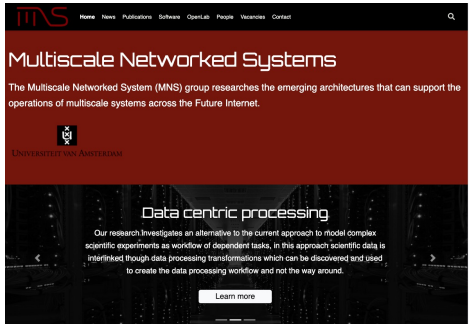
Who am I?



I'm a member of the MultiScale Networked Systems (MNS) of the informatics institute of the University of Amsterdam. In 2019, I joined the National eScience Center as a Technical Lead working on optimized data handling. My research interests are in scientific workflow management systems for e-Infrastructure.

- Keywords: Distributed and grid/cloud-based systems, workflows, Service Oriented Architecture

On-going Projects:

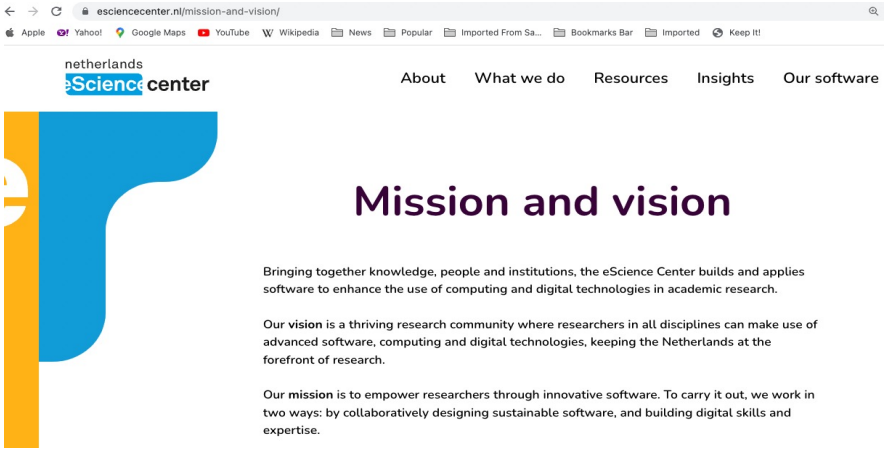
- DL4LD - Data Logistics 4 Logistic Data (2018-2023)
- EPI-Enabling Personal Interventions (2019-2023)

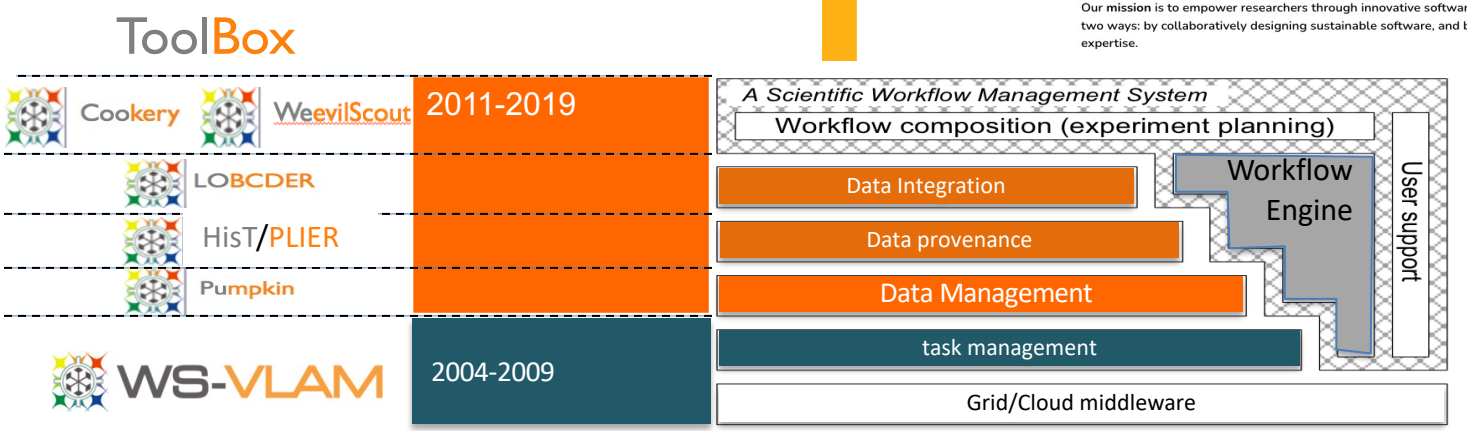
On-going education-related responsibilities:

- Coordinator Big Data Track of the Joined VU-UvA MSc Computer Science Program
- Coordinator HPC and Big Data course organised with Dutch National computer centers SURFsara
- Member of the Examinations Board - Software Engineering

Supervision:

- PhD students
- MSc students





Funding

What I will be covering in this course

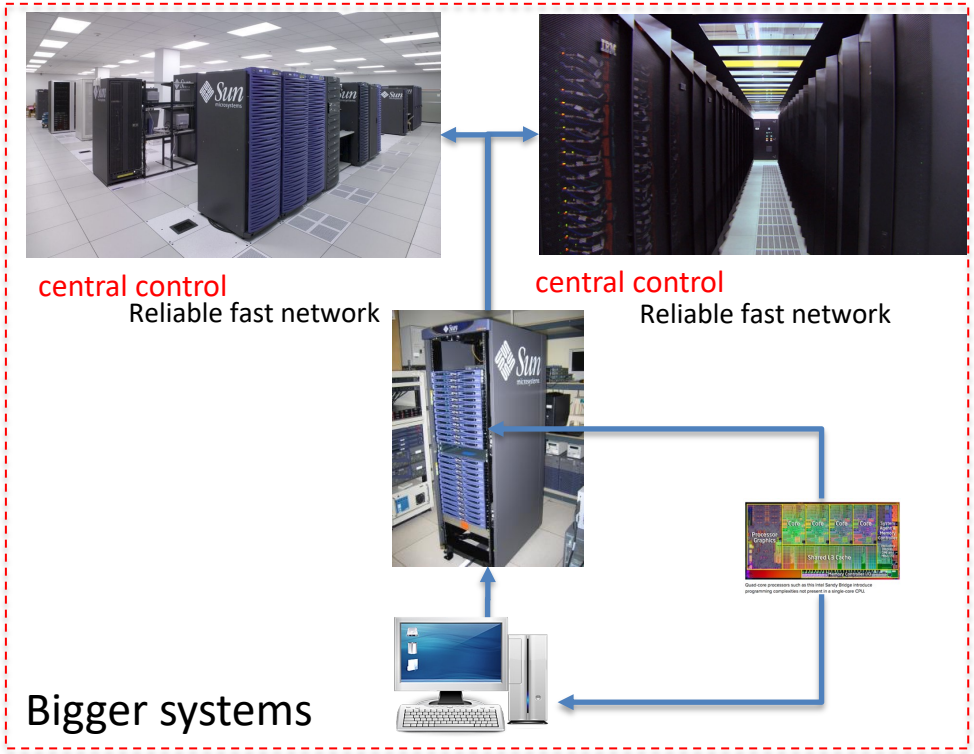
- Message passing Interface
- Big Data module
  - (Micro) Service Oriented Architecture
  - Virtualization (VM / Container)
  - Big Data Platform (MapReduce, Spark, HDFS)
- Application /Guest Lectures
  - Jason Maassen "Efficient computing at Netherlands eScience Center"
  -

