CSCI 381/780: Parallel and Distributed Computing
Parallel Programming with MPI

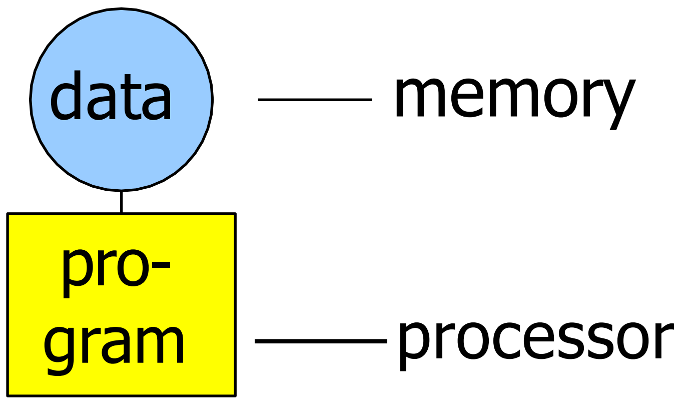
Jun Li
Queens College & Graduate Center
jun.li@qc.cuny.edu

Roadmap

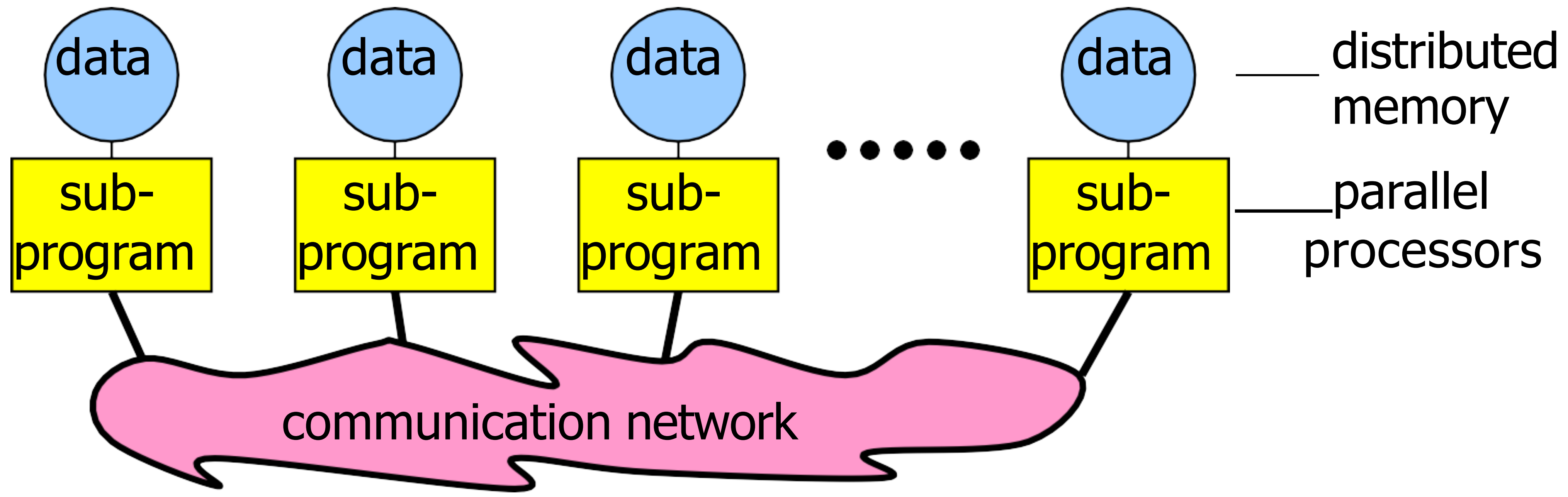
MPI Overview

The Message-Passing Programming Paradigm

