Parallel, distributed models and programming paradigms

Big Data, Machine Learning and Social Network Analysis Mini-Workshop and Tutorial

Camille Coti

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, France

Dec. 16th 2014

Outline

Distributed systems

A **distributed systems** is a set of processes called $p_0, p_1, ..., p_{n-1}$ linked together by a **communication system**

- Every process is executing a program
- Every process has its own control system and its own instruction stream
  $\rightarrow$ SIMD = not a distributed system
- Processes communicate with each other through the communication system, which is not necessarily a point-to-point network

Configuration of the system

- A configuration is the set of the states of the processes at a given moment
- If e_k = state of the process k, a configuration C is $\bigcup_{k=0}^{n-1} e_k$

Message-passing communications

Processes themselves are **state machines**. The state of a process is changed by **events**, which can be:

- Internal: defined by the algorithm executed by the process;
- Reception: message arriving from the communication system;
- Sending: message sent on the communication system.

Message passing:

- Processes execute **sending and reception primitives**:
  - $send(buffer, destination)$
  - $receive(buffer, source)$
- Every sending **must match** a reception (and *vice versa*)
- **Asynchronous**: the communication delay is finite but arbitrary.

Shared memory communications

State-reading model:

- Each process has a set of neighboring processes
- Each process can read the (full) **state** of its neighbors
- NB: each neighbor of process p will read the same state of p.

link-register model:

- There exist **memory registers** between two (or more) processes
- The processes that access a register r can write (primitive: $write(buffer, r)$) or read (primitive: $read(buffer, r)$) **atomically** into or from this register.

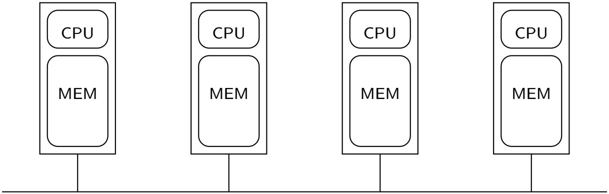
Distributed memory

In practice :

- A **set of processes**
- Each process has its **own memory**
- They are connected by a peer-to-peer interconnection network: a network (Ethernet, IB, Myrinet, Internet...) or the system bus.

The programmer is in charge with **data locality**

- **Explicit** data movements between processes
- If process P_i needs some data which is in the memory of process P_j, then the programmer must *explicitly* move it from P_j to P_i .

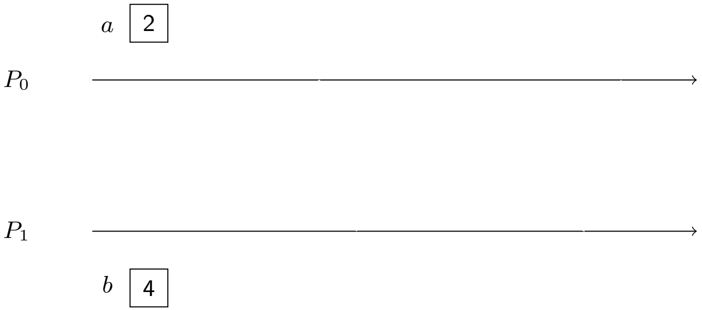


Two-sided communications

Two-sided communications

- Primitives $send/recv$
- A $send$ primitive **must** match a $recv$ primitive (and vice versa)

Consequence: when some data is moved between two processes, both processes take an **active** part of the data movement.

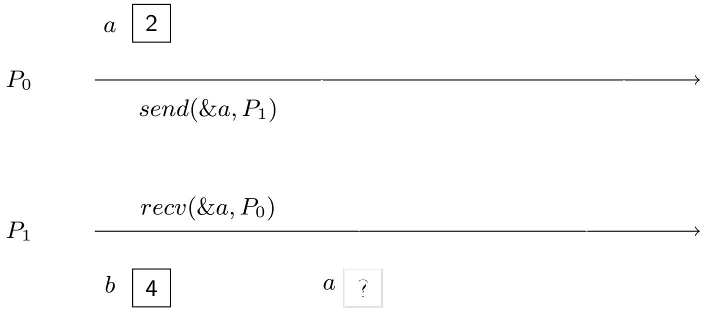


Two-sided communications

Two-sided communications

- Primitives $send/recv$
- A $send$ primitive **must** match a $recv$ primitive (and vice versa)

Consequence: when some data is moved between two processes, both processes take an **active** part of the data movement.

