Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`
Then, this code is running 4 times in parallel !

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])

{ int n;    double result;
  int my_rank, num_procs;
```

application-related data

MPI-related data

```c
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Now, each process knows who it is:
number *my_rank* out of *num_procs* processes

```c
  if (my_rank == 0)
  { printf("Enter the number of elements (n): \n");
    scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

reading the application data *n* from stdin only by process 0

process 0 is sender, all other processes are receivers

broadcasting the content of variable *n* in process 0 into variables *n* in all other processes

```c
  result = 1.0 * my_rank * n;
  printf("I am process %i out of %i handling the %ith part of n=%i elements,result=%f\n",
                    my_rank,   num_procs,          my_rank,      n,                   result);
```

doing some **application work** in each process

```c
  if (my_rank != 0)

  { MPI_Send(&result,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
  }
  else
  { int rank;
   printf("I'm proc 0: My own result is %f \n",result);
   for (rank=1; rank<num_procs; rank++)
    {
       MPI_Recv(&result,1,MPI_DOUBLE,rank,99,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("I'm proc 0: received result of
         process %i is %f \n", rank, result);
    }
  }
  MPI_Finalize();
}
```

send to process 0

sending some results from all processes (except 0) to process 0

Process 0: receiving all these messages and, e.g., printing them

receiving the message from process *rank*

```
Enter the number of elements (n): 100

I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0

I'm proc 0: received result of process 1 is 100.0
I'm proc 0: received result of process 2 is 200.0
I'm proc 0: received result of process 3 is 300.0
```

Jun Li, Department of Computer Science, CUNY Queens College

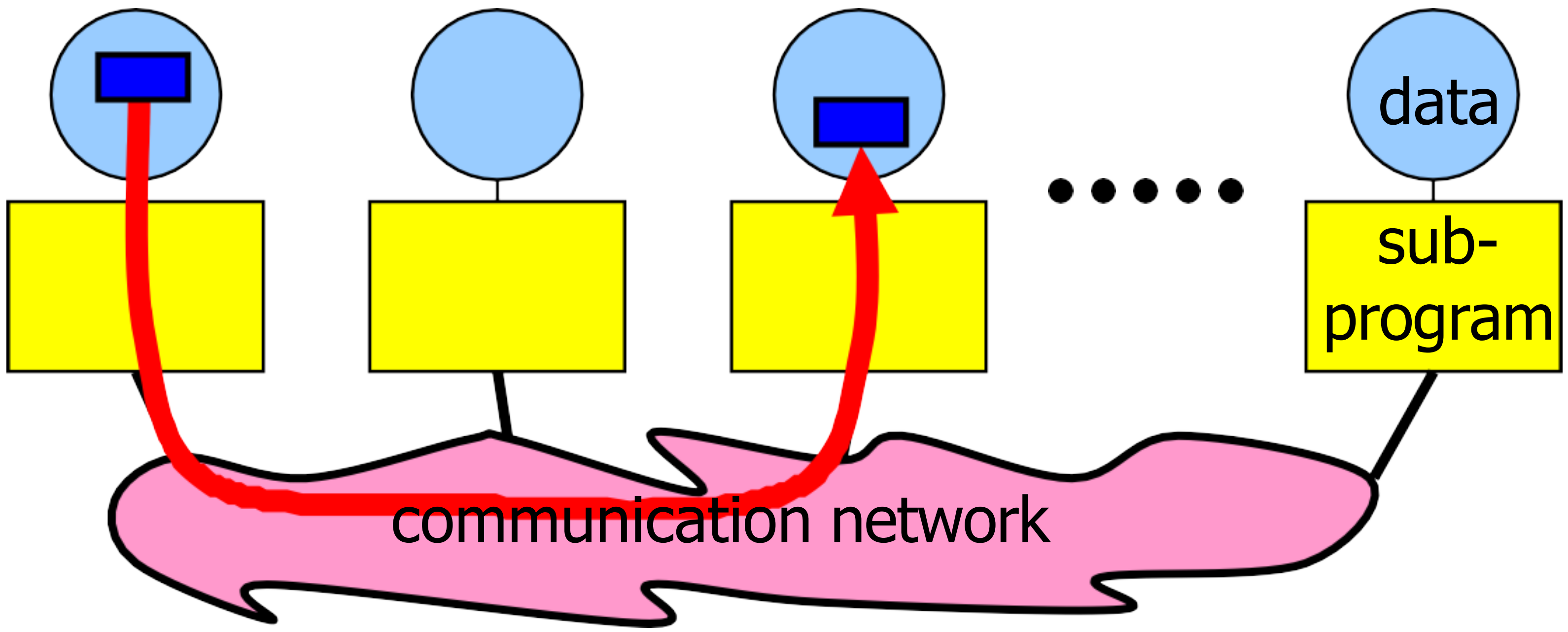
Run mpiexec -n 4 python first-example.py

```python
from mpi4py import MPI

# application-related data
n = None
result = None

comm_world = MPI.COMM_WORLD
# MPI-related data
my_rank = comm_world.Get_rank() # or my_rank = MPI.COMM_WORLD.Get_rank()
num_procs = comm_world.Get_size() # or ditto ...

if (my_rank == 0):
   # reading the application data "n" from stdin only by process 0:
   n = int(input("Enter the number of elements (n): "))

# broadcasting the content of variable "n" in process 0
# into variables "n" in all other processes:
n = comm_world.bcast(n, root=0)