- Sequential Programming Paradigm



- Message-Passing Programming Paradigm

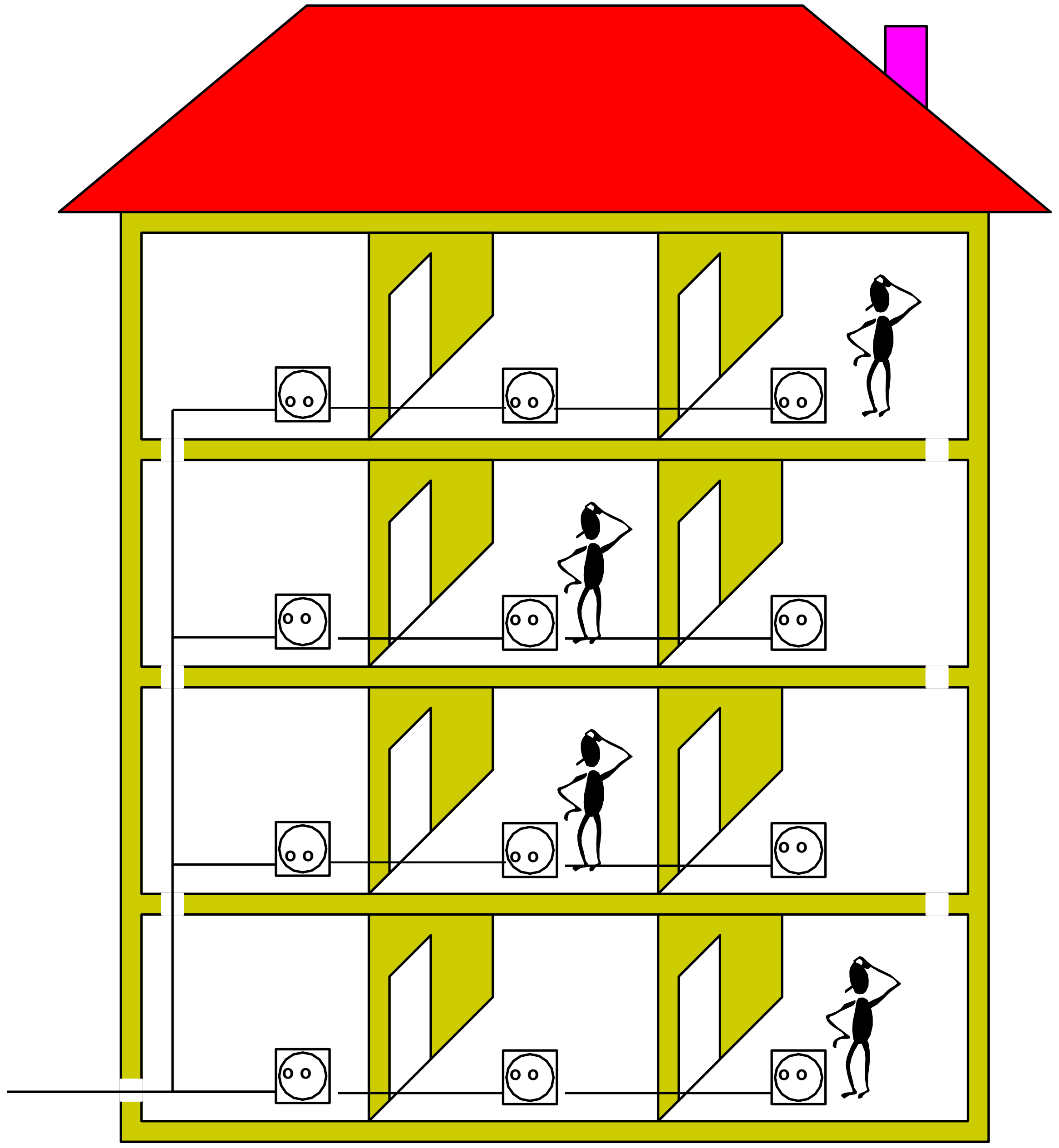


Analogy: Electric Installations in Parallel

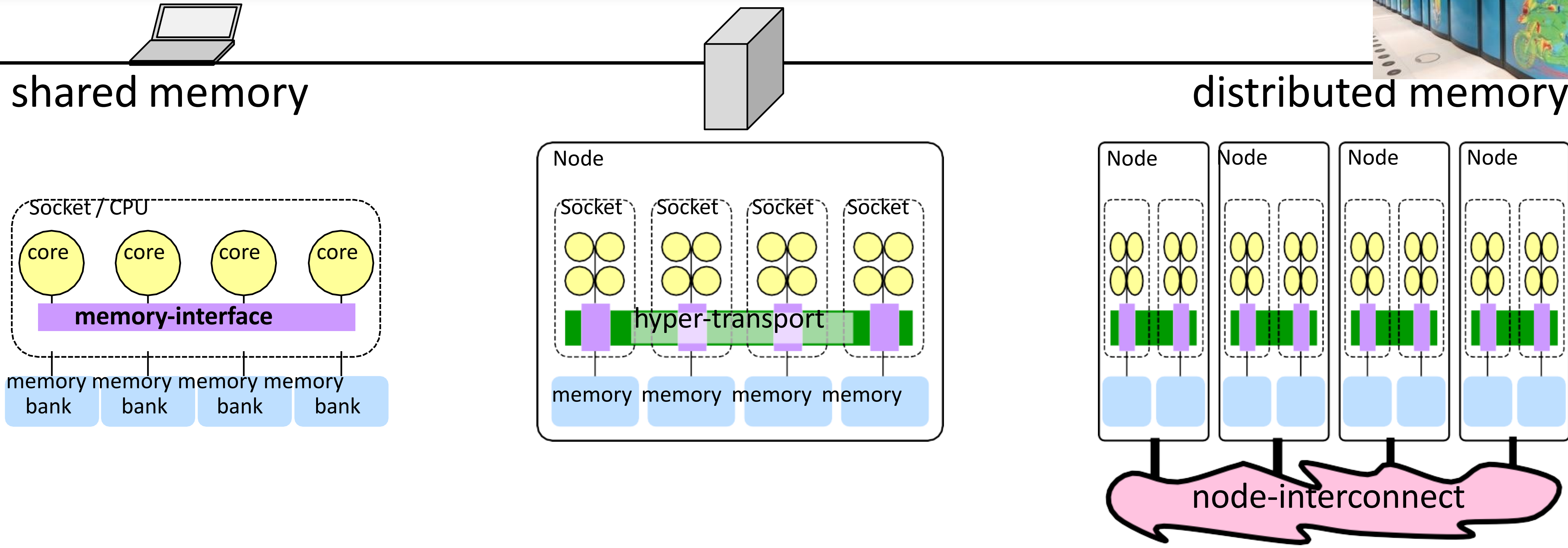
- MPI sub-program
  = work of one electrician
  on one floor