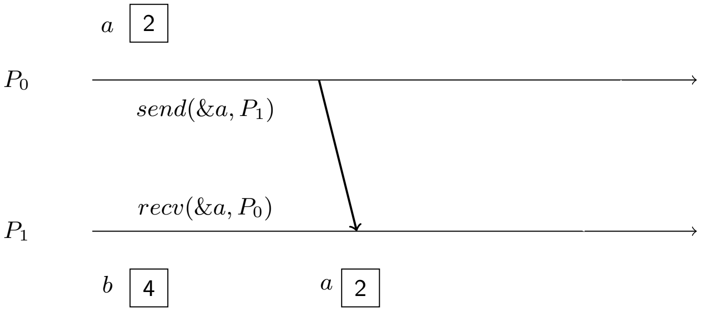


Two-sided communications

Two-sided communications

- Primitives $send/recv$
- A $send$ primitive **must** match a $recv$ primitive (and vice versa)

Consequence: when some data is moved between two processes, both processes take an **active** part of the data movement.

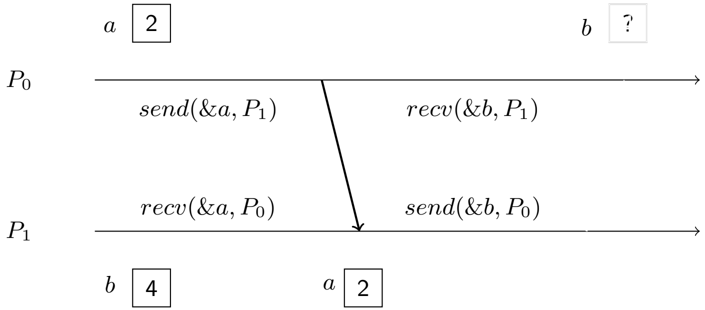


Two-sided communications

Two-sided communications

- Primitives $send/recv$
- A $send$ primitive **must** match a $recv$ primitive (and vice versa)

Consequence: when some data is moved between two processes, both processes take an **active** part of the data movement.

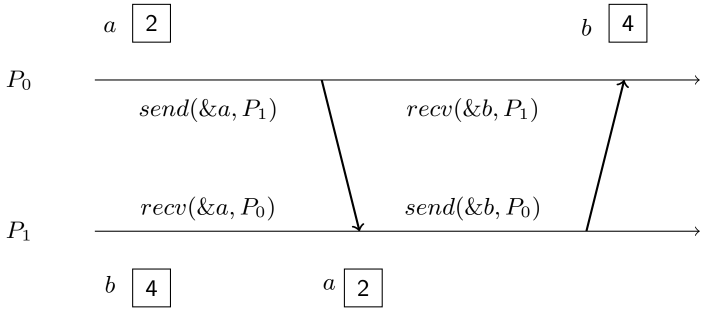


Two-sided communications

Two-sided communications

- Primitives *send*/*recv*
- A *send* primitive **must** match a *recv* primitive (and vice versa)

Consequence: when some data is moved between two processes, both processes take an **active** part of the data movement.



Example

Example of a library for programming parallel programs on distributed memory using two-sided communications: **MPI**

- *De facto* standard for parallel programming
- Complete control of the data locality ("*assembly language of parallel programming*")
- Portable
- Powerful: can be used to write programs in other models
- Point-to-point but also collective communications

Pros:

- Complete control of the data locality
- Very good performance

Cons:

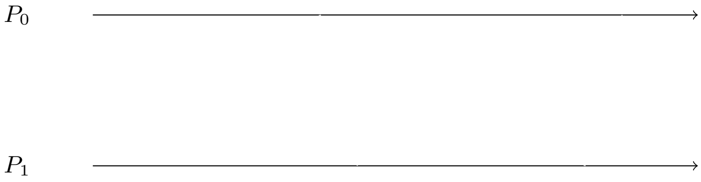
- Both processes (source and destination) must cooperate
- Strongly synchronous