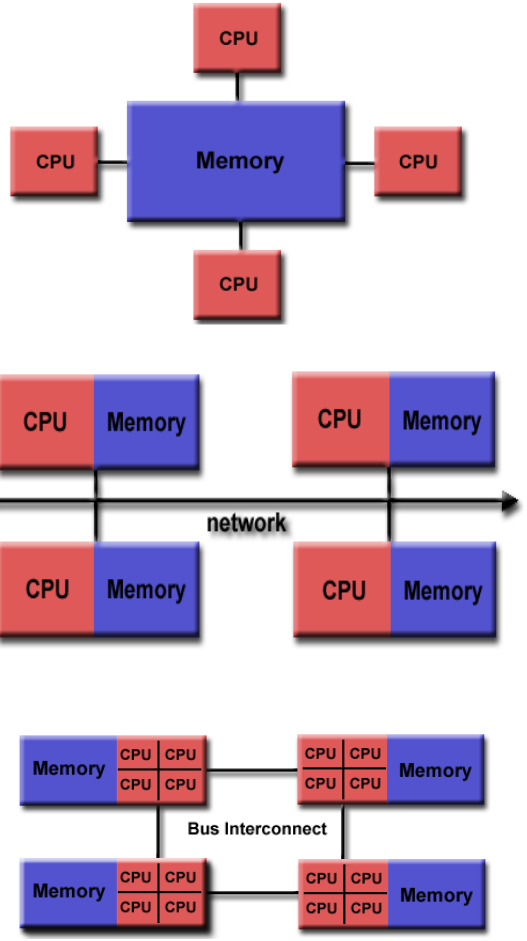
Message passing Interface

MPI

Adam Belloum

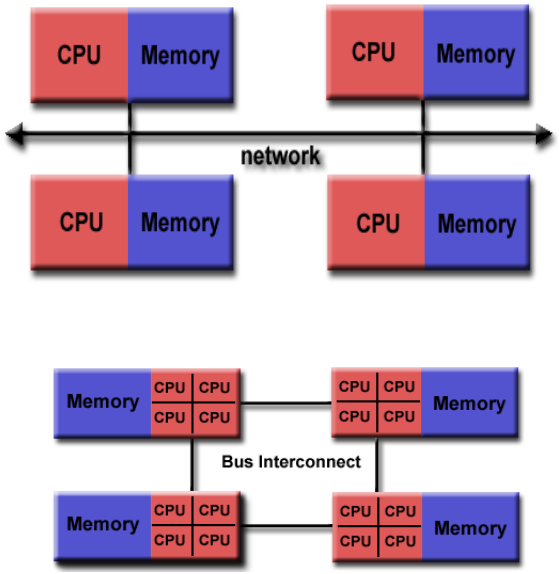
Computing Resources





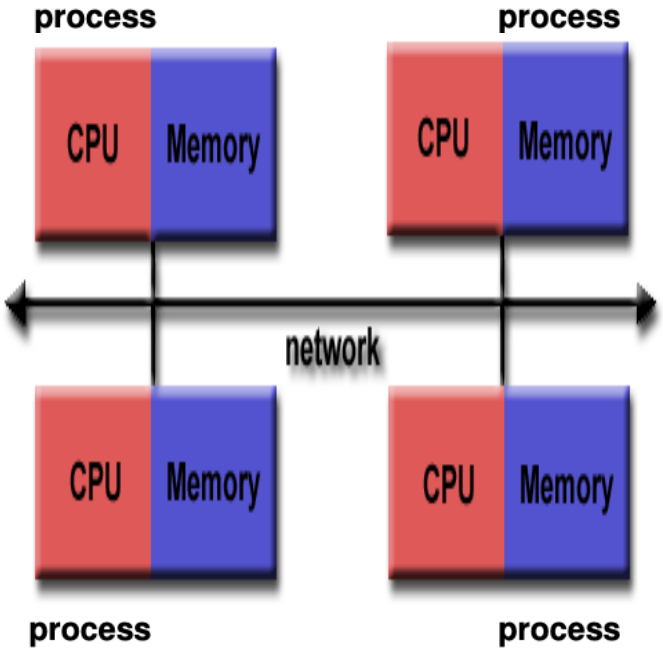
Content

- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchcroneous / Asynchroneous
  - Point-to-point
  - Collective
  - System Buffer
- Data Model
- MPI programs
- Examples



What is MPI

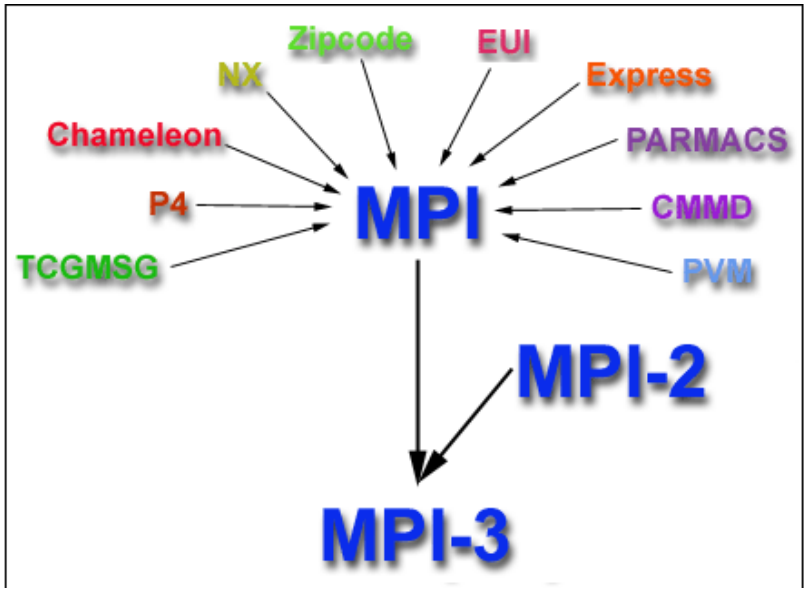
- [illegible]
- [illegible]
  - [illegible]
- [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
  - [illegible]


process
process
CPU
Memory
CPU
Memory
network
CPU
Memory
CPU
Memory
process
process

[illegible]

MPI Since 1992

- April 1992:
  - Preliminary draft proposal for message passing
- November 1992:
  - MPI Forum founded: over 80 individuals from 40 organizations (hw/sw vendors, academia, application scientists).
- November 1993:
  - Draft MPI standard presented
- **May 1994**:
  - Final version released: MPI-1
- June 1995:
  - Version 1.1 | minor corrections
  - MPI Forum starts work on MPI-2
- **July 1997:**
  - MPI-2 standard released
- **September 2012:**
  - MPI-3 standard released
- **November , 2020**
  - Draft of MPI 4.0



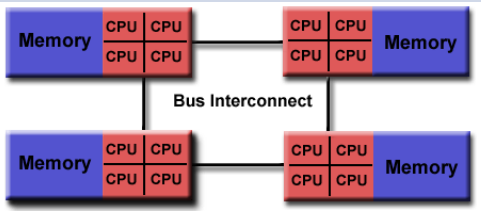
Many different MPI implementations exist !

Free implementations: LamMPI, MPICH, OpenMPI.

MPI-2

- MPI-2 was a major revision to MPI-1 adding new functionality and corrections

Key areas of new functionality in MPI-2	
Dynamic Processes	**extensions** that remove the static process model of MPI. Provides routines to create new processes after job startup
One-Sided Communications	provides routines for one directional communications. Include shared memory operations (**put**/**get**) and remote accumulate operations
Extended Collective Operations	allows for the application of collective operations to inter-communicators
External Interfaces	defines routines that allow developers to layer on top of MPI, such as for **debuggers and profilers**.
Additional Language Bindings	describes C++ bindings and discusses Fortran-90 issues
Parallel I/O	describes MPI support for **parallel I/O**.



MPI-3

The MPI-3 standard was adopted in 2012

Key areas of new functionality in MPI-3	
Non blocking Collective Operations	permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
New One-sided Operations	to better handle **different memory models**.
Neighborhood Collectives	**extends** the distributed graph and Cartesian process topologies with additional communication power.
Fortran 2008 Bindings	**expanded** from Fortran90 bindings
MPIT Tool Interface	allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely **performance tools**).
Matched Probe	**fixes** an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment

Language bindings

- MPI defines the syntax and semantics of a library routines to a wide range of users writing portable message passing programs in Fortran or C

- However new language bindings have been developed
  - Python: pyMPI [1], Mpi4py [2], Pypar [3], MYMPI, [4], MPI submodule [5]
  - Java: mpiJava, [6]
  - R: Rmpi [7], pbdMPI [8]

1. http://www.hpjava.org/mpiJava.html
2. https://sourceforge.net/projects/pympi/
3. https://bitbucket.org/mpi4py/mpi4py
4. https://code.google.com/archive/p/pypar/
5. https://mympi.co.uk
6. ScientificPython
7. https://cran.r-project.org/package=Rmpi
8. https://cran.r-project.org/package=pbdMPI

Documentation and Books on MPI

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books on MPI and MPI-2:
  - MPI: The Complete Reference, volumes 1 and 2, MIT Press, 1999.
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition), by Gropp, Lusk, and Skjellum, MIT Press, 1999.
  - Using MPI-2: Extending the Message-Passing Interface, by Gropp, Lusk, and Thakur, MIT Press, 1999
- Other information on Web:
  - https://hpc-tutorials.llnl.gov/mpi/
  - https://mpitutorial.com/tutorials/

Content

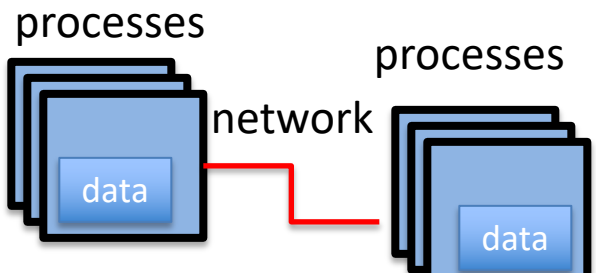
- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchcroneous / Asynchroneous
  - Point-to-point
  - Collective
  - One-side
  - System Buffer
- Data Model
- MPI program
- Examples

MPI programming model

- MPI is an application programming interface (API) for communication between separate processes
  - Processes do not interfere with each other

- MPI enable processes to communicate
  - by moving the data in a form of messages from the address space of one process to another process.