# doing some application work in each process, e.g.:
result = 1.0 * my_rank * n
print(f"I am process {my_rank} out of {num_procs} handling the {my_rank}ith part of n={n} elements, result={result}")

if (my_rank != 0):
   # sending some results from all processes (except 0) to process 0:
   comm_world.send(result, dest=0, tag=99)
else:
   # receiving all these messages and, e.g., printing them
   rank = None
   print(f"I'm proc 0: My own result is {result}")
   for rank in range(1,num_procs):
      result = comm_world.recv(source=rank, tag=99)
      print(f"I'm proc 0: received result of process {rank} is {result}")
```

Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - mail box
  - phone line
  - fax machine
  - etc.
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:

– sending process	– receiving process
– source location	– destination location
– source data type	– destination data type
– source data size	– destination buffer size

i.e., the ranks

basic or **derived**
datatypes

Addressing

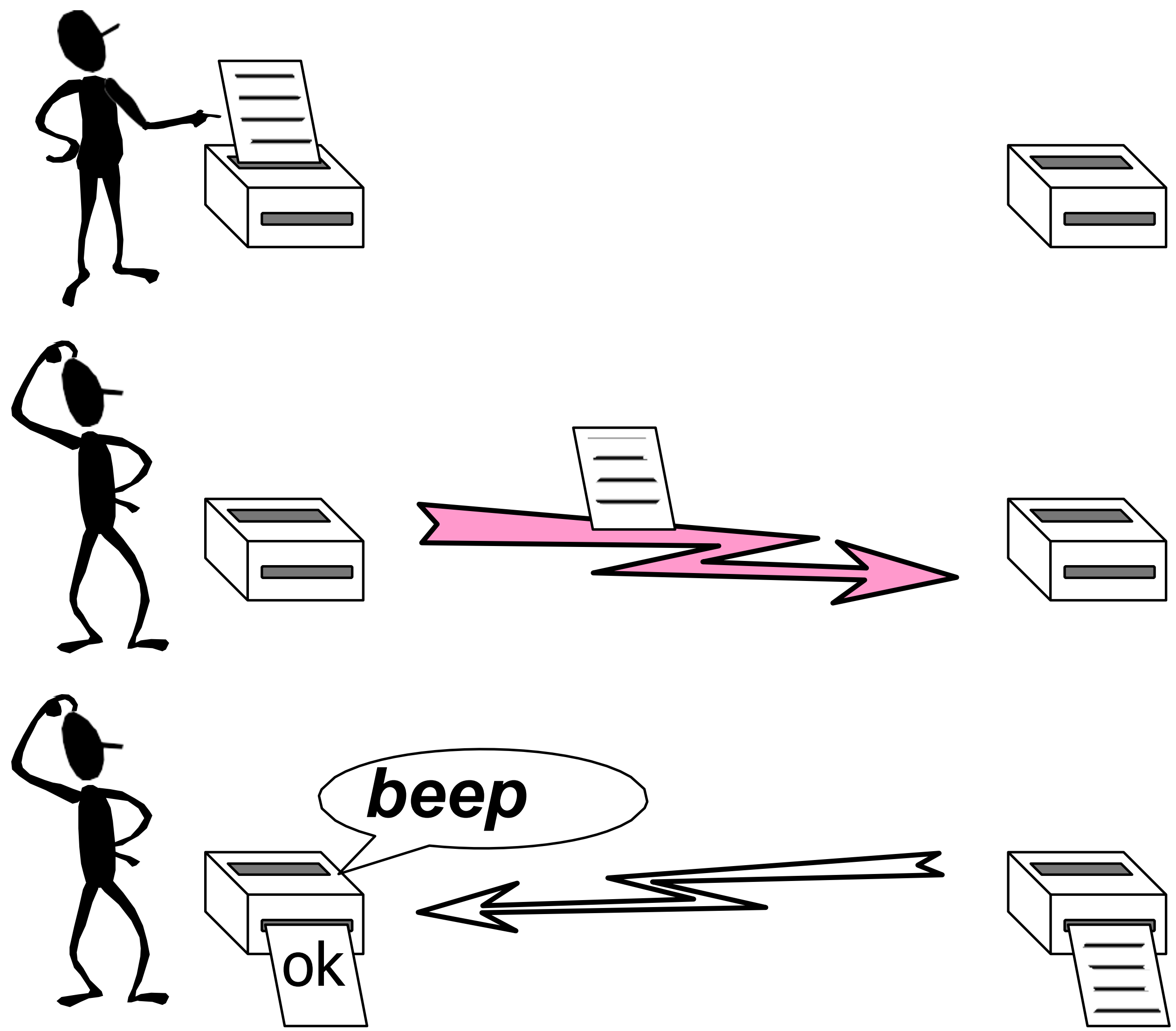
- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

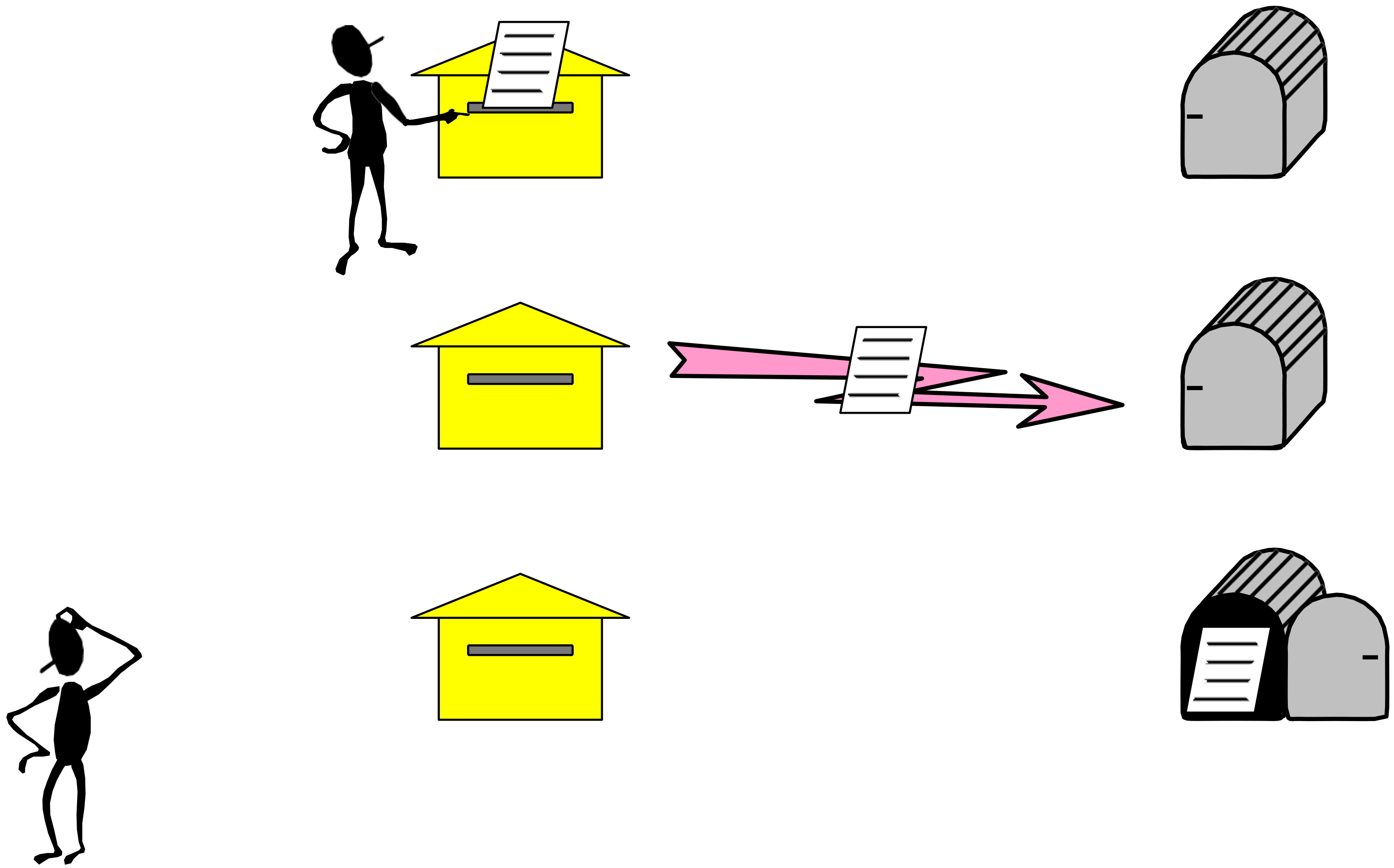
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left.



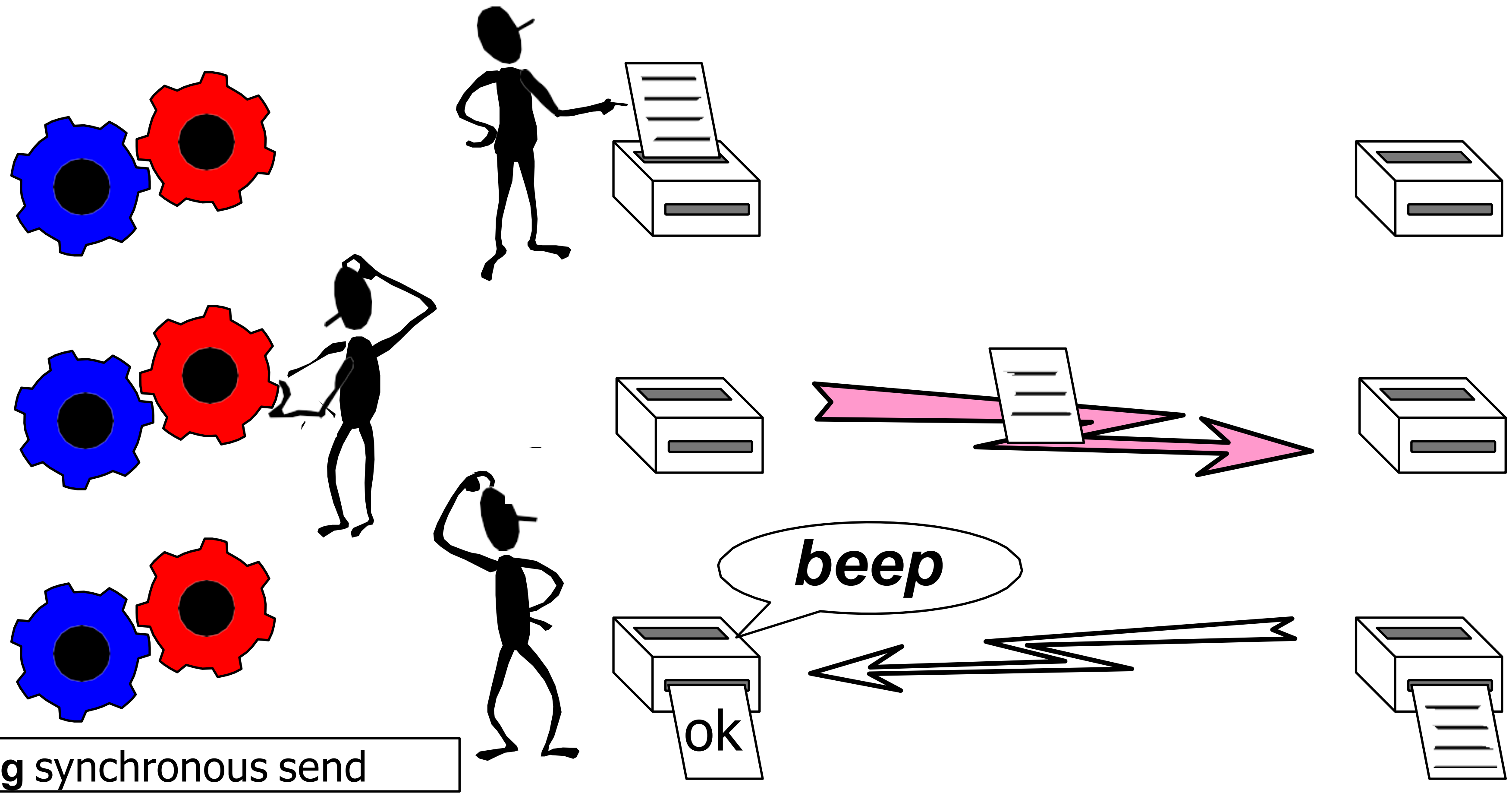
Blocking Operations

- Operations are activities, such as
  - sending (a message)
  - receiving (a message)
- Some operations may **block** until another process acts:
  - synchronous send operation **blocks until** receive is posted;
