University of Pittsburgh

High Performance Computing with Python

Esteban Meneses, PhD
emeneses@pitt.edu

Kenneth P. Dietrich School of Arts and Sciences
Center for Simulation and Modeling (SaM)

THE FOLLOWING PRESENTATION HAS BEEN RATED

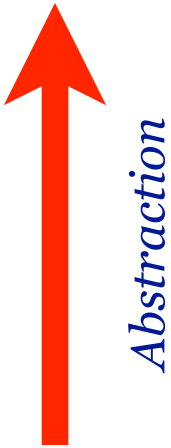
PG-18	PROGRAMMERS STRONGLY CAUTIONED
MOST CODES ARE NOT PROPERLY OPTIMIZED	
PYTHON FANATICS MAY FIND SOME MATERIAL OFFENSIVE	

Downloads

http://coco.sam.pitt.edu/~emeneses/teaching/hpc-python

Tools

Python	*Data management*
MPI4Py	*Interface*
MPI	*Communication*



Source of images: http://www.wikipedia.org/

Why Python?

- Simple syntax, very expressive.
- Good mix of programming paradigms.
- Great for rapid **prototyping**.
- One of the 3 most popular languages in high performance computing (HPC); along with C and Fortran.
- Mature language (since 1991) with a huge user base and lots of extension libraries.

Why MPI?

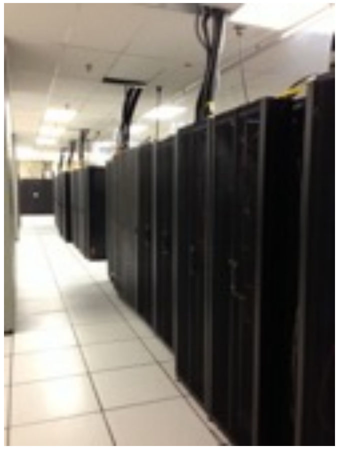
- *De facto* standard language for parallel computing on HPC gear.
- Simple communication model between processes in a program.
- Multiple implementations; highly efficient for different platforms.
- Well established community (since 1994).
- Large ecosystem of tools and applications built on MPI.

Why MPI4Py?

- Well regarded implementation of MPI on Python among competing alternatives.
- Clean and efficient MPI interface for Python.
- Covers most of the MPI-2 standard, including dynamic process creation.
- Extensible and compatible implementation.
- Responsive technical support.

Scalable Performance

Titan



Frank

Laptop

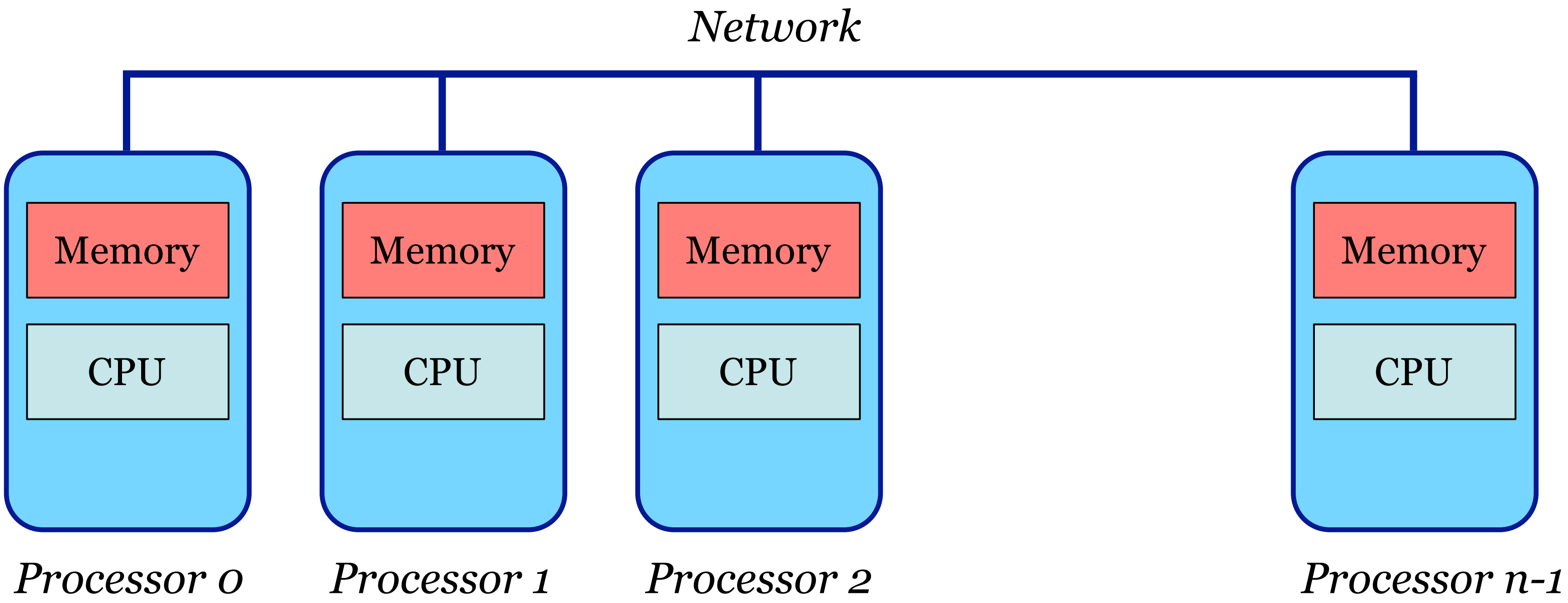
Contents

- Basics of Message Passing Interface (MPI)
  - Point-to-point communication
  - Collective communication operations
- Parallel 5-point Stencil in MPI

Basics of Message Passing Interface (MPI)

Distributed Memory Systems

- Each processor has its own private memory.
- A network connects all the processors.



Message-Passing Paradigm

- A parallel program is decomposed into processes, called **ranks**.
- Each rank holds a portion of the program's data into its private memory.
- Communication among ranks is made explicit through messages.
- Channels honor first-in-first-out (FIFO) ordering.

Single-Program Multiple-Data (SPMD)

- All processes run the same program, each accesses a different portion of data.
- All processes are launched simultaneously.
- Communication:
  - Point-to-point messages.
  - Collective communication operations.