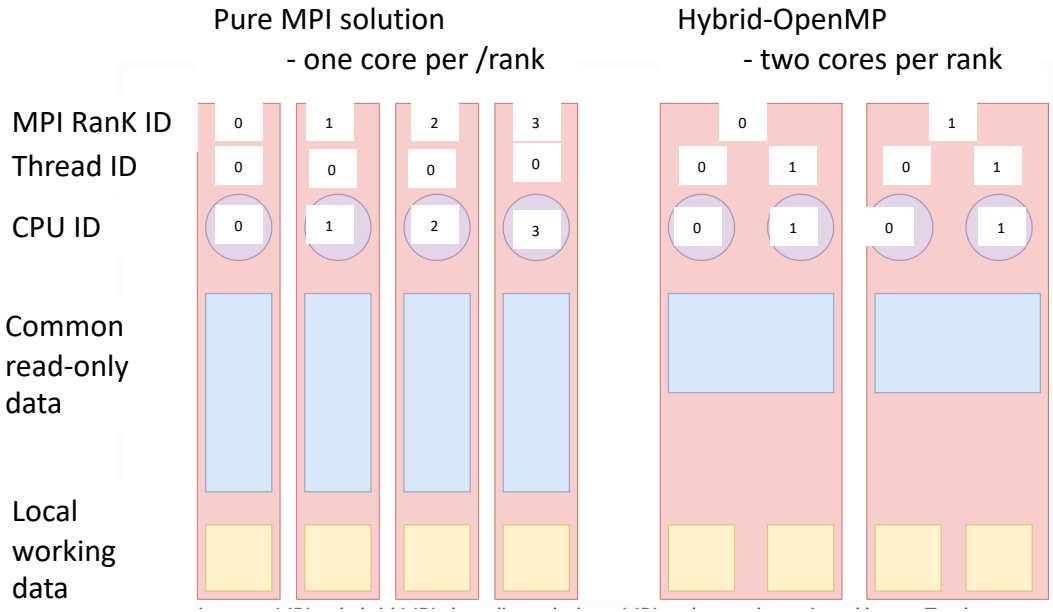
This way MPI help to create a **parallel program** which implement message-passing parallel programming model



MPI Programing model

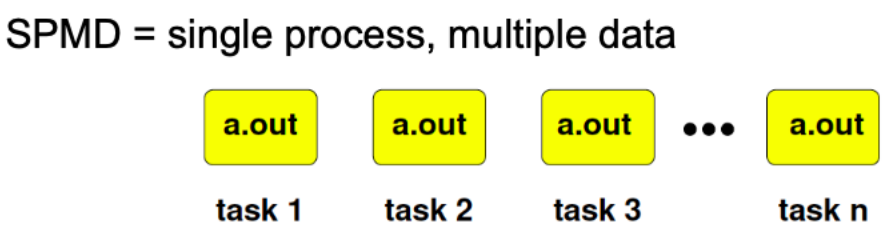
- Originally, MPI was designed for **distributed memory architectures**, 1990s.
- Later MPI was adapted to handle both **distributed, shared/hybrid memory architectures** seamlessly.
- MPI handles different interconnects and protocols.

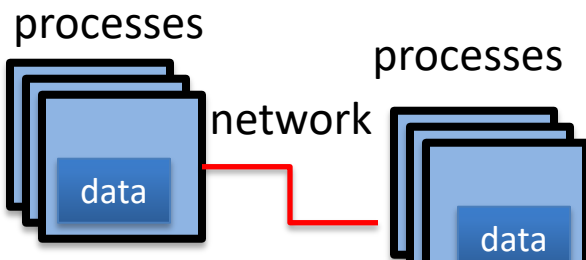
The programming model remains a distributed memory model however, regardless of the underlying physical architecture of the machine.



Execution Model

- Parallel program **is launched as set** of **independent**, **identical** processes
  - The same program code/instructions can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  - `mpirun, mpiexec, srun, aprun, …`





Message Passing Interface

- MPI is an application programming interface (API) for communication between separate processes
  (The most used approach for distributed parallel computing)
- MPI programs are portable and scalable
- MPI is flexible and comprehensive
  - Large (over 120 routines in MPI, …)
  - Concise (often only 6 procedures are needed)
- MPI standardization by MPI Forum [1]

[1] http://www.mpi-forum.org

Content

- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchcroneous / Asynchroneous
  - Point-to-point
  - Collective
  - One-side
  - System Buffer
- Data Model
- MPI Programs
- Examples

Naming Convention

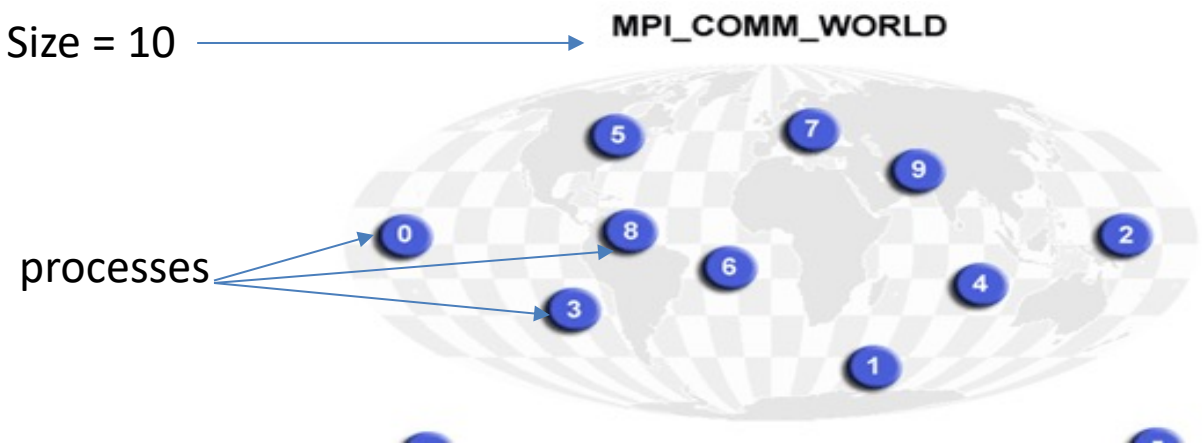
```
MPI_Xysend(parameter,...)
MPI_Xyrecv(parameter,...)

Where X = {I, S, B, …}
```

MPI Communicator

– Communicator is an object connecting a group of processes

– Initially, there is always a communicator MPI_COMM_WORLD which contains all the processes

– Most MPI routines require communicator as an argument



Routines of the MPI Library

- Environment Management Routines
- Communication Routines
  - send/receive routines
  - Point-to-point and Collective
  - Communication
- Synchronisation Routines

- MPI routines require at least information about:
  - The communicator
  - number of processes
  - rank of the process
  - Communicating processes (sending and receiving messages)

Five MPI Commands

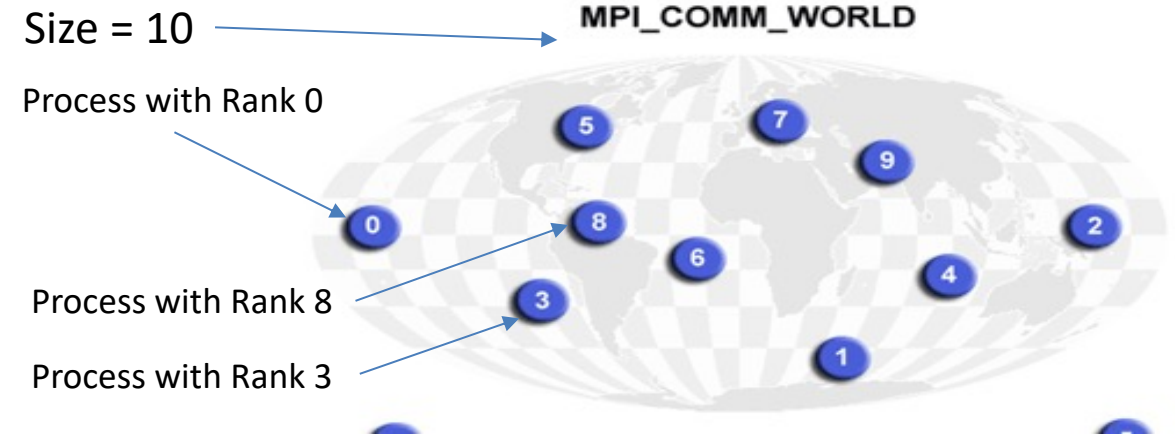
- Set up the MPI environment
  - `MPI_Init()`
- Information about the communicator
  - `MPI_Comm_size(comm, size)`
  - `MPI_Comm_rank(comm, rank)`
- Finalize MPI environment
  - `MPI_Finalize()`

MPI- Rank

- MPI runtime assigns each process a rank
  - identification of the processes
  - ranks start from **0** and extent to **N-1**

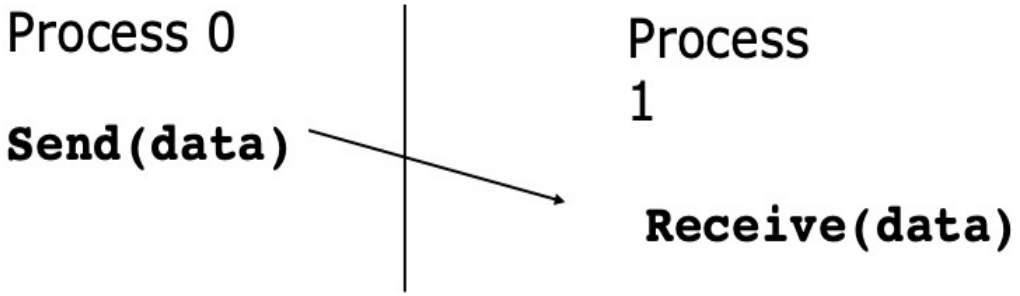
- Processes can perform different tasks and handle different data based on their rank

```
...
if ( rank == 0 ) {
   ...
   }
if ( rank == 1) {
   ...
   }
...
```