MPI: Example

Example: ping-pong

- A process with rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

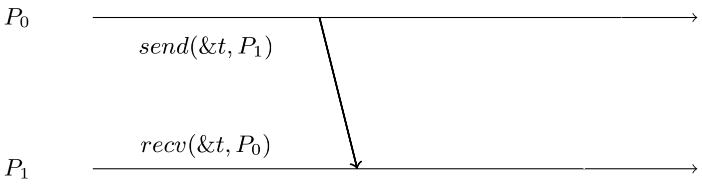


MPI: Example

Example: ping-pong

- A process with rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

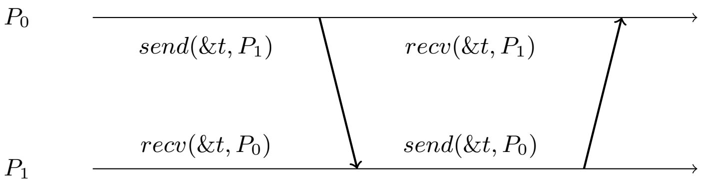


MPI: Example

Example: ping-pong

- A process with rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

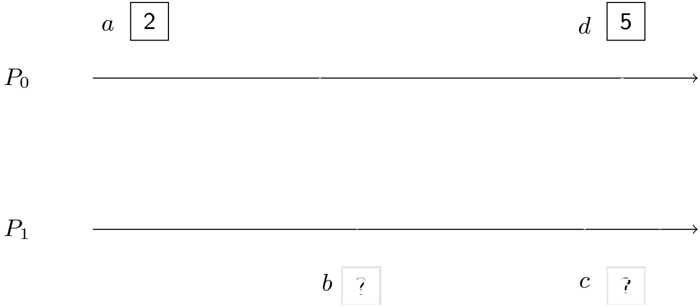


One-sided communications

One-sided communications

- Primitives put/get
- RDMA model : Remote Direct Memory Access
- A process can read/write in another process's memory
- In practice: can be done by RDMA network interface cards (InfiniBand, Myrinet...)

Only one process is taking part of the communication.

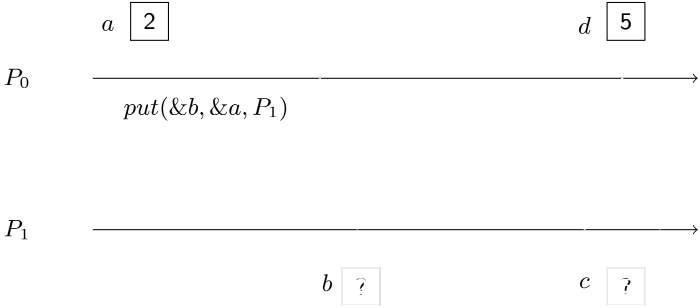


One-sided communications

One-sided communications

- Primitives put/get
- RDMA model : Remote Direct Memory Access
- A process can read/write in another process's memory
- In practice: can be done by RDMA network interface cards (InfiniBand, Myrinet...)

Only one process is taking part of the communication.

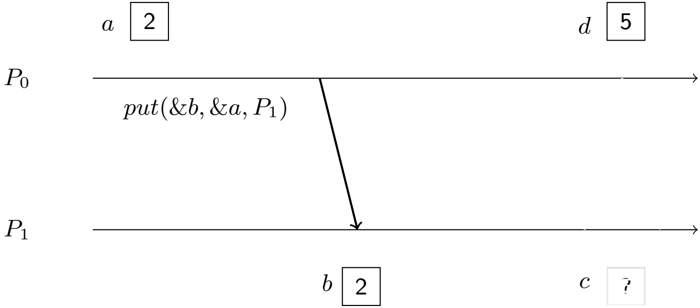


One-sided communications

One-sided communications

- Primitives *put*/*get*
- RDMA model : Remote Direct Memory Access
- A process can read/write in another process's memory
- In practice: can be done by RDMA network interface cards (InfiniBand, Myrinet...)

Only one process is taking part of the communication.