Features of Message Passing

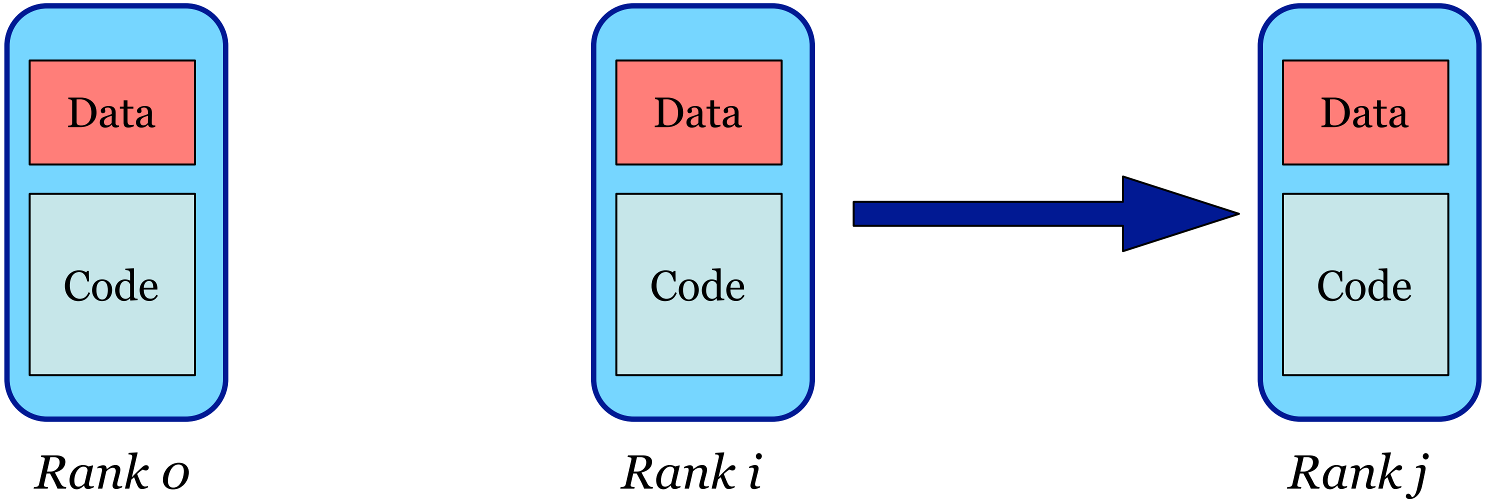
- **Simplicity**: the basics of the paradigm are traditional communication operations.
- **Generality**: can be implemented on most parallel architectures.
- **Performance**: the implementation can match the underlying hardware.
- **Scalability**: the same program can be deployed on larger systems.

Message Passing Interface (MPI)

- Standard for operations in message passing.
- Led by **MPI Forum** (academia & industry).
  - Standards: MPI-1 (1994), MPI-2 standard (1997), MPI-3 (2012).
- Implementations:
  - Open-source: MPICH, Open MPI.
  - Proprietary: Cray, IBM, Intel.

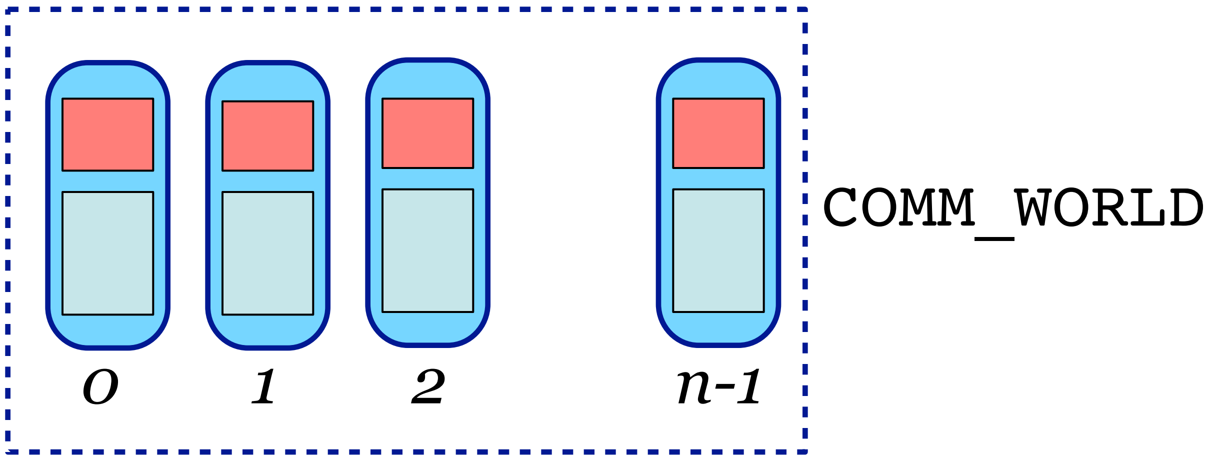
MPI Ranks

- Ranks have private memory.
- Each rank has a unique identification number.
- Ranks are numbered sequentially: `[0,n-1]`.



MPI Communicators

- Groups of ranks among which a rank can communicate.
- `COMM_WORLD` is a communicator including all ranks in the system.



MPI4Py Primer

- Importing library:

```
from mpi4py import MPI
```

- Getting important information:

```
comm = MPI.COMM_WORLD
rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
name = MPI.Get_processor_name()
```

- Executing in parallel:

```
mpirun -np <P> python <code>.py
```

Exercise 1

- Write a parallel Python “Hello World”. Run the program with different number of ranks.
- Example:

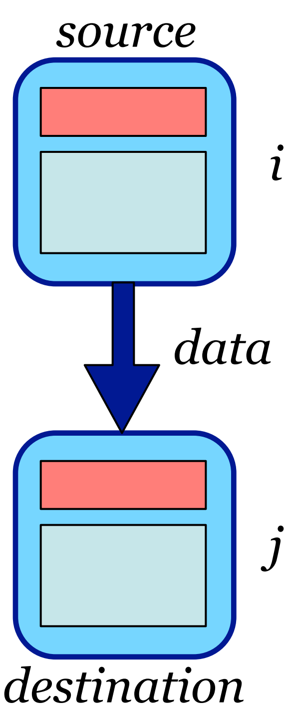
```
mpirun -np 4 python hello.py
```

- Output:

```
Hello, world! This is rank 0 of 4 running on kolmakov.sam.pitt.edu
Hello, world! This is rank 3 of 4 running on kolmakov.sam.pitt.edu
Hello, world! This is rank 1 of 4 running on kolmakov.sam.pitt.edu
Hello, world! This is rank 2 of 4 running on kolmakov.sam.pitt.edu
```

Point-to-point Operations

- Synchronous instructions to send a message from one *source* rank to a *destination* rank.




```
comm.send(data, dest=j)

data = comm.recv(source=i)
```

Exercise 2

- Implement a parallel ping-pong in Python. A message carrying a counter is exchanged between the two ranks 1000 times. Rank 1 will increment the counter upon reception.
- Example:

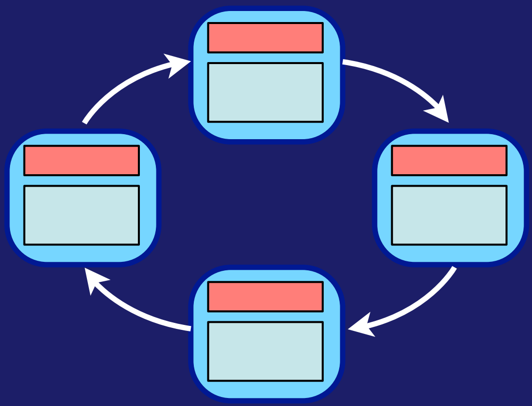
```
mpirun -np 2 python pingpong.py
```

- Output:

```
Total number of message exchanges: 1000
```

Exercise 3

- Implement a parallel Python program that creates a ring of ranks. Each rank gets a random integer value [0,100]. The program computes in each rank the sum of all the values by circulating them around the ring.