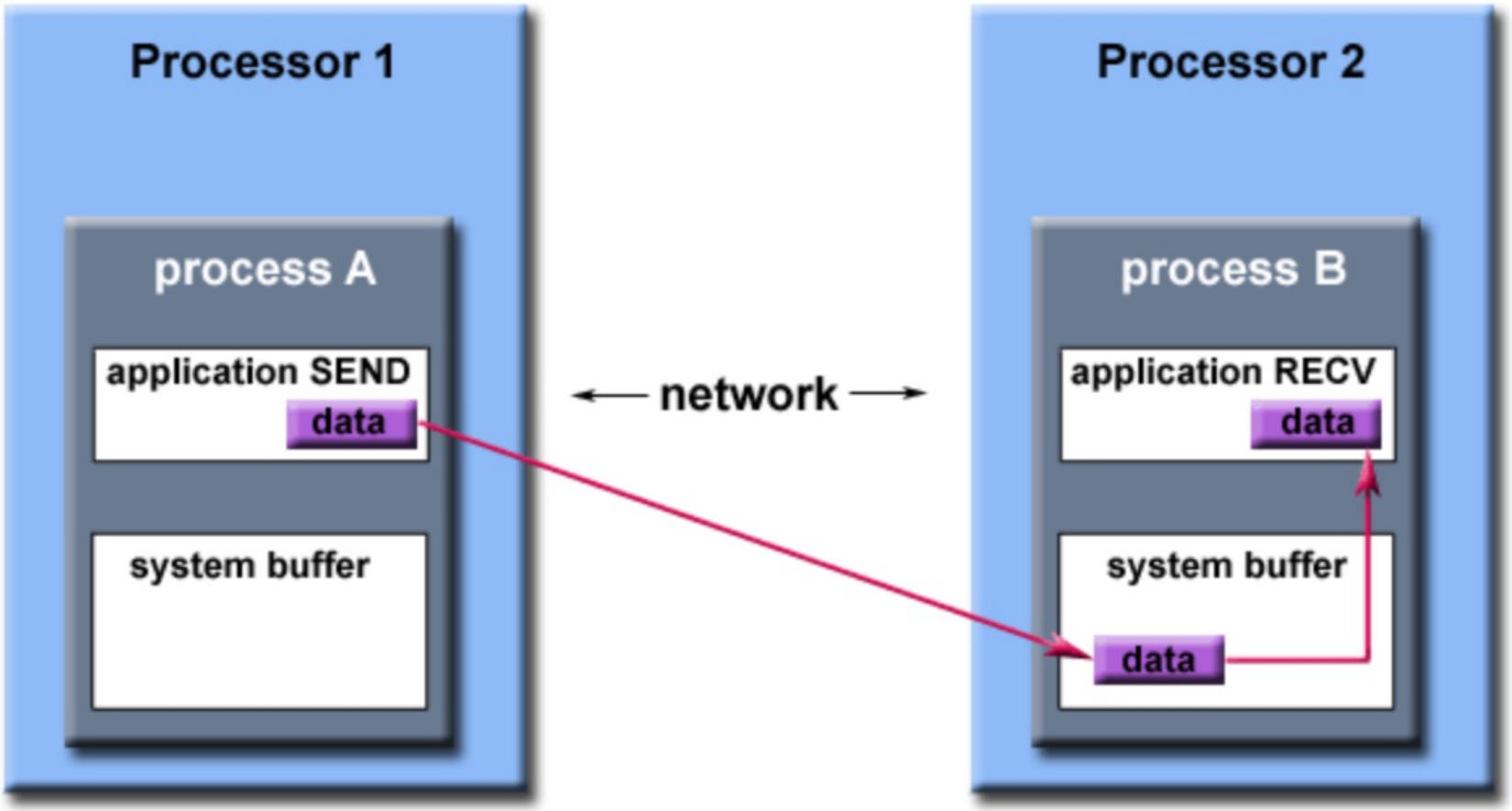


Cooperative Operations

- The message-passing approach makes the exchange of data cooperative
  - Data is explicitly send by one process and received by another
    - any change on the receiving process mem is made with receiver's explicit participation
  - Communication and Synchronization are combined



Cooperative Operations (Buffering)



MPI standard permits the use of a system buffer but does not require it.

Synchronous vs. Asynchronous

- In a perfect world, every send operation would be perfectly synchronized with its matching receive This is rarely the case
  - A synchronous communication is not **complete** until the message has been received.
  - An asynchronous communication **completes** as soon as the message is on the way.

Blocking vs. Non-Blocking

- Blocking, means the program will not continue until the communication is completed.

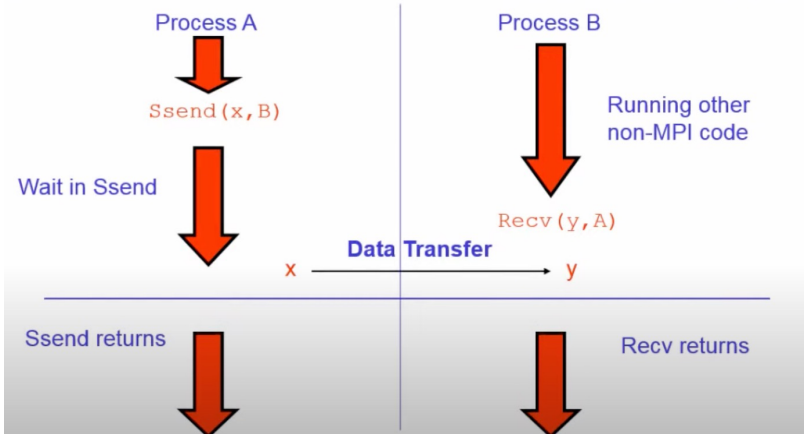
- Non-Blocking, means the program will continue, without waiting for the communication to be completed.
  - Split communication operation into initiation and completion.
  - Identify communication operation with "ID" (handle)

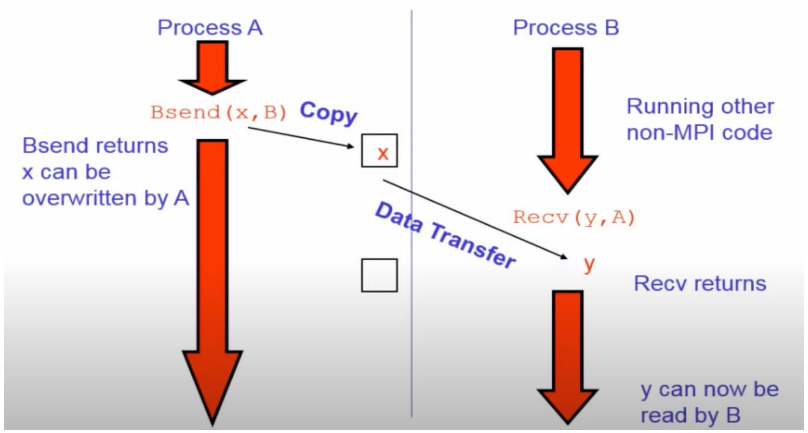
➢ **Overlap communication with computation.**

Modes for Sending

`MPI_Send`

	Blocking	Non-blocking
Synchronous Mode.	(`MPI_Ssend`)	(`MPI_ISsend`)
Buffered Mode	(`MPI_Bsend`)	(`MPI_IBsend`)
Ready Mode	(`MPI_Rsend`)	(`MPI_IRsend`)





Note:

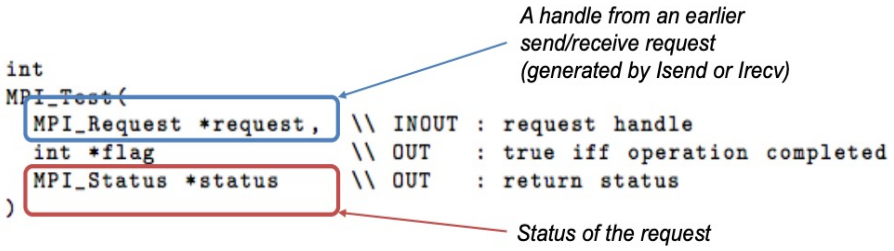
`MPI_Recv` receives messages sent in Any mode,

Blocking vs Non-blocking

Blocking	Non-blocking
Send "returns" after the data has left • It is safe to modify the application buffer for reuse • **Synchronious**: requires handchskeing with receive task • **Asynchronious** : a **system buffer** is used to hold the data for eventual delivery to the receive	**Send** and **Receive** return almost immediately • A simple request to the MPI library • Do not wait for any communication events to complete
Receive "returns" after the message has arrived and is ready for the use by the program	Overlap communication with computation.

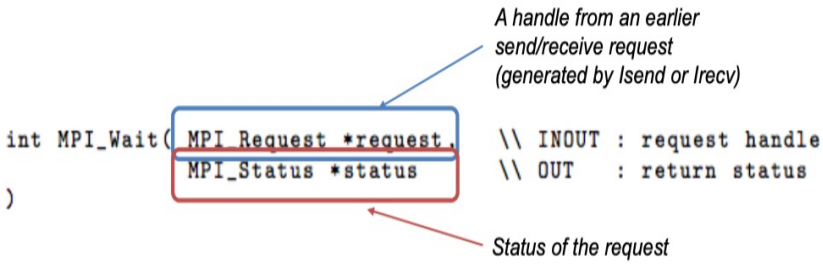
Non-blocking comm: Synchronisation Routines

Non-blocking comm: MPI_Test



- **Checks status** of non-blocking send or receive operation.
- Returns immediately ("non-blocking").
- Flag indicates completion status of operation.
  - If operation is completed:
    - sets request handle to MPI_REQUEST_NULL.
    - returns additional status data structure.
  - If operation is still pending, MPI_Test does nothing.

Non-blocking comm: MPI_Wait



- Finishes non-blocking send or receive operation.
- Does not return before communication is completed.
- Sets request handle to MPI_REQUEST_NULL.
- Returns additional status data structure.

What is printed?

```
// code P0
…
a= 100;
send (&a, P1);
a= 0;
```

```
// code P1
…
a= 99;
receive (&a, P1);
printf("%d\n", a)
```

What is printed?

