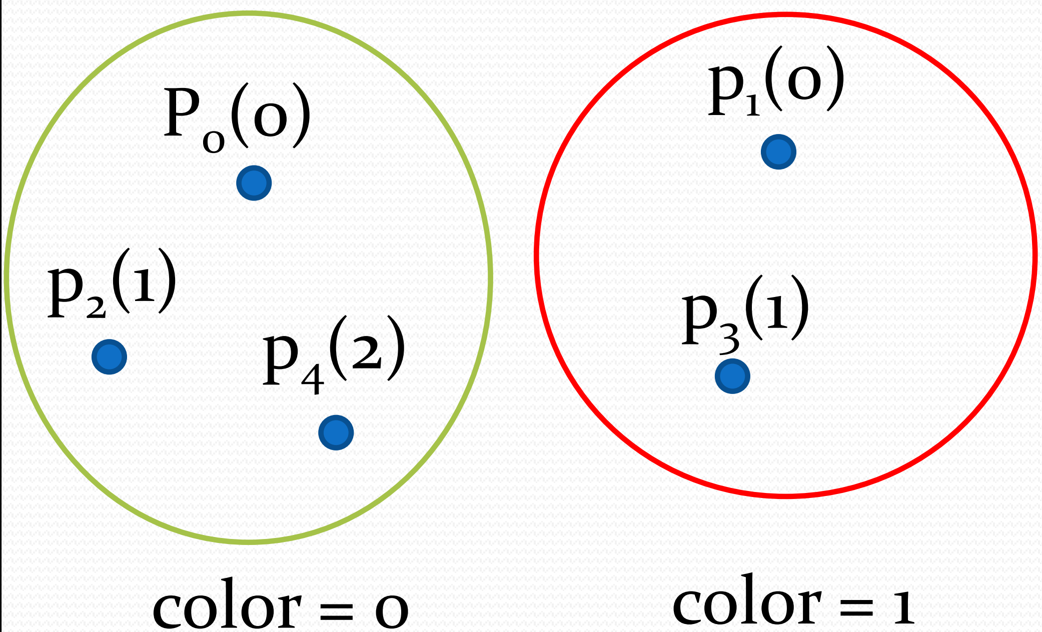
A `key` controls how process IDs are assigned within a group. A process with the lowest key gets the ID=0, the next process gets the ID=1, and so on.

Communicators (4)

Consider the following fragment:

```
int pID = MPI.COMM_WORLD.Rank();
int color = pID % 2; // make two groups
Intracomm c = MPI.COMM_WORLD.Split(color, pID);
```

If we run it for 5 processes, we will obtain two groups:



Note: you can call `c.Split()` to split these groups further!

Communicators (5)

Now we can try some in-group collective operations:

```
MPI.Init(args);
int pID = MPI.COMM_WORLD.Rank();
Intracomm c = MPI.COMM_WORLD.Split(pID % 2, pID);
int recv[] = new int[1];
c.Reduce(new int[]{c.Rank()}, 0,
         recv, 0, 1, MPI.INT, MPI.SUM, 0);
if(c.Rank() == 0)
   System.out.println("I am " + pID +
                      ", group sum is " + recv[0]);
MPI.Finalize();
```

```
mpjrun -np 5 hello_comm
I am 1, group sum is 1
I am 0, group sum is 3
```

Conclusions

- MPI is a mechanism that helps us to write parallel programs.
- By using MPI, we can decide what to do, and MPI decides how to do it.
- MPI program is a collection of identical processes (N copies of the same process), having unique IDs (ranks).
- These processes can communicate by sending messages.
- There are also collective operations available (work on certain data together).
- MPI parallelism is abstract: MPI will use all your physical processors to run the threads, but it can also run several processes on one processor.