- Example:

```
mpirun -np 4 python ring.py
```

- Output:

```
[2] Total sum: 172
[3] Total sum: 172
[0] Total sum: 172
[1] Total sum: 172
```

Collective Communication Operations

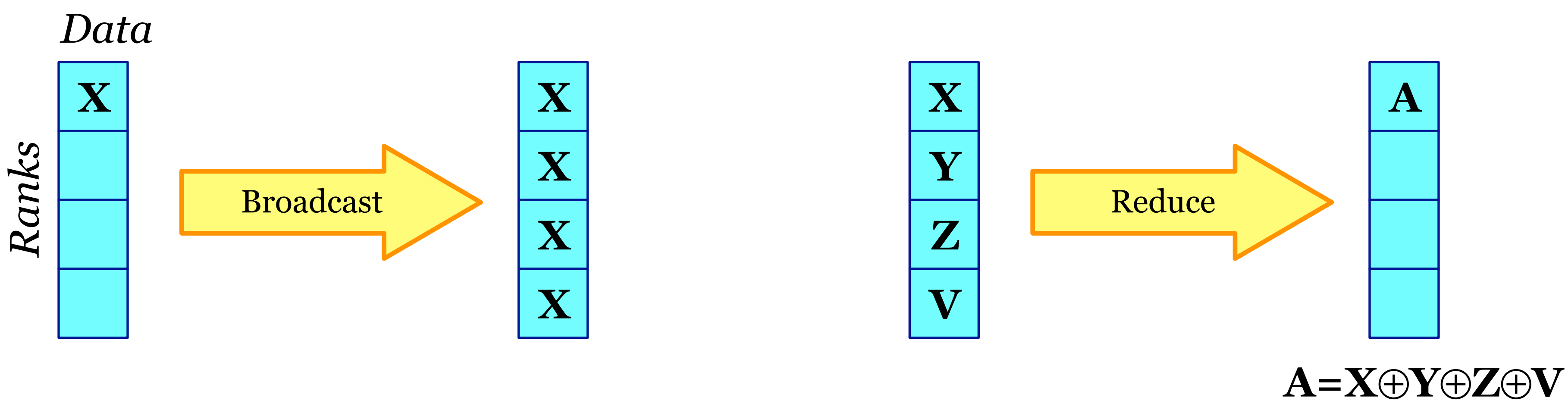
- Instructions to exchange data including all the ranks in a communicator.
- The **root** rank indicates the source or destination of the operation.
- **Broadcast**: one to many.

```
comm.bcast(data, root=0)
```

- **Reduction**: many to one.

```
comm.reduce(data, op=MPI.SUM, root=0)
```

Collectives



Exercise 4

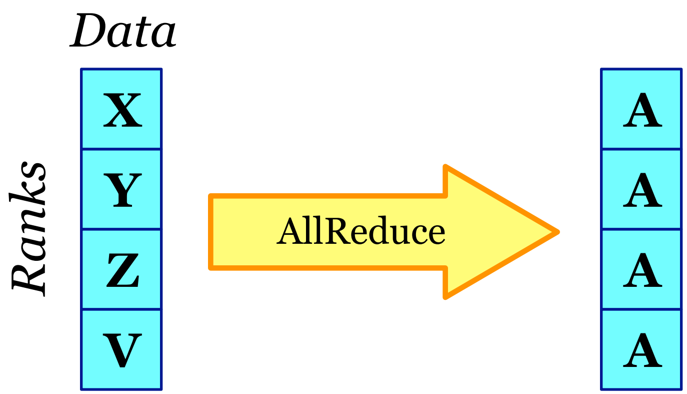
- Write a parallel Python program where each rank gets a random integer value [0,100]. The program gets in each rank the sum of all the values in the ranks using only broadcast and reduction.
- Example:

```
mpirun -np 4 python collective.py
```

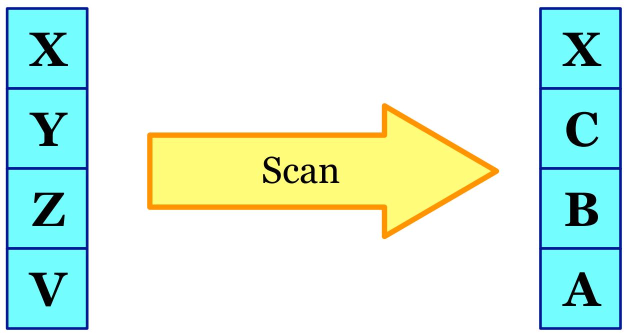
- Output:

```
[0] Total sum: 220
[1] Total sum: 220
[3] Total sum: 220
[2] Total sum: 220
```

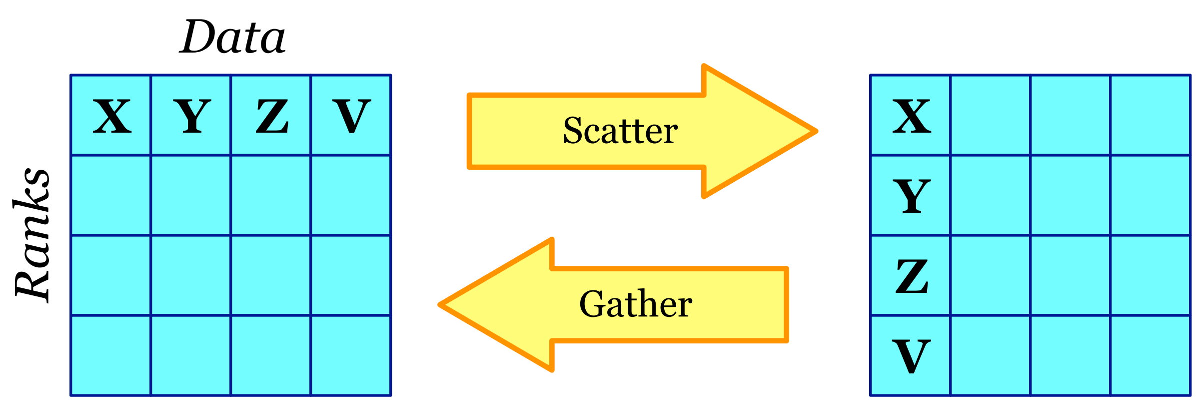
Collectives (cont.)