- **Blocking** send / **Blocking** receive

P0 sends a=100, P1 updates its local a = 100, **prints 100.**

- **Blocking** send / Non-blocking receive

P0 sends a=100, P1 updates its local a = 100 or not , **prints 100 or 99**

- **Non-blocking** send / **Blocking** receive

P0 sends a=100 **or** a=0, P1 updates its local a = 100 **or** a = 0 , **prints 100 or 0**

- **Non-blocking** send / **Non-Blocking** receive

P0 sends a=100 **or** a=0, P1 updates its local a = 100 **or** a = 0 , **prints 100 or 0 or 99**

Overlapping computation and comm

- Observation: communication inhibit parallelism
  - Communication use network, adaptors, DMA controller,
  - Computation user cores, vector, vecto unit, float units, …
- Idea: exploit parallelism
  - Let the communication happend in the background
  - Run communication in paralllel with computation
- Implementation
  - Initiate message sending as soon as possibles
  - Provide receive buffer as soon as old data is no longer needed

The Best performance

Recommendations.

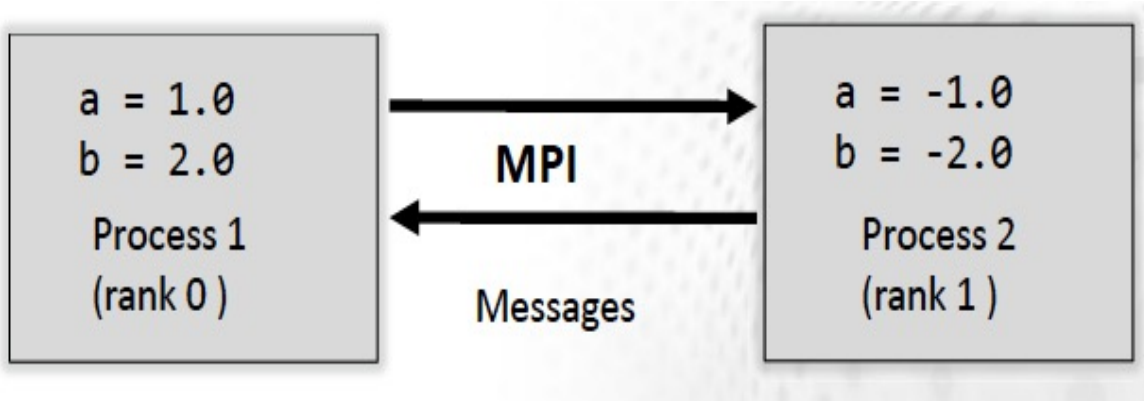
If you use	Consequence
`MPI_Ssend`	in that case, an MPI implementation can completely avoid **buffering data**
`MPI_Send`	It allows the MPI implementation the maximum flexibility in choosing how to deliver your data (**Unfortunately**, one vendor has chosen to have MPI_Send emphasize buffering over performance; on that system, `MPI_Ssend` may perform better.)
nonblocking routines	are necessary, then try to use `MPI_Isend` or `MPI_Irecv`. Use MPI_Bsend only when it is too inconvenient to use `MPI_Isend`
remaining routines,	`MPI_Rsend`, `MPI_Issend`, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

source: https://www.mcs.anl.gov/research/projects/mpi/sendmode.html

Point to point Communications

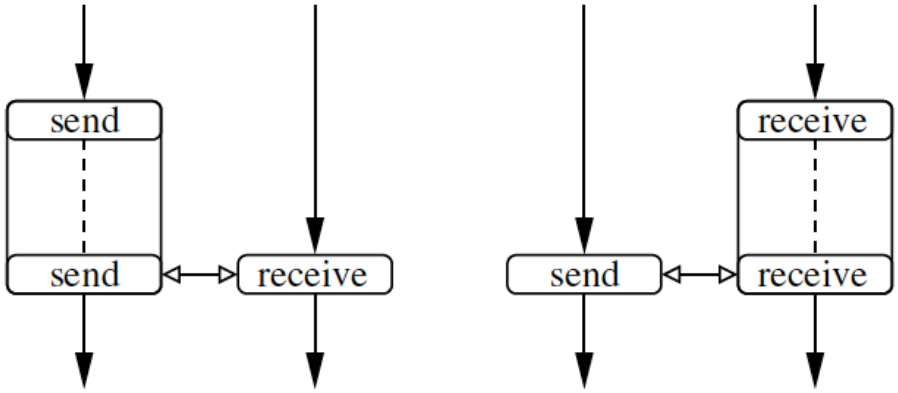
- MPI processes are independent, they communicate to coordinate work
- Point-to-point communication
  - Messages are sent between two processes

- ONE Sender
- ONE Receiver
- ONE Message

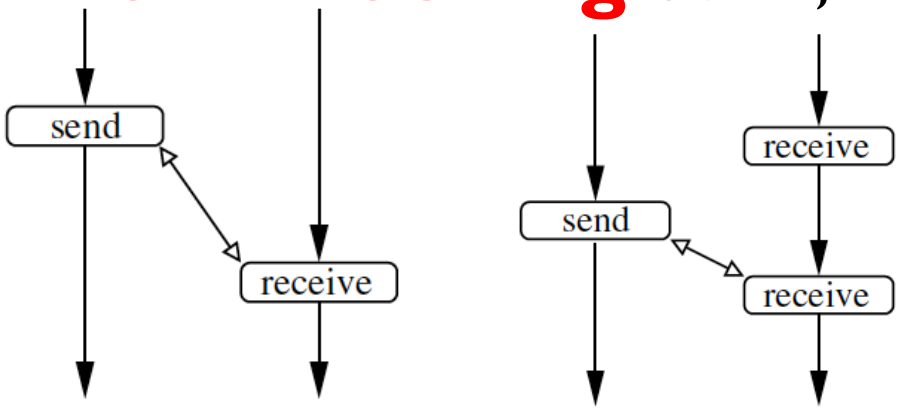


Point to point Communications

- **Blocking*** send, Blocking receive



- **Non-Blocking** send, -Non-Blocking receive



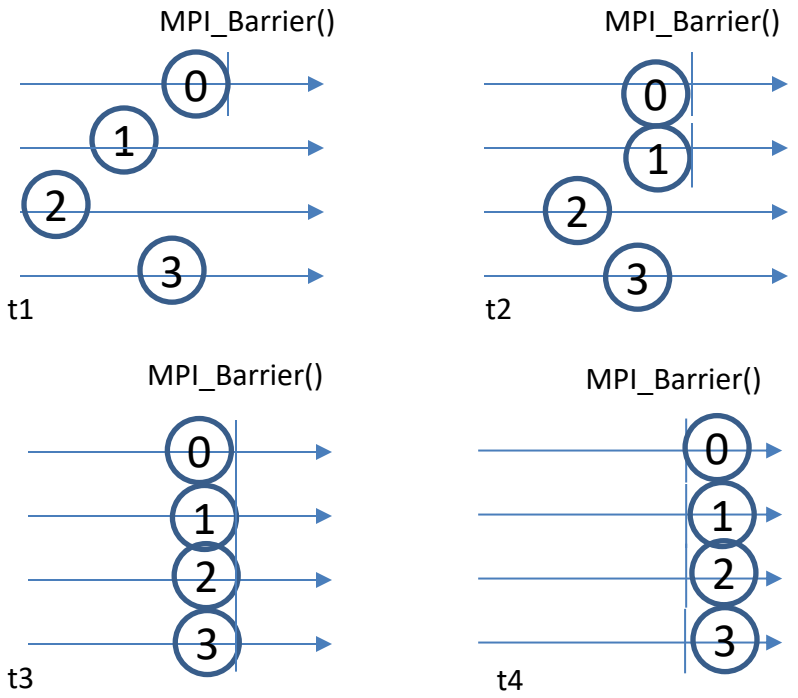
Collective operations in MPI

- Collective communication routines must involve **all** processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicators can be defined by the programmer.

- **Programming Considerations and Restrictions:**
  - Can only be used with MPI predefined datatypes - not with MPI Derived Data Types.
  - Collective communication routines do not take message tag arguments.
  - Collective operations on subsets of processes are accomplished
    - partitioning the processes into new groups and then attaching the new groups to new communicators.
- Note:
  - Unexpected behaviour can occur if even one task in the communicator doesn't participate.
    - It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.
  - MPI-2 extended collective operations to allow data movement between inter-communicators.
  - MPI-3, collective operations can be blocking or non-blocking.

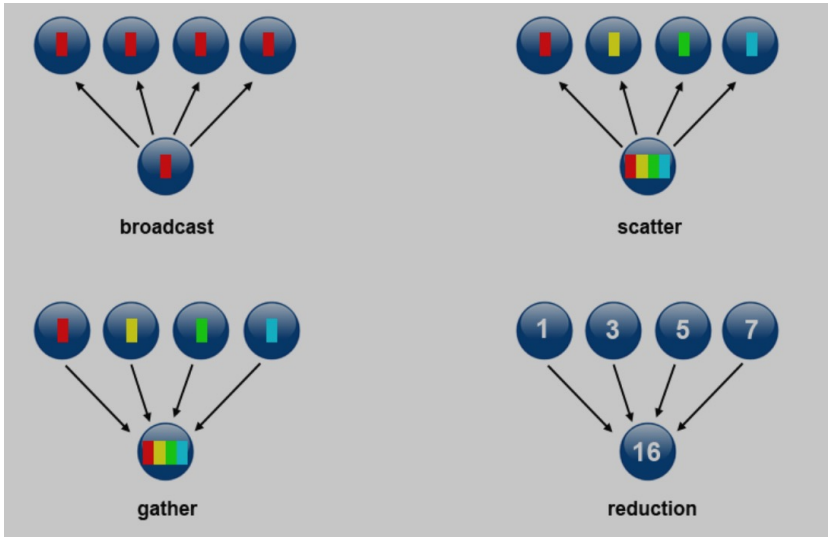
Collective operations in MPI

- Point to point
  - By careful message passing
- Collective
  - MPI_Barrier(comm)
    - Synchronization group of processes
    - All precesses block untill all have reached the barrier