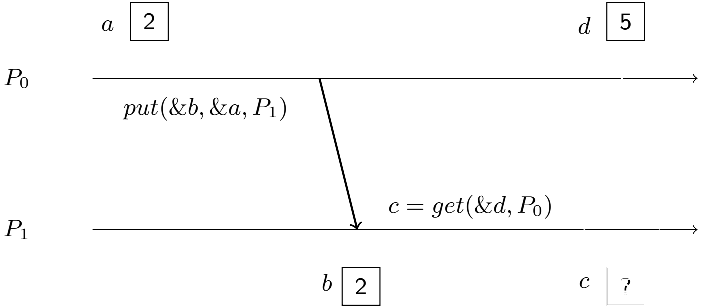


One-sided communications

One-sided communications

- Primitives put/get
- RDMA model : Remote Direct Memory Access
- A process can read/write in another process's memory
- In practice: can be done by RDMA network interface cards (InfiniBand, Myrinet...)

Only one process is taking part of the communication.

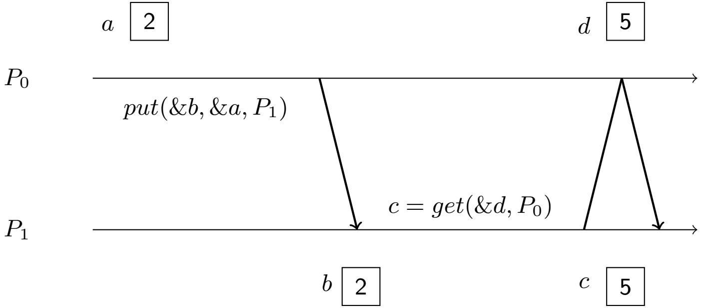


One-sided communications

One-sided communications

- Primitives put/get
- RDMA model : Remote Direct Memory Access
- A process can read/write in another process's memory
- In practice: can be done by RDMA network interface cards (InfiniBand, Myrinet...)

Only one process is taking part of the communication.



Examples

Examples :

- One-sided communications of **MPI**
- Put/get functions of **UPC**
- **OpenSHMEM**

OpenSHMEM

- Descendant of Cray's SHMEM, GI SHMEM... from the 90s
- Recent standardization effort, due to needs coming from current architectures.

Pros:

- Very fast communications
- Particularly well adapted to current hardware architectures
- Does not require both processes to be ready to communicate

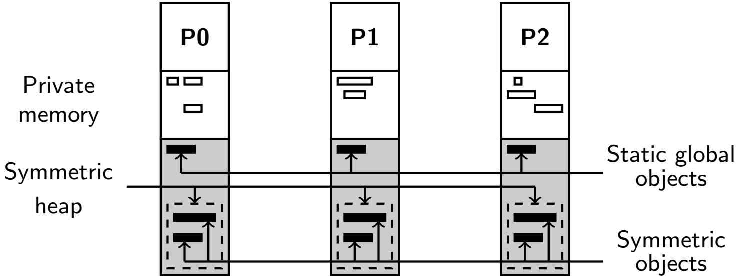
Cons:

- Sensitive model, risk of race conditions
- Necessitates symmetric process memories

OpenSHMEM

Memory model: **symmetric heap**

- Private memory vs shared memory (heap)
- Memory allocation in the shared heap is a *collective communication*



OpenSHMEM : Example

Allocation in the shared heap :

- shmalloc function
- Warning: collective

Data movements:

- Fonctions shmem_*_put, shmem_*_get
- One function for each data type

```
short* ptr = (short*)shmalloc( 10 * sizeof( short ) );
if (_my_pe() == 0) {
    shmem_long_put( ptr, source, 10, 1 );
}
```

Global Address Space

Concept of **global address space** :

- Program distributed memory just like shared memory
- Participation from the **compiler**
- The union of the distributed memories is seen by the programmer **as a shared memory**

In practice:

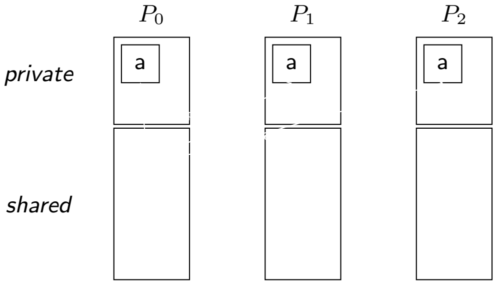
- The programmer declares the **visibility** of his/her variables: private (by default) or **shared**
- Arrays: The programmer declares the size of the blocks that will be placed on each process
- The compiler is in charge with:
  - **Distributing the shared variables** in the memory of the processes
  - **Translating remote accesses** $(a = b)$ into communications

Issues related to the fact that the memory is distributed **are not seen** by the programmer.