  - receive operation **blocks until** message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

Nonblocking Operations

Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must ***test*** or ***wait*** for the completion of the nonblocking operation



nonblocking synchronous send

Non-Blocking Operations (cont'd)

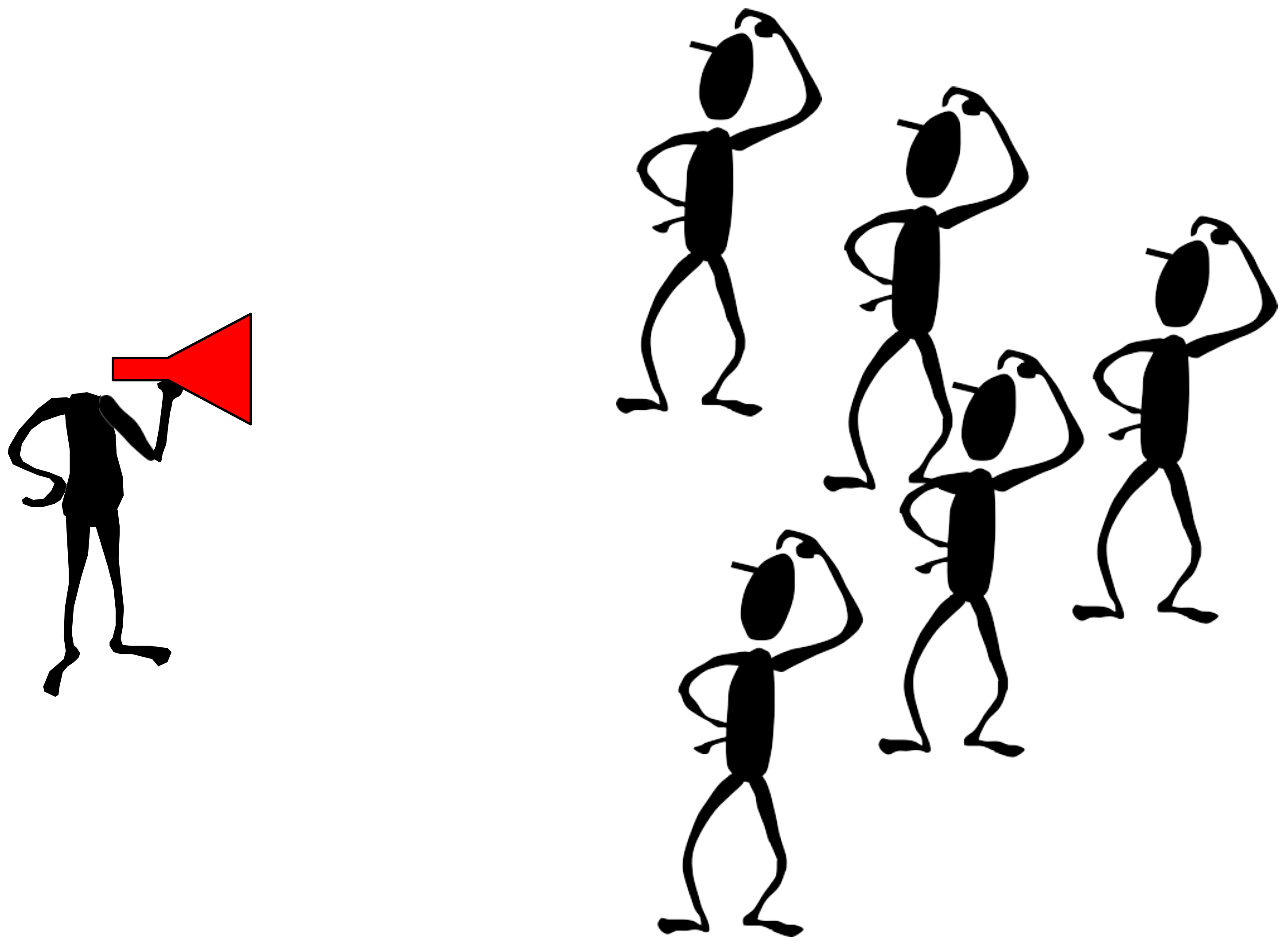
- All nonblocking procedures must have a matching wait (or test) procedure. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.
- Nonblocking procedures are not the same as sequential subroutine calls:
  - the operation may continue while the application executes the next statements!

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms.
- Can be built out of point-to-point communications.

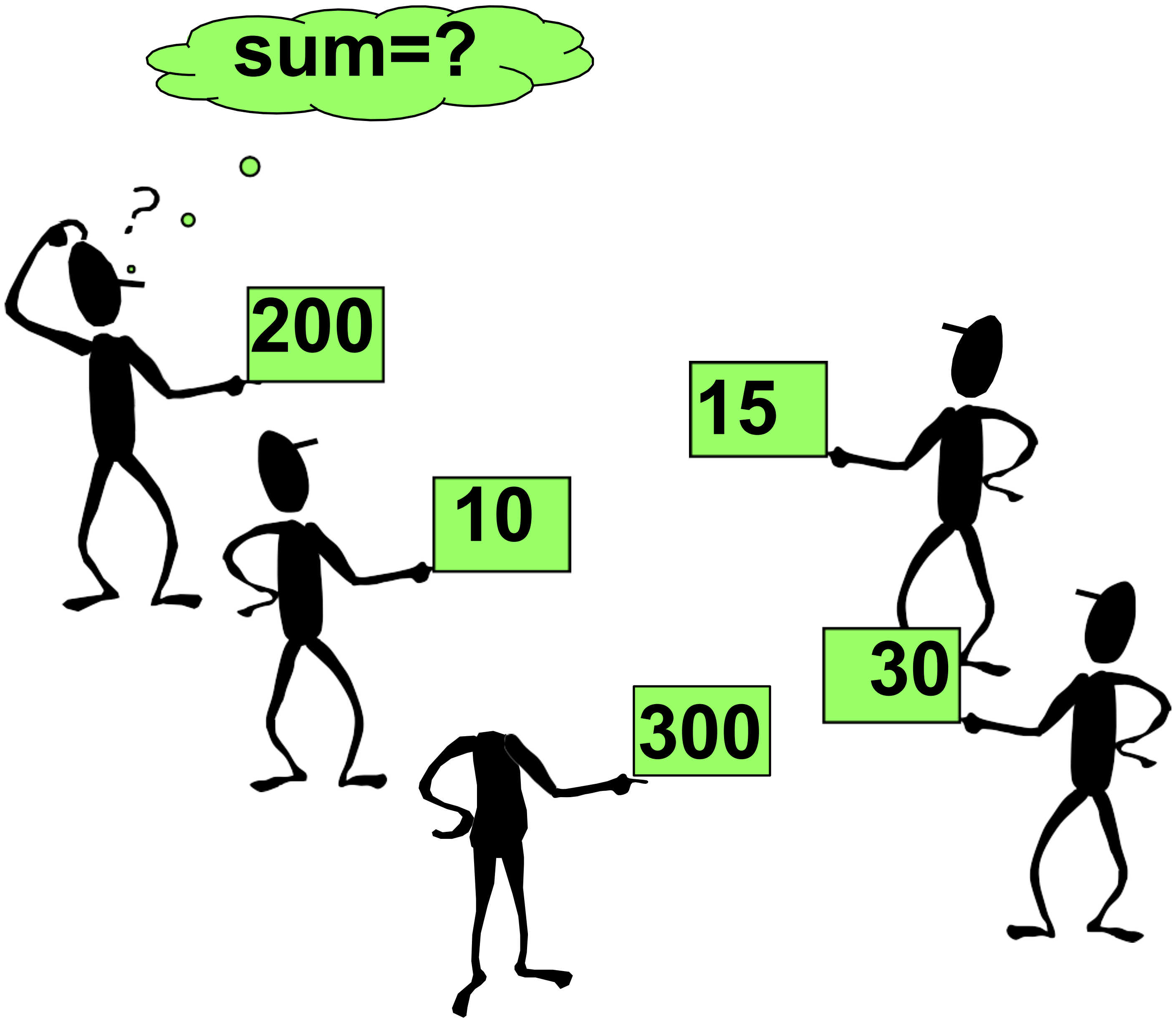