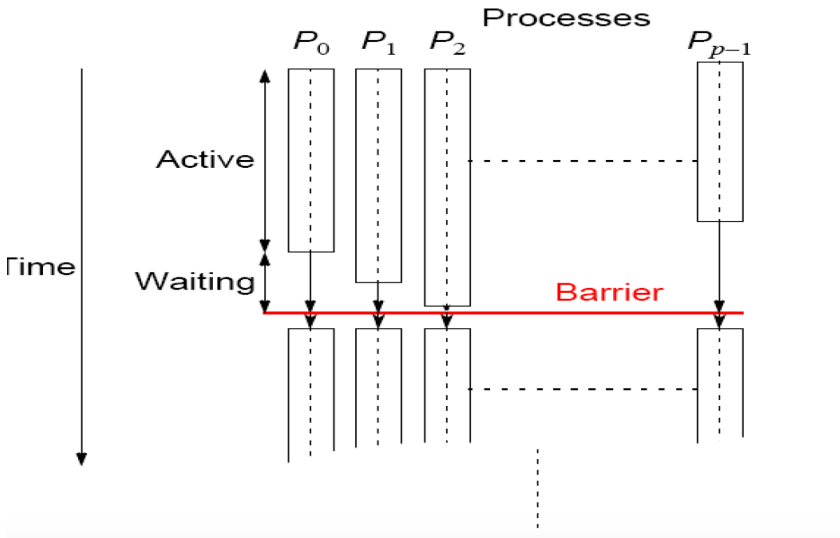


Types of Collective Operations:

- Data Movement
  - broadcast,
  - scatter/gather,
  - all to all.
- Example: reductions
  - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.).

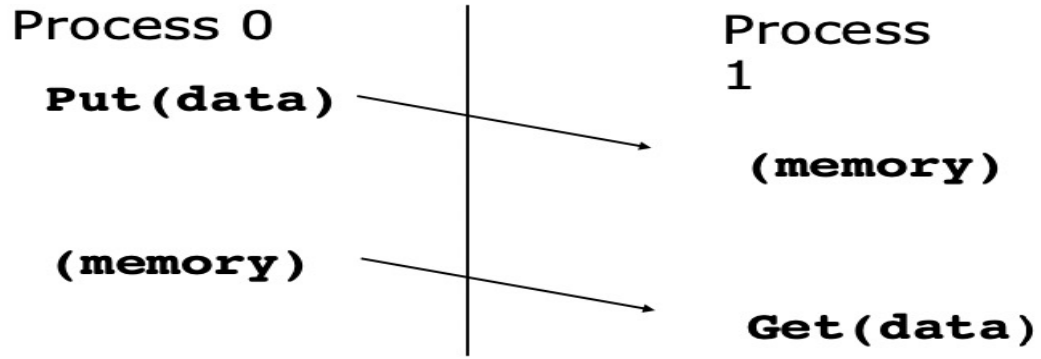


- Synchronization
  - processes wait until all members of the group have reached the synchronization point.



One-side operations

- Only one process needs explicitly praticipate
- One-side operation
  - Remote memory read / write
- An advantage
  - Communication and synchronization are decoupled



Source of Deadlocks

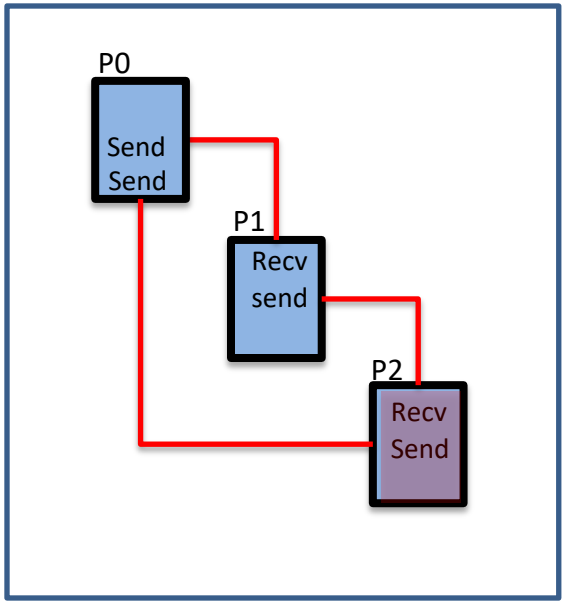
- Send **a large** message from process 0 to process 1
  - If there is insufficent storage at the destination, the send must wait for the user to provide the memory space
  - what happened with the code

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

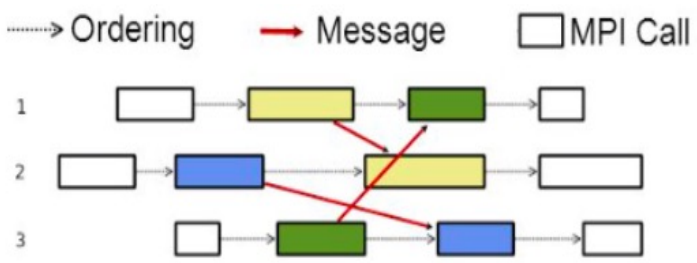
Note: The Send is called: unsafe because it depends on the availabity of system buffers

Message ordering

- The order of messages is preserved for :
  - ONE source
  - **and** ONE destination
  - **using** ONE communicator



- Is message ordering transitive ? NO !!



Point to point MPI routines

Point-to-Point Communication Routines			
MPI_Bsend	MPI_Bsend_init	MPI_Buffer_attach	MPI_Buffer_detach
MPI_Cancel	MPI_Get_count	MPI_Get_elements	MPI_Ibsend
MPI_Iprobe	MPI_Irecv	MPI_Irsend	MPI_Isend
MPI_Issend	MPI_Probe	MPI_Recv	MPI_Recv_init
MPI_Request_free	MPI_Rsend	MPI_Rsend_init	MPI_Send
MPI_Send_init	MPI_Sendrecv	MPI_Sendrecv_replace	MPI_Ssend
MPI_Ssend_init	MPI_Start	MPI_Startall	MPI_Test
MPI_Test_cancelled	MPI_Testall	MPI_Testany	MPI_Testsome
MPI_Wait	MPI_Waitall	MPI_Waitany	MPI_Waitsome

and many more ….

call convention

- blocking routine ➔ **MPI_Xxxx**(parameter,...)
- Non blocking routine ➔ **MPI_Ixxx**(parameter,...)

```
MPI_Send(...)  {  handle = MPI_Isend(...)
                  ...
                  MPI_Wait( handle, ...)

MPI_Recv(..)   {  handle = MPI_Irecv(...)
                  ...
                  MPI_Wait( handle, ...)
```

Collective MPI routines

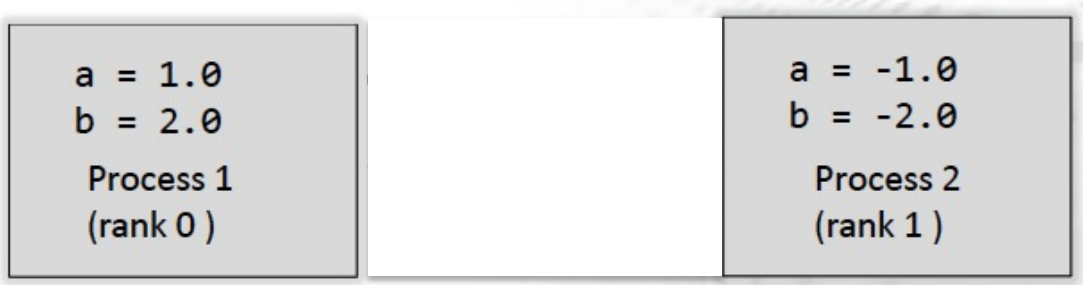
Collective Communication Routines			
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce	MPI_Alltoall
MPI_Alltoallv	MPI_Barrier	MPI_Bcast	MPI_Gather
MPI_Gatherv	MPI_Op_create	MPI_Op_free	MPI_Reduce
MPI_Reduce_scatter	MPI_Scan	MPI_Scatter	MPI_Scatterv

- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchcroneous / Asynchroneous
  - Point-to-point
  - Collective
  - One-side
  - System Buffer
- Data Model
- First MPI program