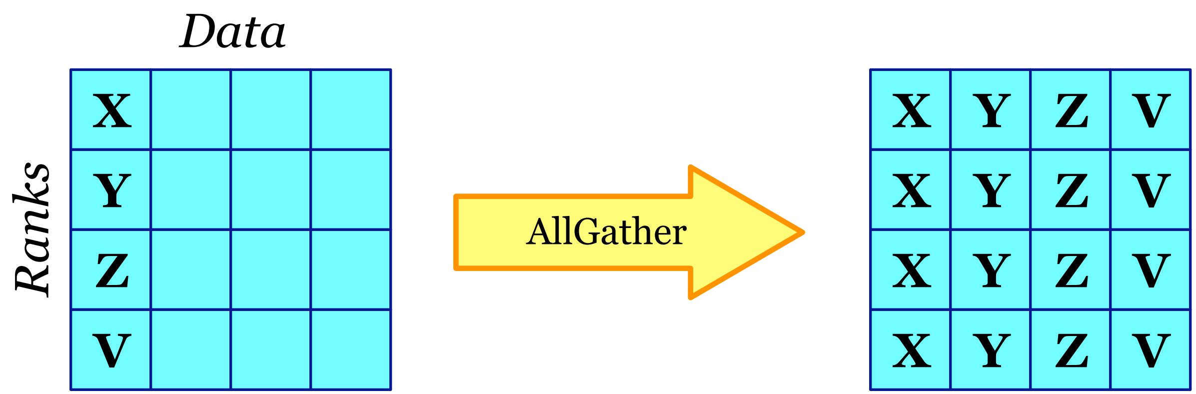
$A = X \oplus Y \oplus Z \oplus V$

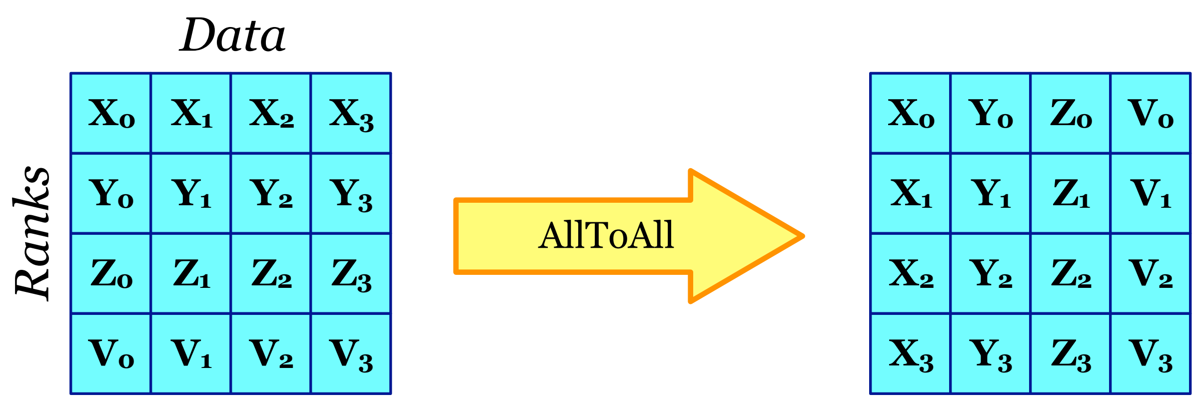


$C = X \oplus Y$
$B = X \oplus Y \oplus Z$
$A = X \oplus Y \oplus Z \oplus V$



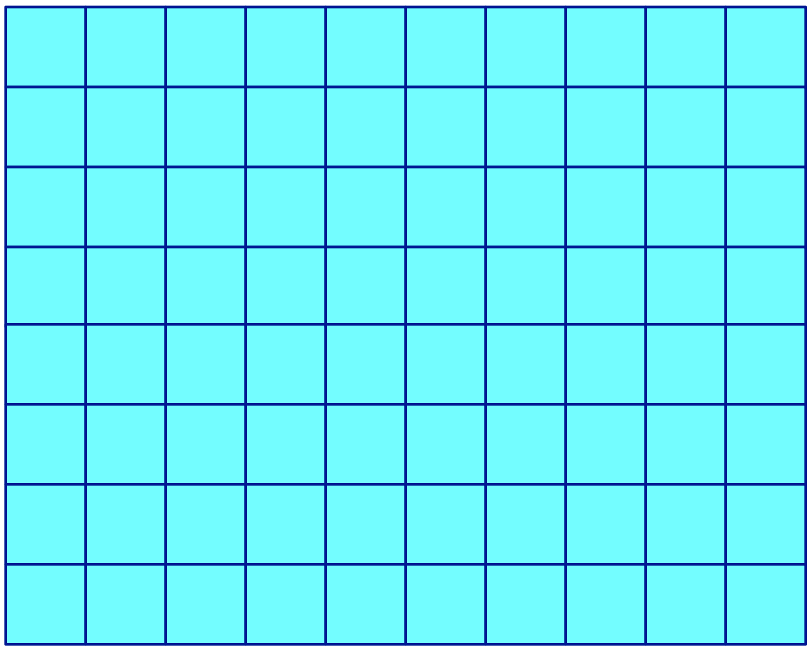
Collectives (cont.)