Broadcast

- A one-to-many communication.



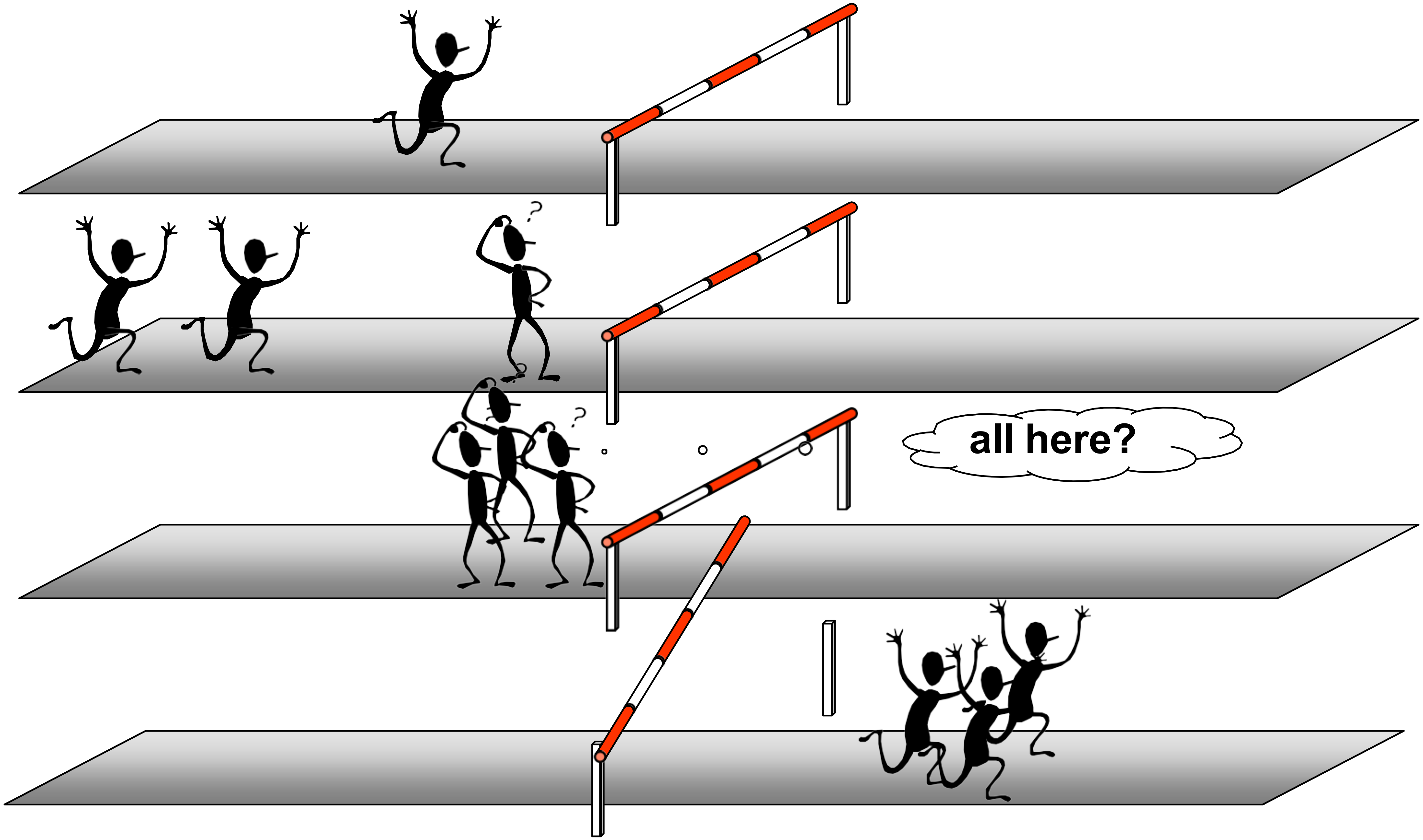
Reduction Operations

- Combine data from several processes to produce a single result.

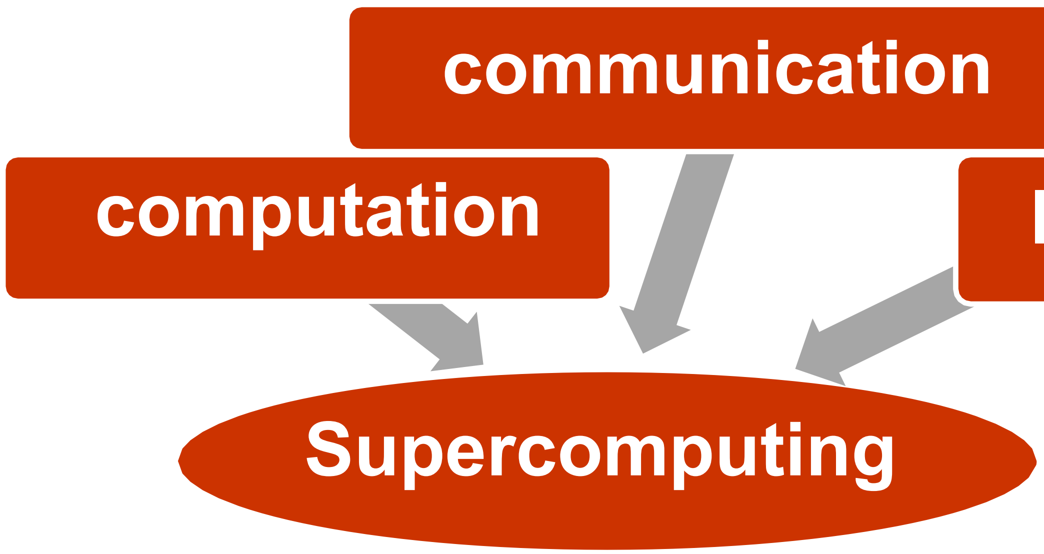


Barriers

- Synchronize processes.



Parallel File I/O



parallel computation → need for parallel I/O

→ do parallel I/O

calculation on	time for computation	time for serial I/O
1 core	64 min = 98.5 % of total time	1 min = 1.5 % of total time
64 cores	1 min = 50 % of total time	1 min = **50 %** of total time

Table: example with serial I/O

1) waste of resources
2) negative side effects on other users

Process Model & Language Binding

Header files

C

- C / C++

#include <mpi.h>

Python

- Python

from mpi4py import MPI

MPI Function Format

In C and Python: case sensitive

C

- C / C++: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);

Python

- Python: result_value_or_object = input_mpi_object.mpi_action(parameter, ...)

direct communication of numPy arrays (like in C)	comm_world = **MPI.COMM_WORLD** comm_world.**Send**((snd_buf, ...), ...) comm_world.**Recv**((rcv_buf, ...), ...)

Or with object-serialization:
comm_world.**send**(snd_buf, ...)
rcv_buf = comm_world.**recv**(...)

Mixed cases:

- MPI procedures in C — MPI_Xxx_mixed
- MPI type declaration — MPI_Xxx_mixed
- MPI constant — MPI_XXX_UPPER

Initializing MPI

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C

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

MPI-2.0 and higher: Also MPI_Init(NULL, NULL);