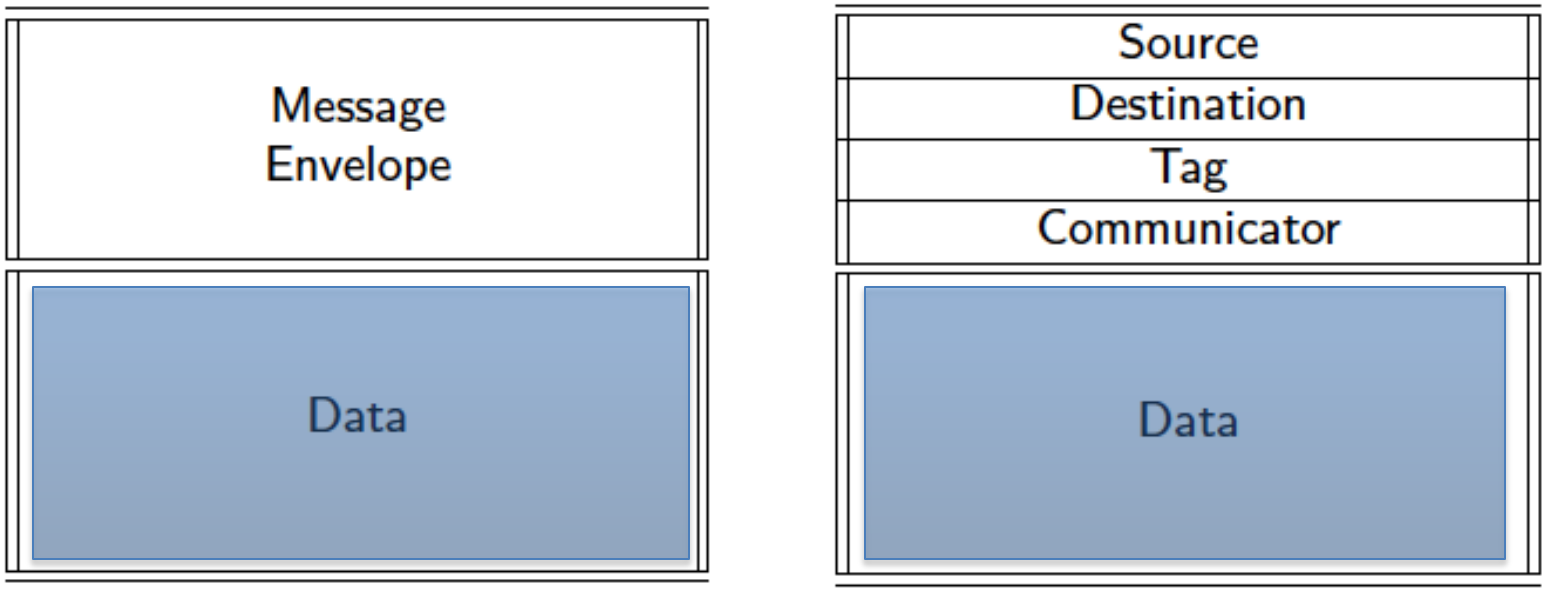
Data Model

- All variables and data structures are local to the process

- Processes can exchange data by sending and receiving **messages**



Message



- Message envelope:
  - Source: sender task id
  - Destination: receiver task id
  - Tag: Number to distinguish different categories of messages

Message - Data

- Data Type
  - For reasons of portability, MPI predefines its **elementary** data types.
  - Programmers may also create their **own data types** (Derived Data Types).
- Type Matching
  - Sender: Variable type must match MPI type.
  - Transfer: MPI send type must match MPI receive type.
  - Receiver: MPI type must match variable type.

MPI datatype	C datatype
MPI_CHAR	char
MPI_SIGNED_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	int
MPI_UNSIGNED unsigned	int
MPI_LONG	long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
…	

```
char buf[100];
MPI_Send(buf, 10, MPI_BYTE, dest, tag, comm);
```
mismatch

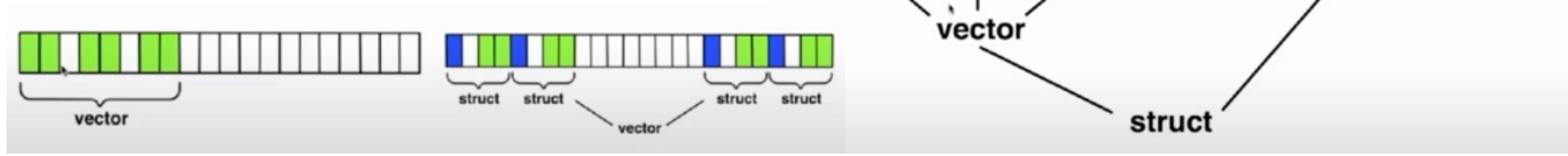
```
long buf[100];
MPI_Recv(buf, 10, MPI_INT, source, tag, comm, status);
```
mismatch

```
MPI_Send(buf, 10, MPI_INT, 1, tag, comm);
MPI_Recv(buf, 40, MPI_BYTE, 0, tag, comm, status);
```
mismatch

Message - Data

- Derived Data Types
  - MPI provides facilities to define own data structures based upon sequences of the MPI primitive data types
- MPI provides several methods for constructing **derived data types**:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Derived Data Type Routines



- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchronous / Asynchronous
  - Point-to-point
  - Collective
  - One-side
  - System Buffer
- Data Model
- MPI program
- Examples

Writing MPI Programing

Steps for programming with MPI

Desing the parallel application

- Data distribution
- Work per (worker)thread
  - symmetric or not
- Define the role of the master process
  - Distributed the work
  - Can/Cannot participate in the computation
- Write the code for one process
  - Differentiate functionality based on rank
  - Map global indices to local indices

Challenges:

- Domain decomposition must be explict
- There is NO shared memory …
- what happened with data that need sharing

Design the parallel computation

Model of paralle computation

- Define tasks and data interactions

- Examples
  - Data parallelism
    - Potential Trivailly /embarrassingly parallel
  - Task parallelism
    - Fork-join is a mix of data and task parallelism
  - Farmer/worker
  - Devide and conquer
  - Bulk synchronous

Structure of MPI program

Include MPI header files

```
#include <mpi.h>
```

Call `MPI_Init()`

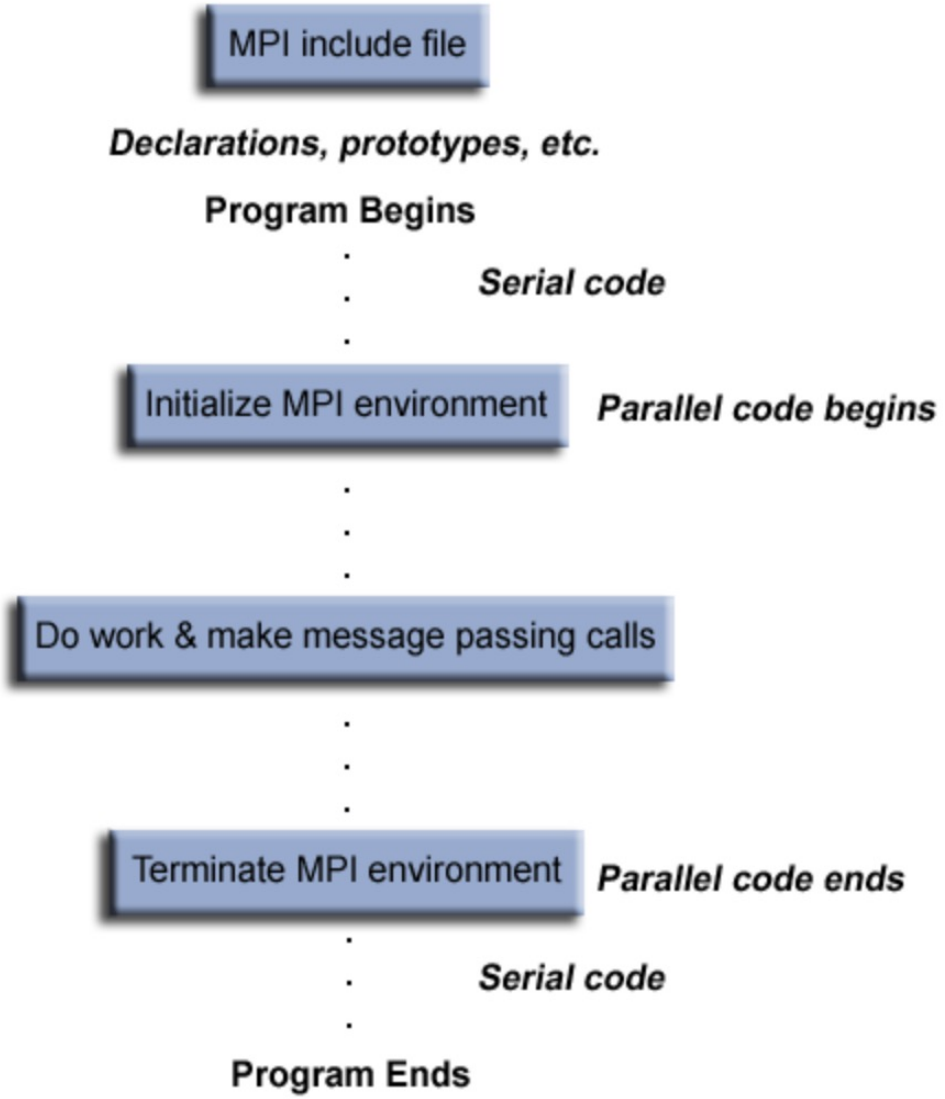
…

<Write the actual program>

….