Examples

PGAS languages:

- Unified Parallel C (UPC), Titanium, CoArray Fortran



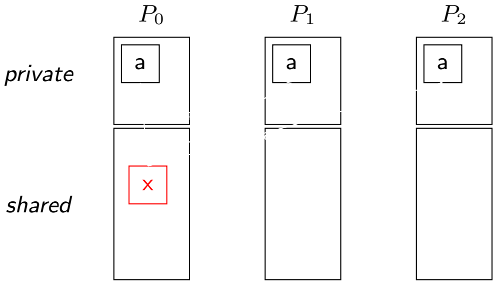
int a;

shared int x;

Examples

PGAS languages:

- Unified Parallel C (UPC), Titanium, CoArray Fortran



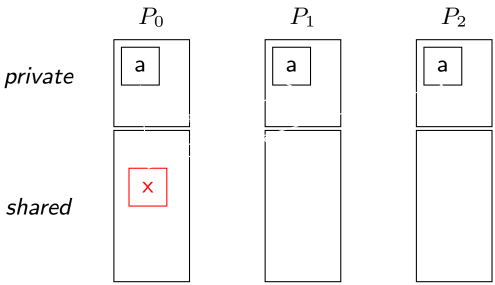
int a;

shared int x;

Examples

PGAS languages:

- Unified Parallel C (UPC), Titanium, CoArray Fortran



int a;

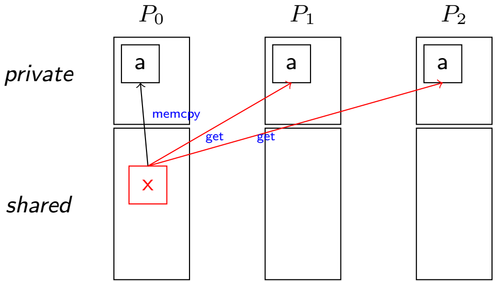
shared int x;

a = x;

Examples

PGAS languages:

- Unified Parallel C (UPC), Titanium, CoArray Fortran



int a;

shared int x;

a = x;

UPC: Example

Example :

- A variable x is shared, and therefore accessible from all the processes
  - The compiler will place it in the memory of a process of its choice.
- Process 0 (called thread in UPC terminology) initializes it to 42.
- A global barrier makes sure that all the processes have reached this point of the program.
- All the processes read the value of x and put it into a private variable of their own.
  - The compiler generates inter-process network communications (in all likelihood *get*)

```
shared int x;
int a;
if( 0 == MYTHREAD ) {
    x = 42;
}
upc_barrier;
a = x;
```

Bag of tasks

What is a **bag of tasks** ?

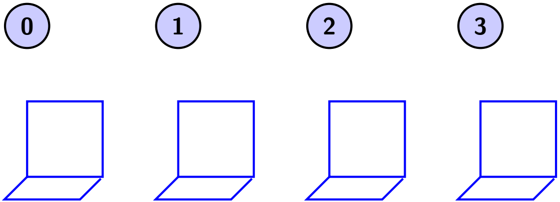
- A set of computations that must be performed
- **Independent** from each other

These computations can be done in **parallel** from each other

→ A bag of tasks can be parallelized *extremely* well!

No communication between the processes that are running the tasks

Tasks



Results

Bag of tasks

What is a **bag of tasks** ?

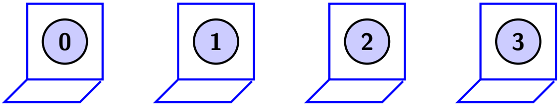
- A set of computations that must be performed
- **Independent** from each other

These computations can be done in **parallel** from each other

→ A bag of tasks can be parallelized *extremely* well!