```
#include <mpi.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  ....
```

Python

- # MPI.Init()

This call is not needed, because automatically called at the import of MPI at the begin of the program

```
from mpi4py import MPI
# MPI.Init() is not needed
....
```

Exiting MPI

C

Python

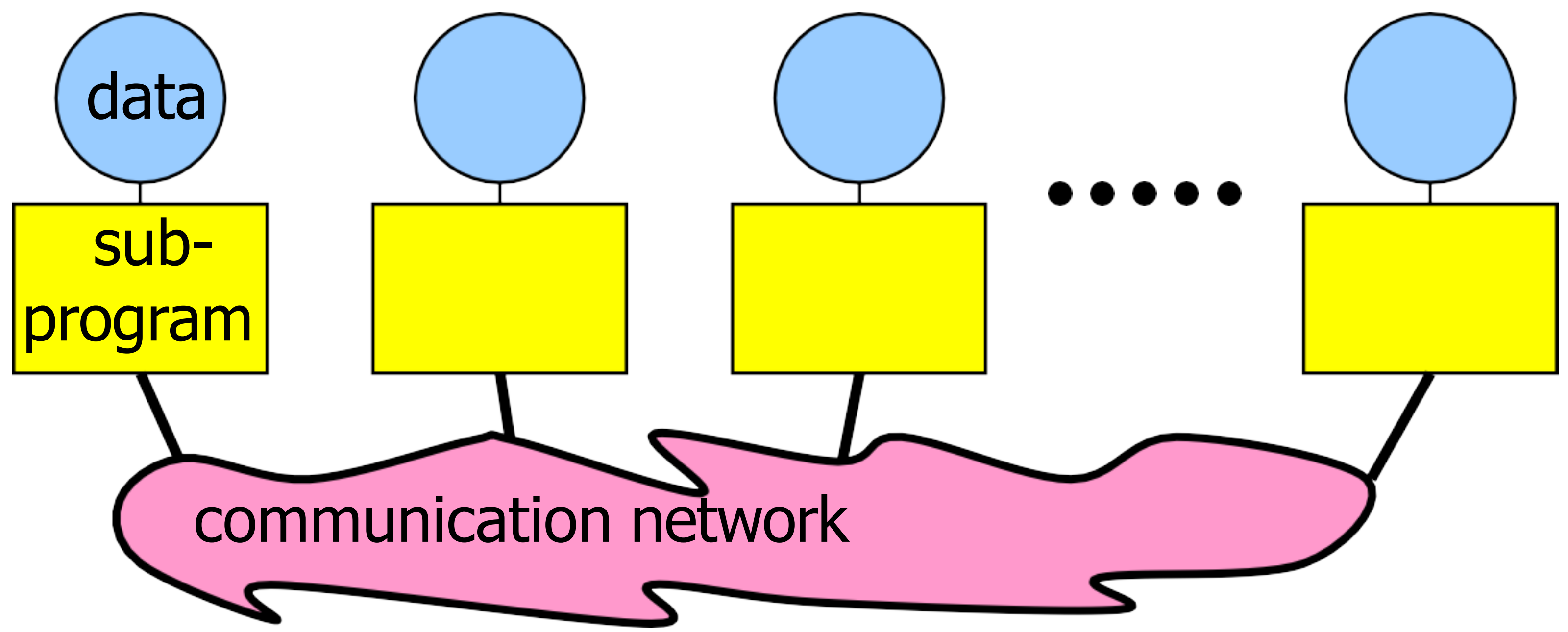
- C/C++: int MPI_Finalize()
- Python: #MPI.Finalize()

This call is not needed,
because automatically called at the end of the program

- **Must** be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden
  - **May** abort the calling process if its rank in MPI_COMM_WORLD is ≠0

Starting the MPI Program

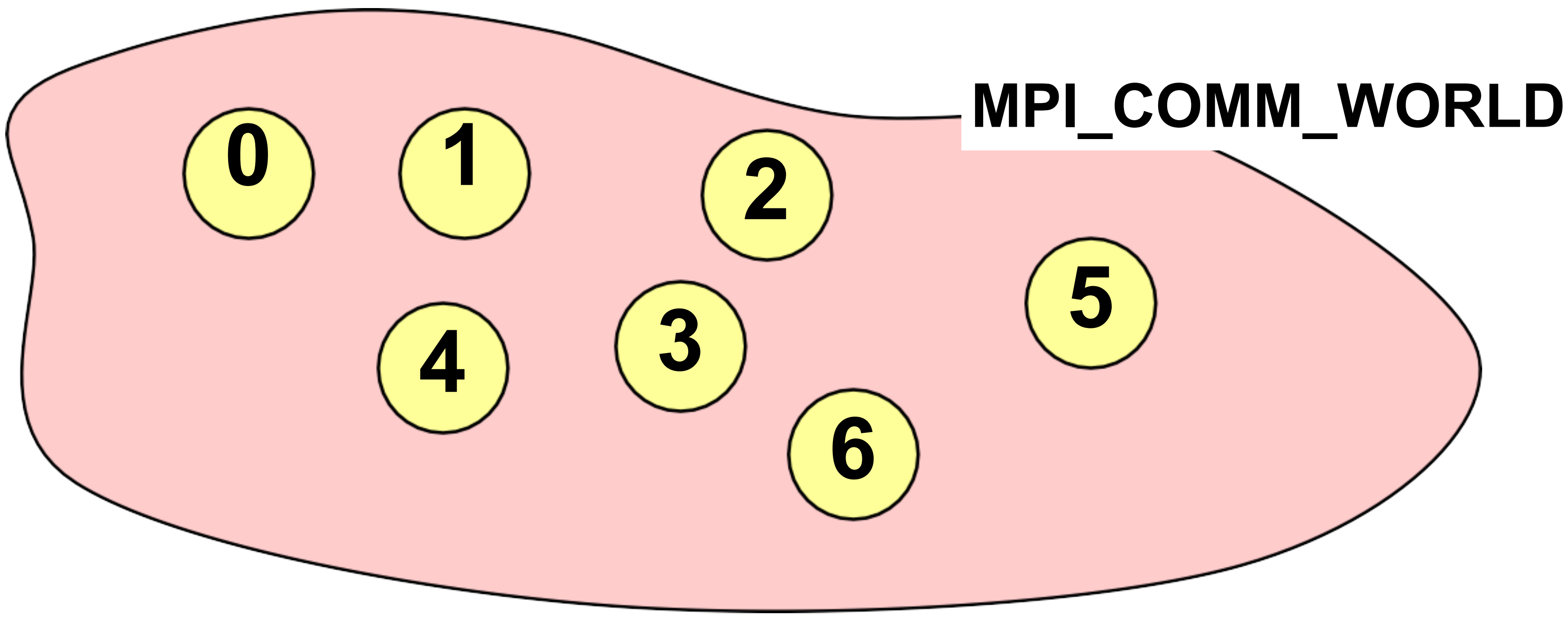
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 and later)



- The parallel MPI processes exist at least after MPI_Init was called.

Communicator MPI_COMM_WORLD

- All processes (= sub-programs) of one MPI program are combined in the **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in
  - mpi.h
  - Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



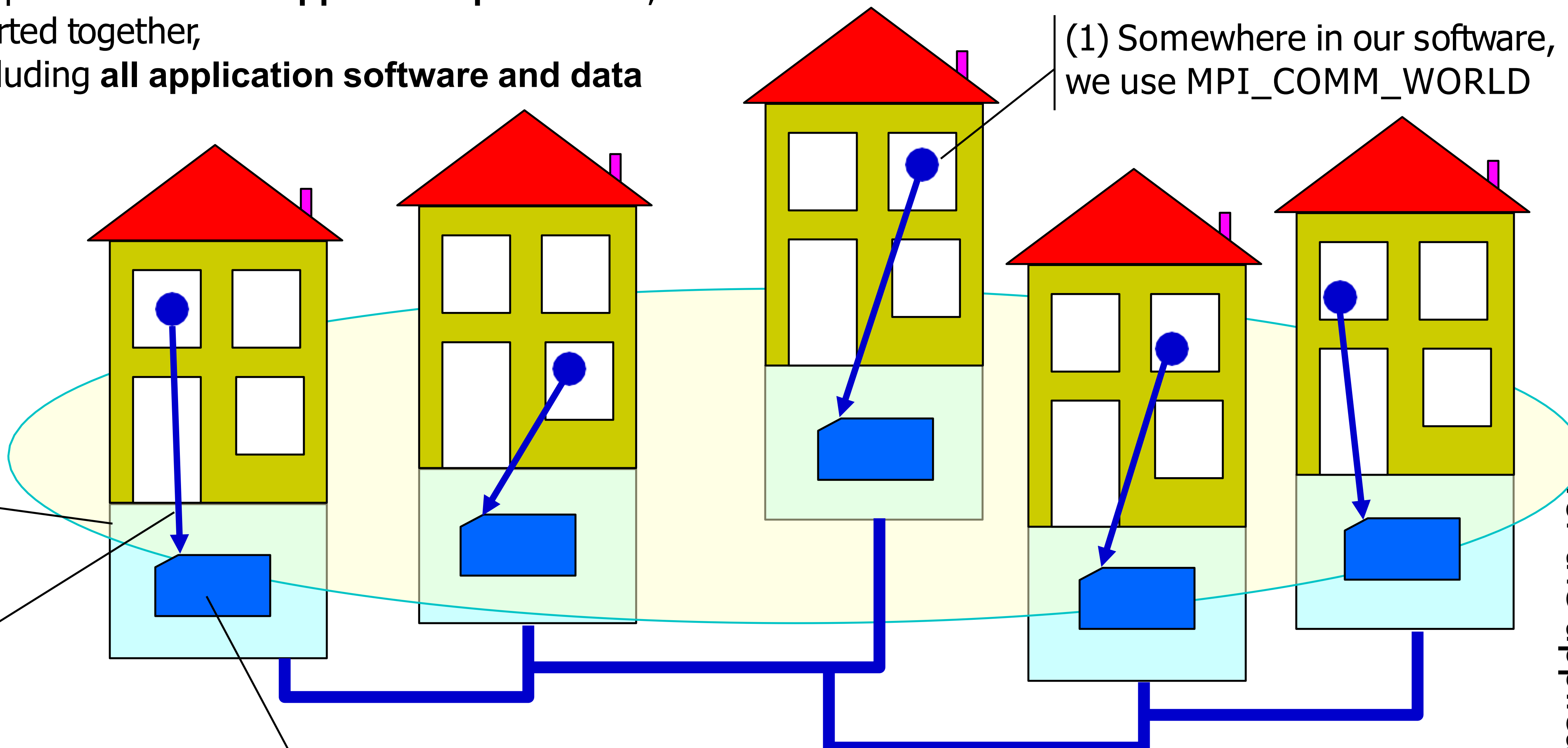
Handles refer to internal MPI data structures

Example with **five MPI application processes**,

- started together,
- including **all application software and data**

(1) Somewhere in our software, we use MPI_COMM_WORLD

(2) Let's look into our MPI library



(3) In each process, the handle MPI_COMM_WORLD refers to an MPI-internal data structure within that process

(4) And here we find the knowledge

- about the own process (e.g. the own rank)
- and about all other processes, and
- how to communicate with them

Visible | Invisible for the application programmer

Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - **predefined constants** in C include file **mpi.h** or **MPI module of mpi4py**
    - Example: MPI_COMM_WORLD or MPI.COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - They are link-time constants, i.e., need not to be compile-time constants.
  - **values returned** by some MPI routines,
    to be stored in variables, that are defined as

C

Python

    - C: special MPI typedefs, e.g., MPI_Comm sub_comm;
    - Python: Type of object defined by the creating function,
      e.g., sub_comm = MPI.COMM_WORLD.Split(…)
- Handles refer to internal MPI data structures