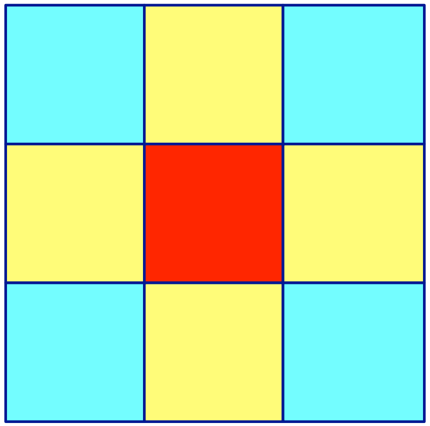


Stencil Algorithm

Two-dimensional stencil



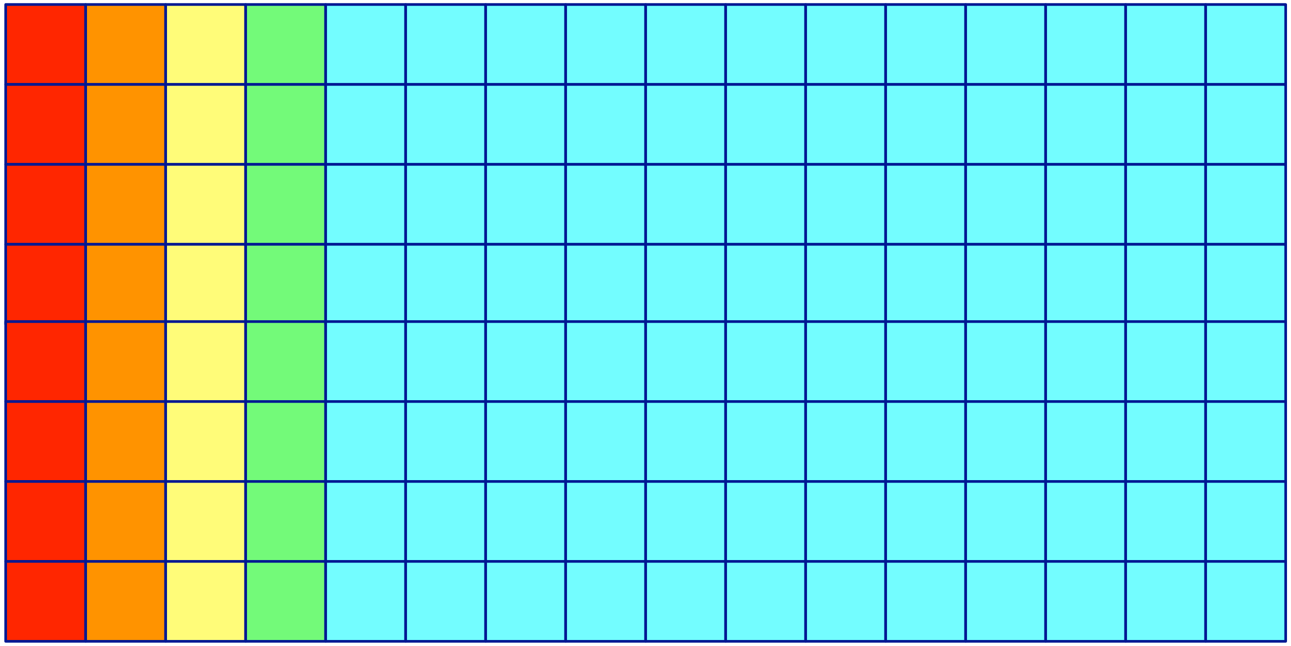
Matrix *M*



5-point stencil

$$M[i,j]=0.2*(M[i-1,j]+M[i,j-1]+ M[i,j]+ M[i+1,j]+M[i,j+1])$$

Heat Transfer



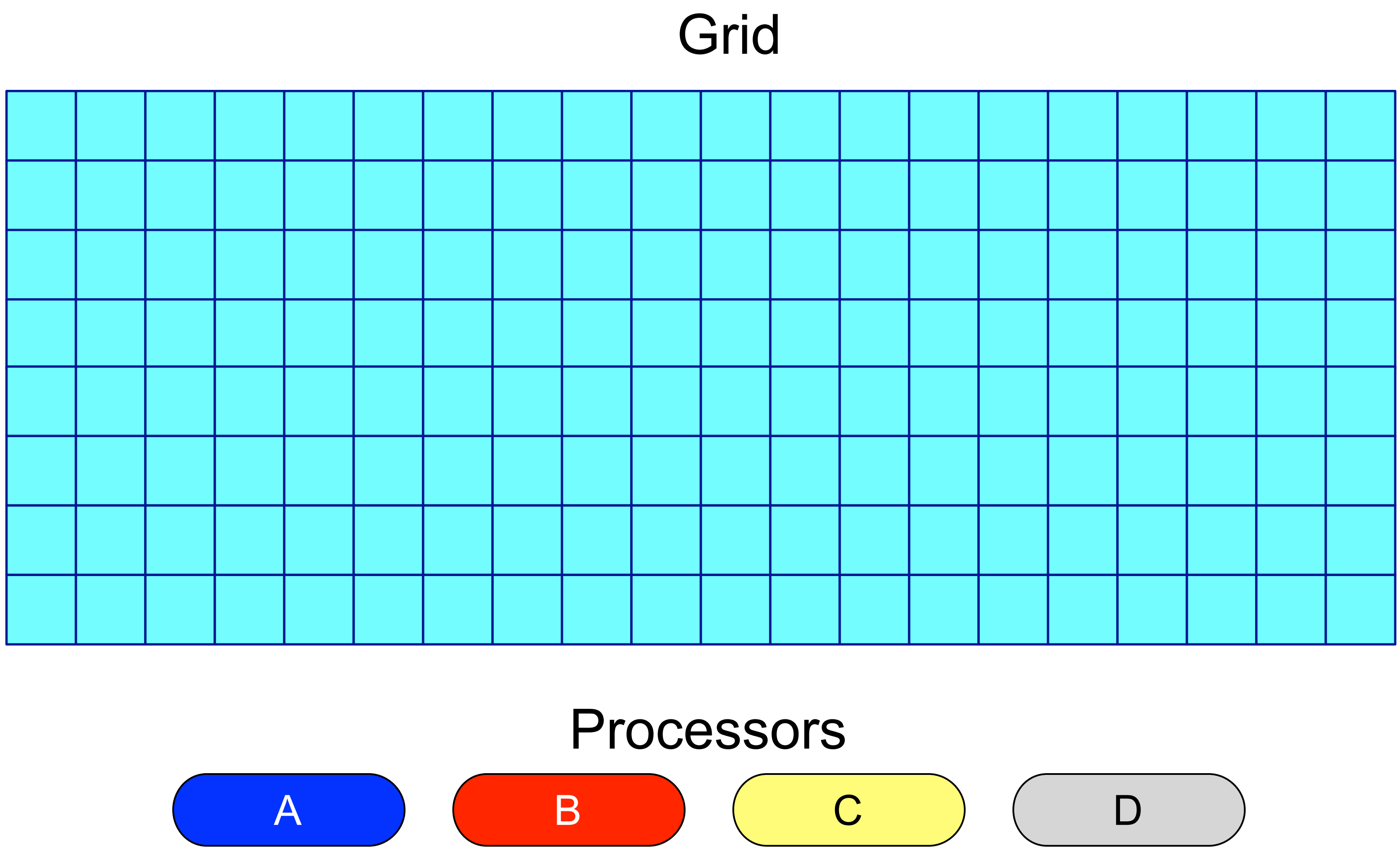
Exercise

- Implement in Python the 2D stencil algorithm. The program should receive these parameters:
  `<x> <y> <tolerance>`
  where *tolerance* represents the minimum error between iterations.