- data
  = the electric installation

- MPI communication
  = real communication
  to guarantee that the wires
  are coming at the same
  position through the floor



Parallel hardware architectures



Socket/CPU

→ **memory interface**

UMA (uniform memory access) SMP (symmetric multi-processing) All cores connected to all memory banks with same speed

Node

→ **hyper-transport**

ccNUMA (cache-coherent non-uniform memory access)

Shared memory programming is possible

Performance problems:

- Threads should be **pinned** to the physical sockets
- **First-touch** strategy is needed to minimize remote memory access

Cluster

→ **node-interconnect**

NUMA (non-uniform memory access)

!! fast access only on its own memory **!!**

Many programming options:

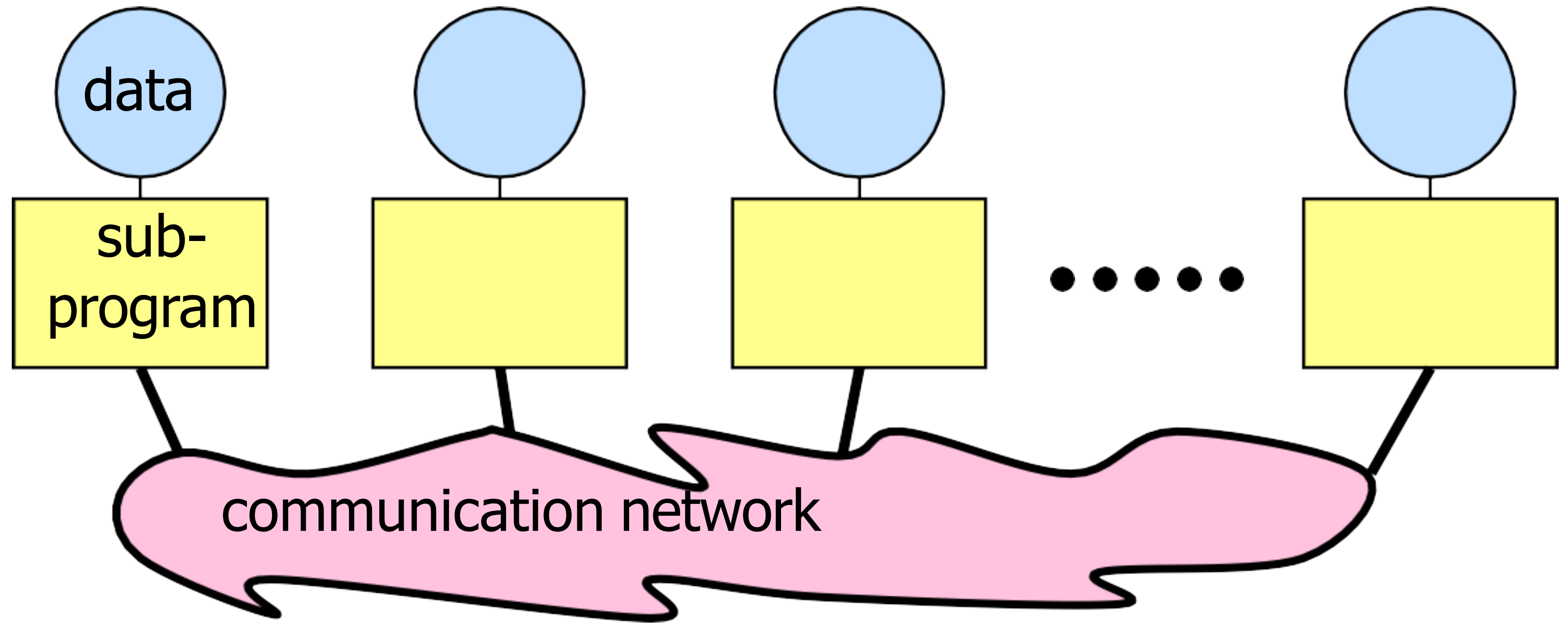
- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