No communication between the processes that are running the tasks

Tasks



Results

Bag of tasks

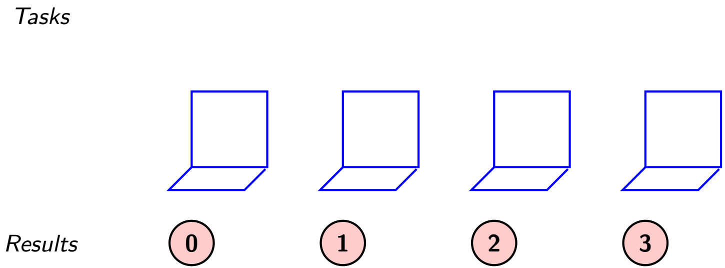
What is a **bag of tasks** ?

- A set of computations that must be performed
- **Independent** from each other

These computations can be done in **parallel** from each other

→ A bag of tasks can be parallelized *extremely* well!

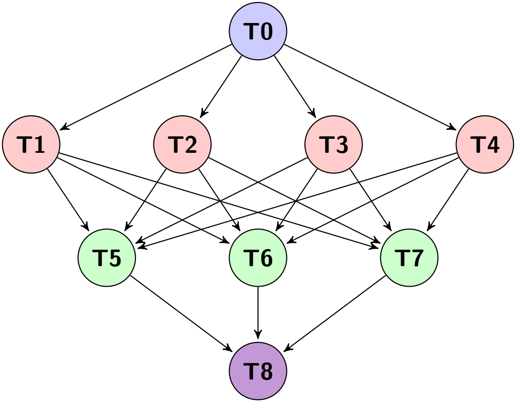
No communication between the processes that are running the tasks



Bag of tasks

A computation can be made of several phases:

- Relations can be defined between those tasks
- Represented by a DAG



Examples

There are many ways to implement a bag of tasks!

- **MPI** → a master distributed the work to workers and gets the results.
- **HTCondor** → designed specifically for it, schedules DAGs on a pool of nodes
- **MapReduce** → a bit particular: the *map* operation computes the tasks in parallel, the *reduce* operation can be used to gather the results

Simple because there is **no communication** between the processes

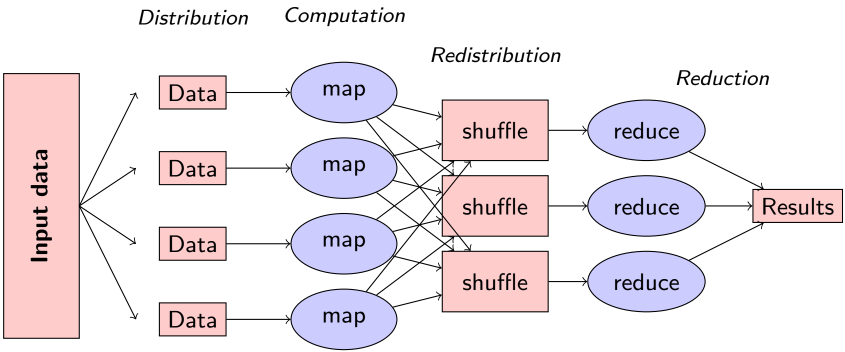
- Requires a coordinator that schedules the tasks
- ... and gather the results at then end.

The only communications are between this coordinator and the computing processes, then between the computing processes and the coordinator.

Regarding MapReduce

Goal of **MapReduce** :

- Process **large volumes** of data
- Not necessarily "big" parallel computing
- Oriented for *big data*, *data mining*...
- Important communication phase between processes during the *reduce* operation



Conclusion

Memory models :

- Distributed → explicit message-passing communications (MPI, OpenSHMEM)
- Shared distributed → global address space, help from the compiler (PGAS languages)

Communication patterns :

- Both processes cooperate → two-sided communications (MPI)
- Remote access → one-sided communications (OpenSHMEM, UPC)
- No inter-process communication → bag of tasks

Problem's data :

- Regular → OpenSHMEM
- Irregular → MPI, UPC
- Very big → MapReduce