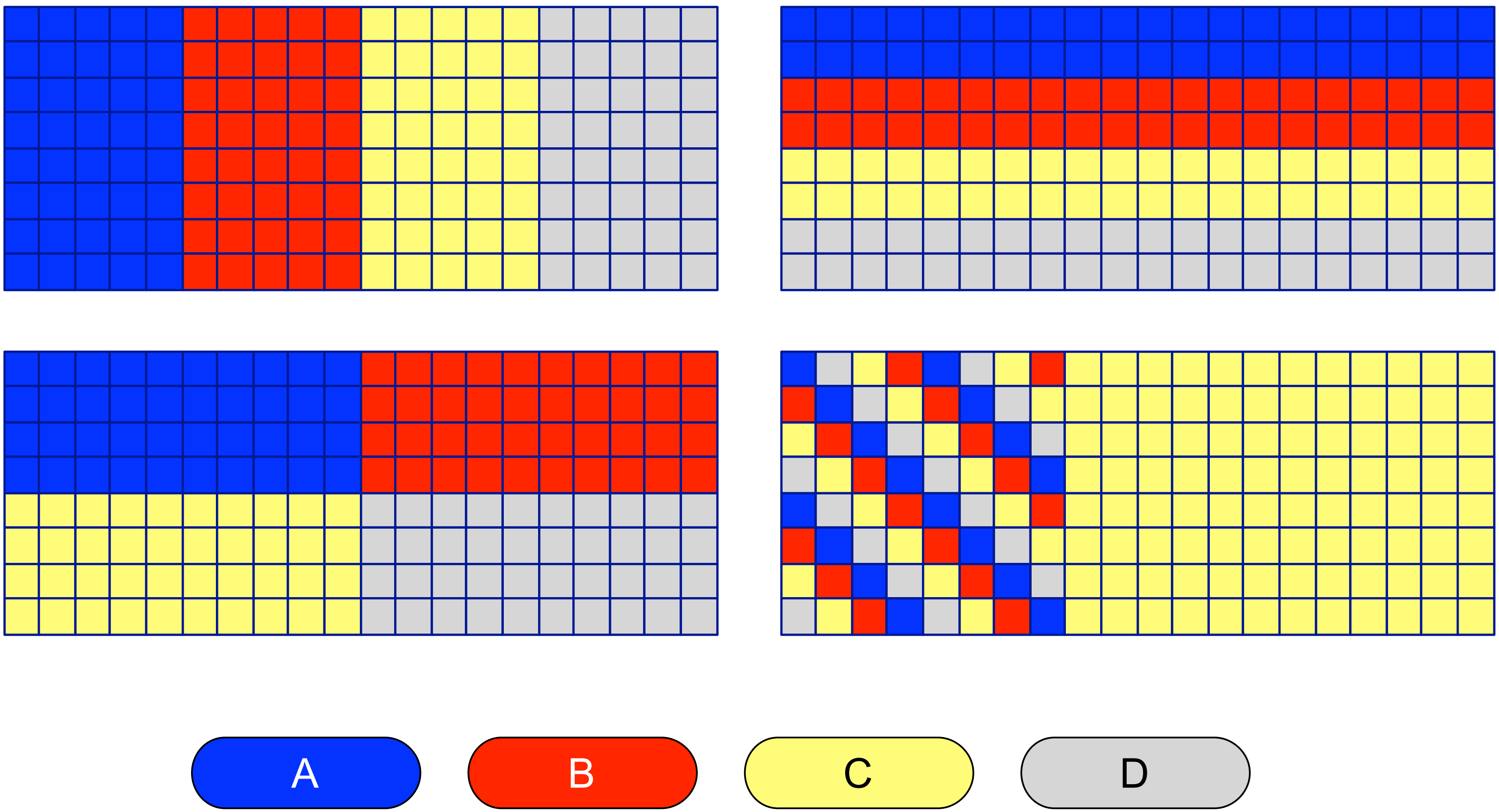
Parallel Stencil Computation

- Foster's Methodology:
  - **Partition**
    - *Load Balance*
  - Agglomeration
  - **Communication**
    - *Transfer Balance*
  - Mapping