$\underbrace{\text{Socket/CPU} \quad \text{Node}}_{\text{Shared memory programming with OpenMP}}$

$\underbrace{\text{Socket/CPU} \quad \text{Node} \quad \text{Cluster}}_{\textbf{MPI works everywhere}}$

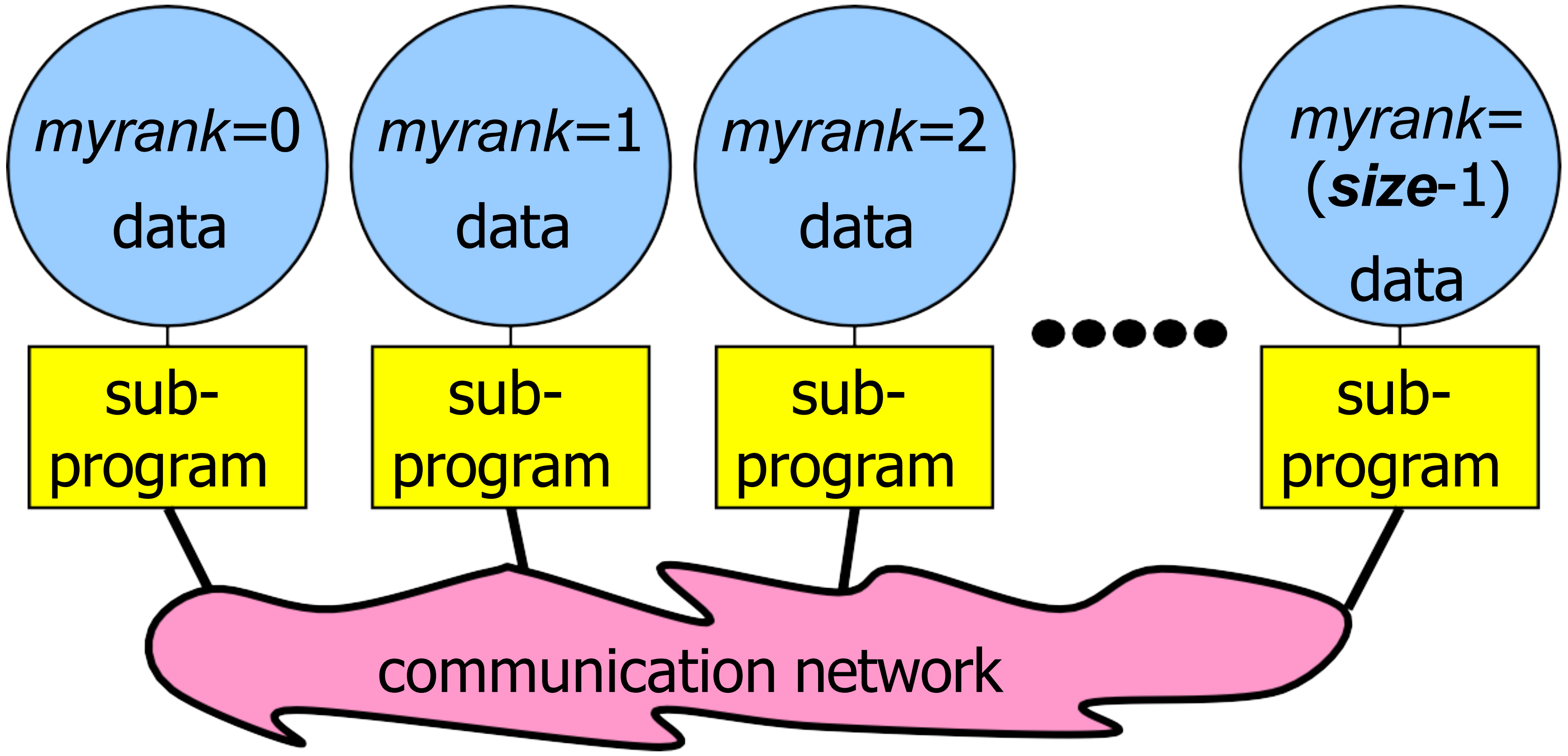
The Message-Passing Programming Paradigm

- Each processor in a message passing program runs a ***sub-program:***
  - written in a conventional sequential language, e.g., C, Fortran, or Python
  - typically the same on each processor (SPMD),
  - the variables of each sub-program have
    - **the same name**
    - **but different locations (distributed memory) and different data!**
    - **i.e., all variables are private**
  - communicate via special send & receive routines (***message passing***)



Data and Work Distribution

- the value of ***myrank*** is returned by special library routine
- the system of ***size*** processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on ***myrank***
- i.e., which process works on which data



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 !

```
#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

```
  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

```
  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

```
  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

```
  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
```