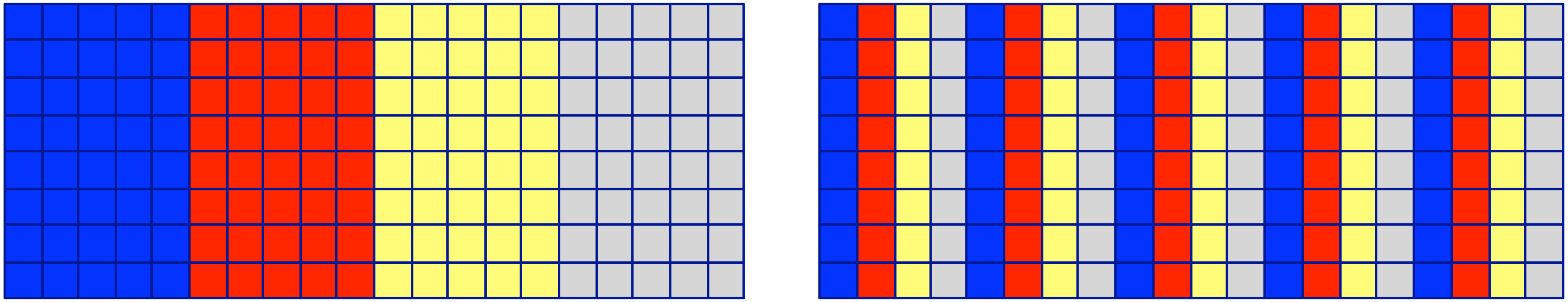
Problem Partition



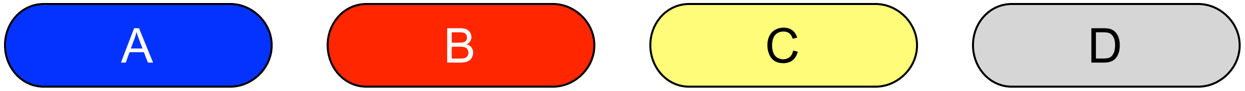
Partition Strategies



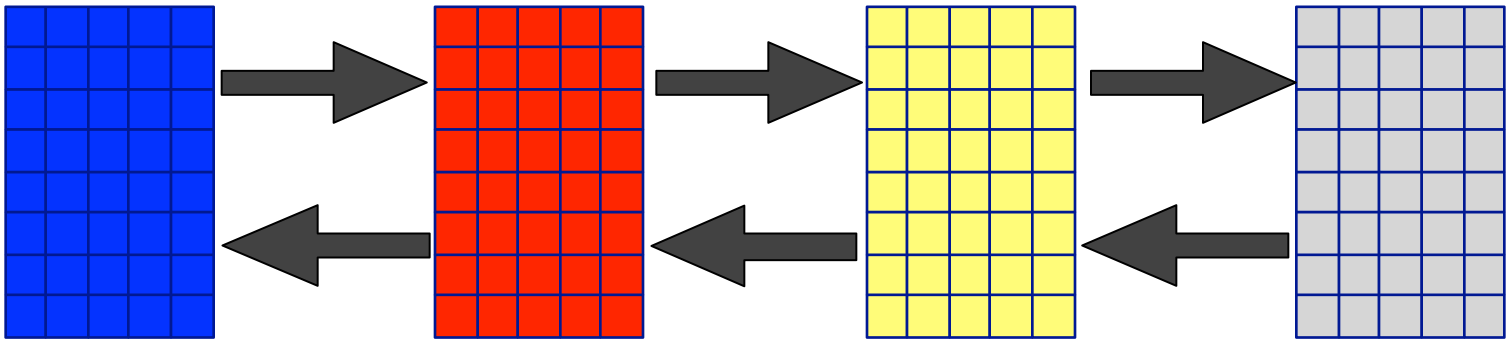
Vertical Decomposition



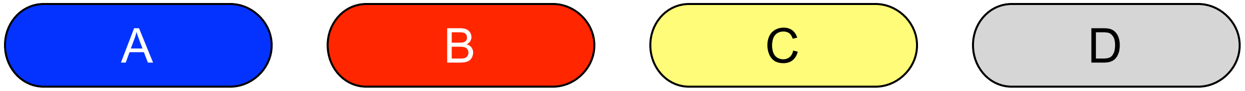
Balance between
frequency of communication and size of messages



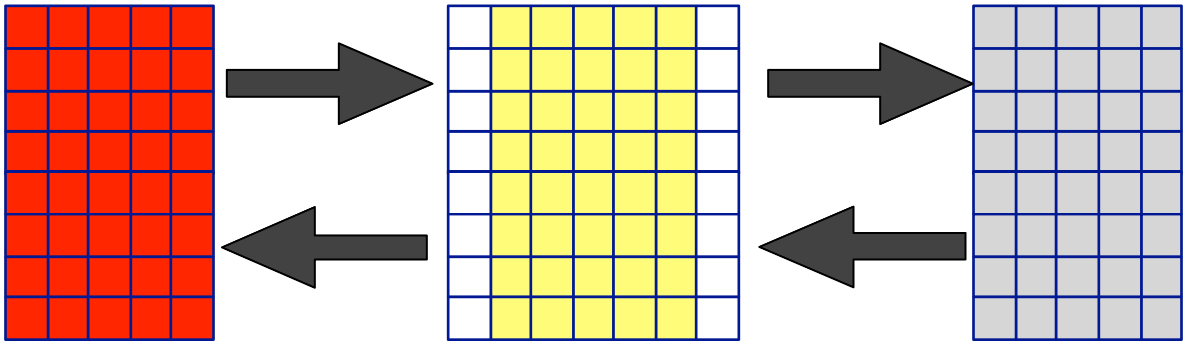
Communication Strategy



Point-to-point messages



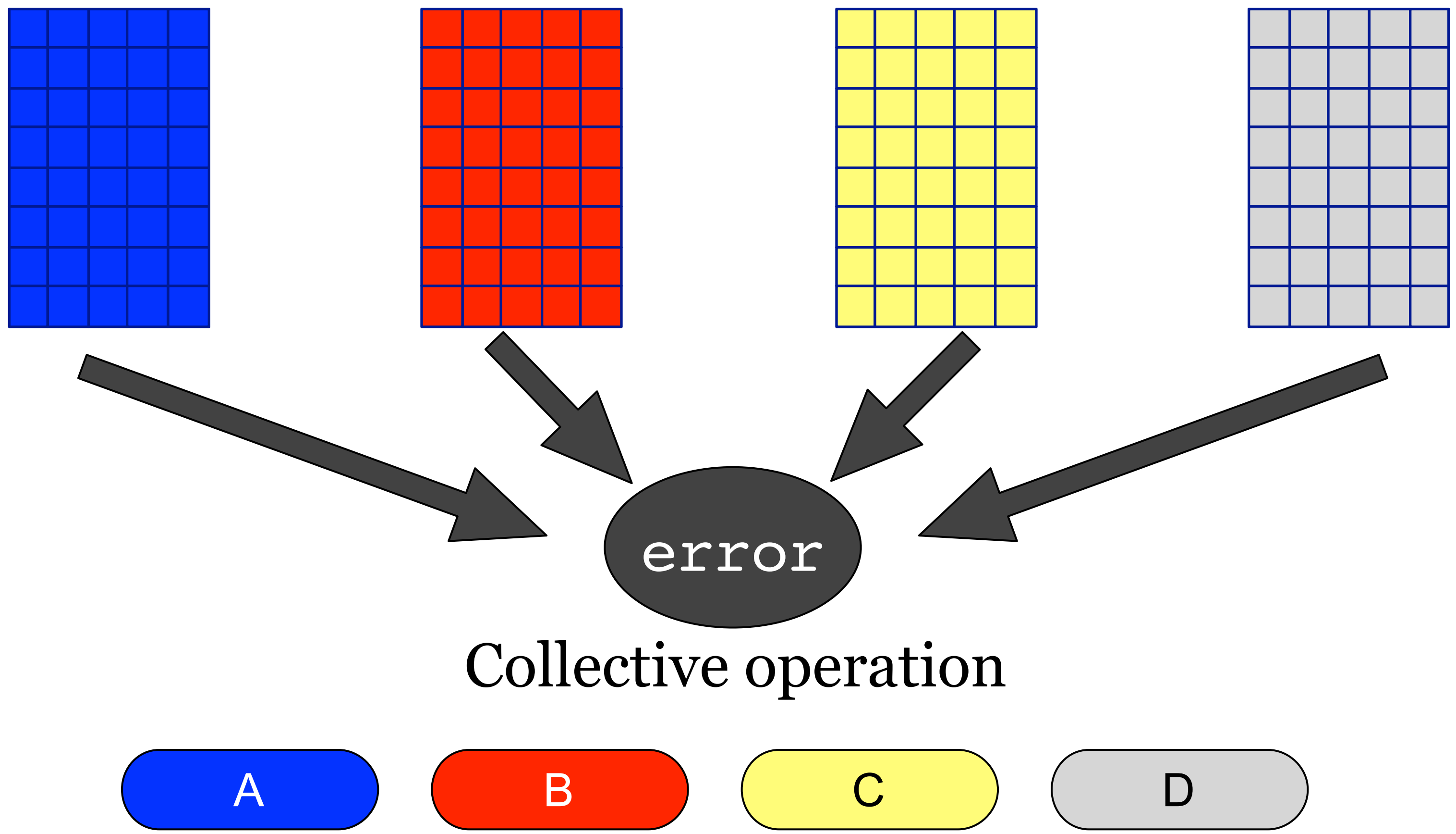
Ghost Cells



Buffers for external data

A B C D

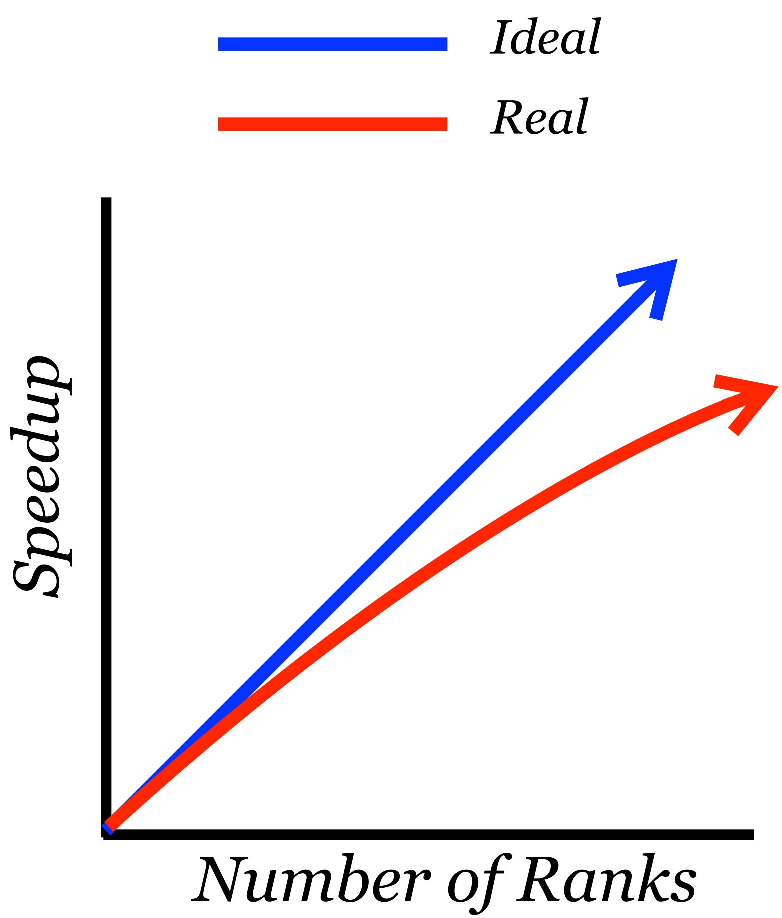
Global Information Strategy



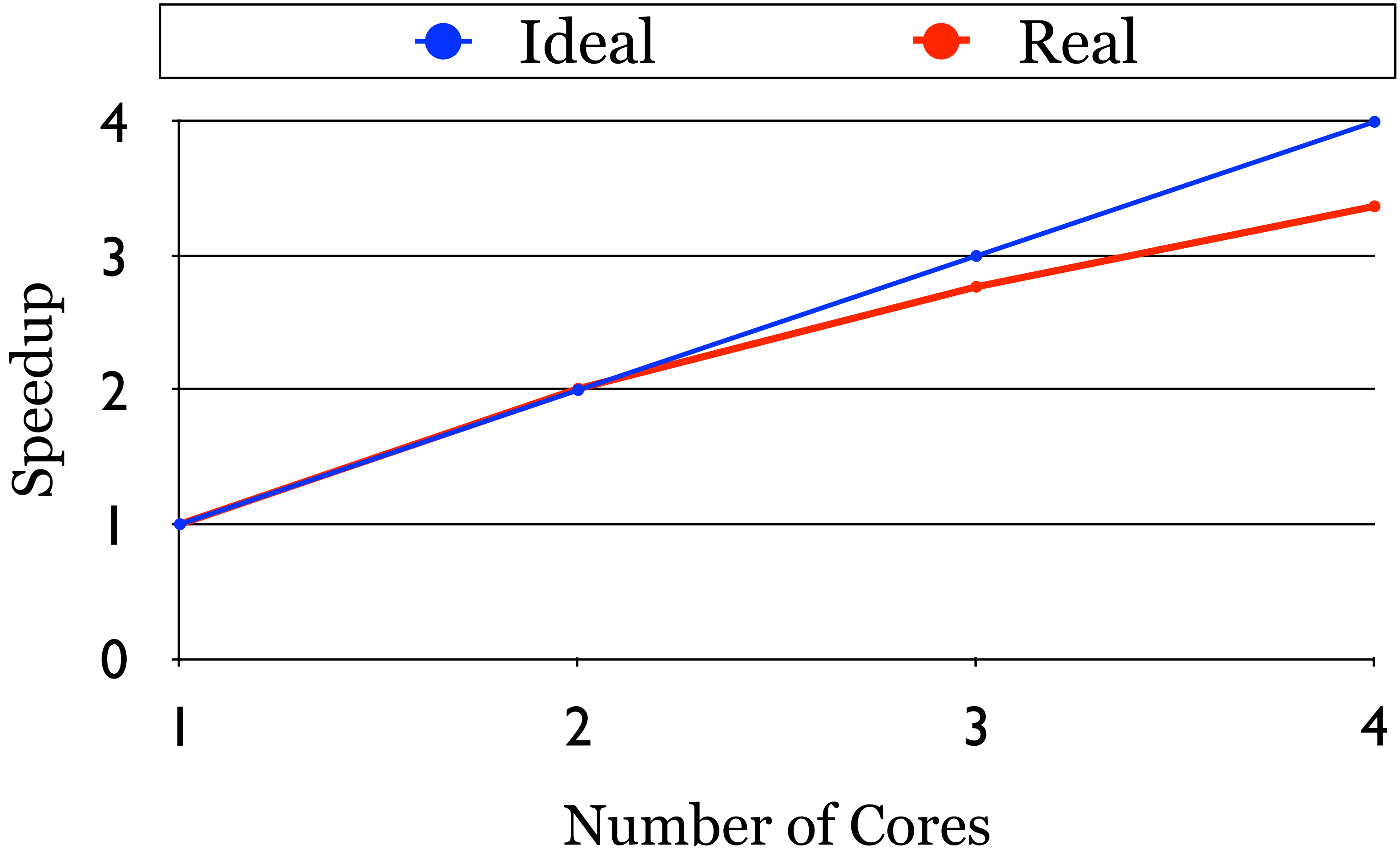
Speedup

- How much faster the parallel version is (compared to the sequential version).
- T_1: sequential time.
- T_P: time in parallel.

$$Speedup = \frac{T1}{T_P}$$



Parallel Stencil Speedup



Concluding Remarks

- MPI consists of a simple communication interface: `send`, `recv`, `bcast`, `reduce`, and more.
- MPI lets you write scalable code, from your desktop all the way to petascale systems.
- Parallelizing codes is an **art**:
  - Problem decomposition.
  - Communication strategy.

Acknowledgments

- Dr. Lisandro Dalcin for his help on understanding the internals of MPI4Py.

Thank you!
Q&A

http://coco.sam.pitt.edu/~emeneses/teaching/hpc-python

emeneses@pitt.edu
esteban.meneses@acm.org