Call `MPI_Finalize()`

before exiting from the main program



First MPI program

- Write a Hello word in MPI which prints the MPI tasks (processes) and their respective rank

```
1.  #include "mpi.h"
2.  int maint (int argc, char *argv[]){
3.    int rc, num_tasks,  my_rank;
4.    rc = MPI_Int (&argc, &argv);                  //initialize MPI runtime
5.    if (rc != MPI_SUCCESS){.                      //check if MPI runtime
6.         printf("Unable to set up MPI \n");
7.         MPI_Abort(MPI_COMM_WORLD, rc);          //Abort MPI runtime
8.    }
9.    MPI_COMM_size(MPI_COMM_WORLD, &num_tasks);//How many?
10.   MPI_COMM_rank(MPI_COMM_WORLD, &my_rank);  //Who am I?
11.   printf ('hello world!-I'm task \
12.           %d pf %d task \n" my rank, \
13.           num_taks);
14.    MPI_Finalize ()                              //Shutdown MPI runtime
15. }
```

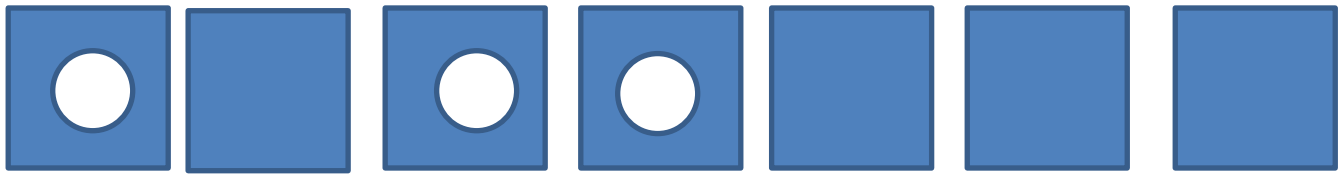
Compile and run

- Compile: `mpicc -o hello-world hello-world.c`
  - Compiler wrapper that does some additional operations before/after compiling the code
- Run: `mpirun -n 4 hello_world`
- Result:

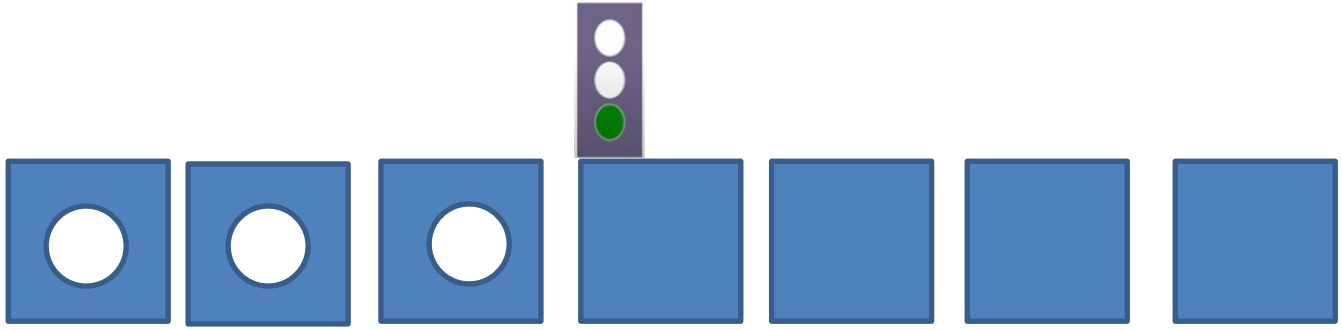
```
Hello World - I'm task 3 of 4 tasks
Hello World - I'm task 0 of 4 tasks
Hello World - I'm task 2 of 4 tasks
Hello World - I'm task 1 of 4 tasks
```

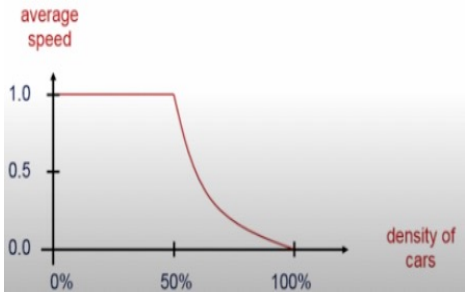
- Introduction to MPI
- Programming Model
- MPI routines
  - Blocking/ Non-blocking
  - Synchronous / Asynchronous
  - Point-to-point
  - Collective
  - One-side
  - System Buffer
- Data Model
- MPI program
- Examples (Think parallel)

Simple Traffic model

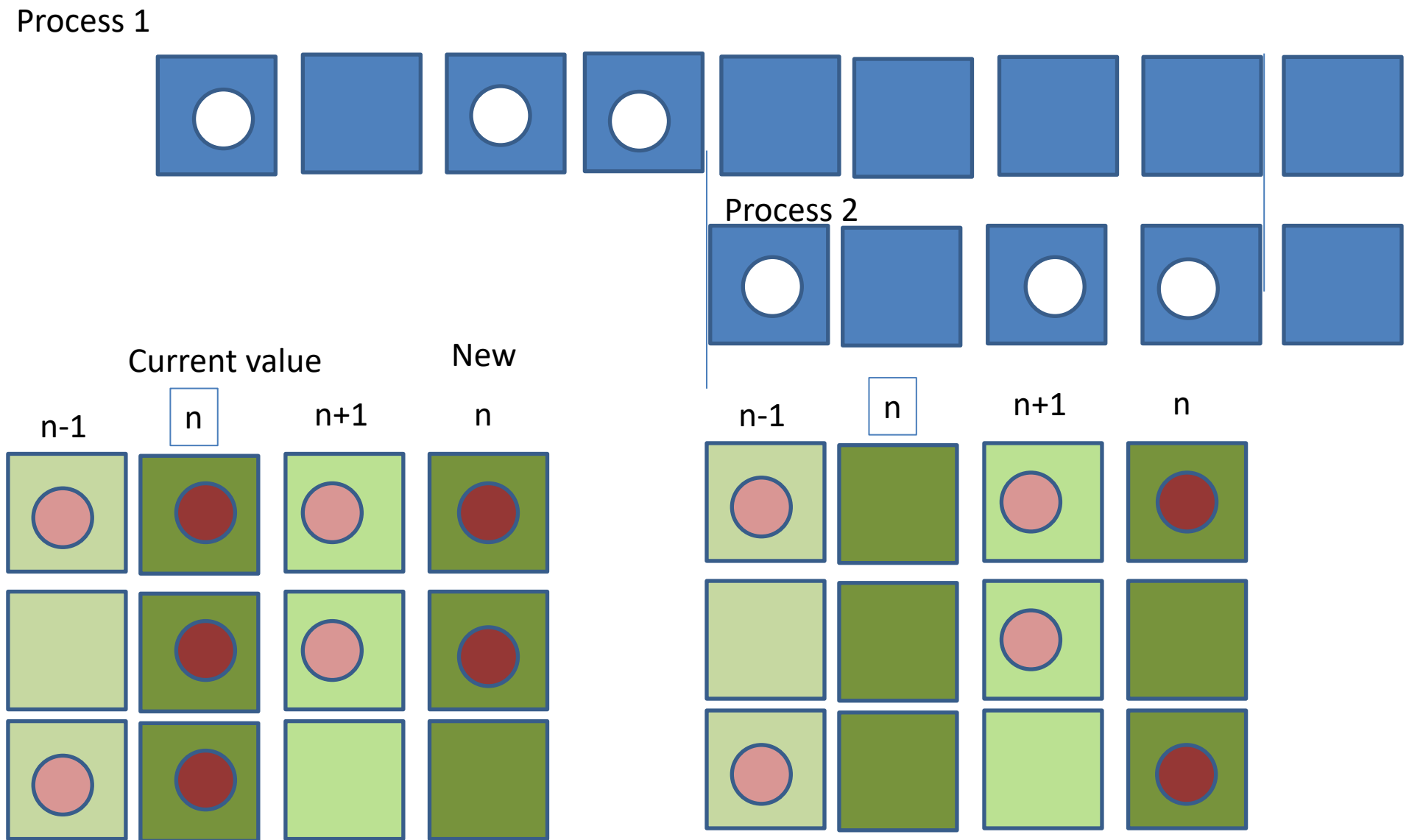


- Each cell can be
  - Occupied (1)
  - Non-occupied (0)
- Update rule
  - Instantaneously (all the car move at the same time)

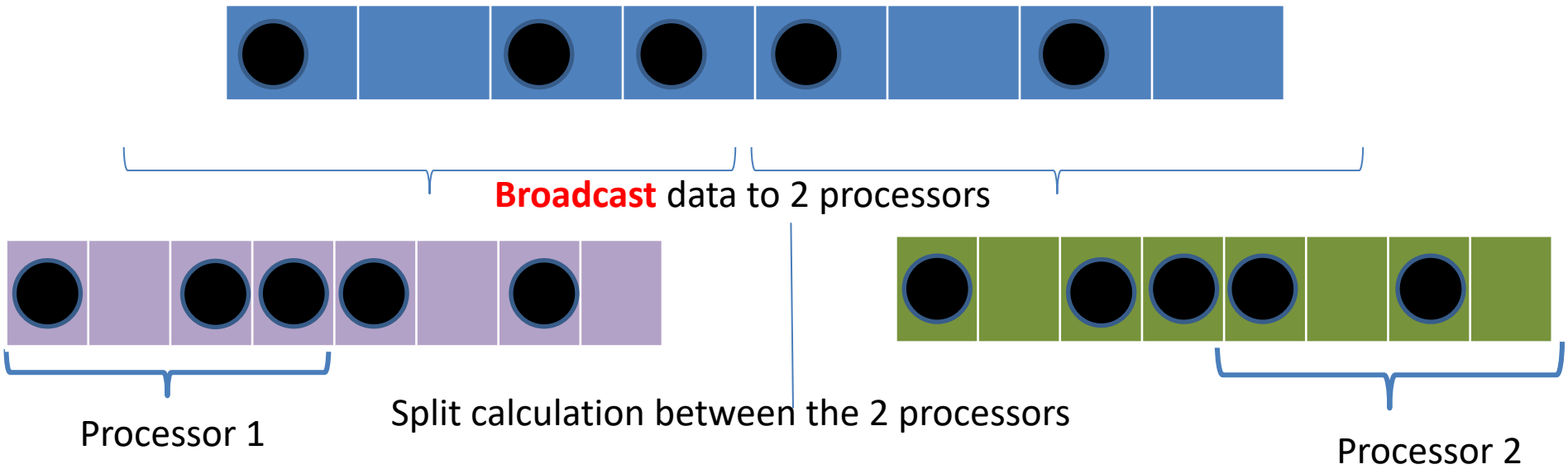




Simple Traffic model

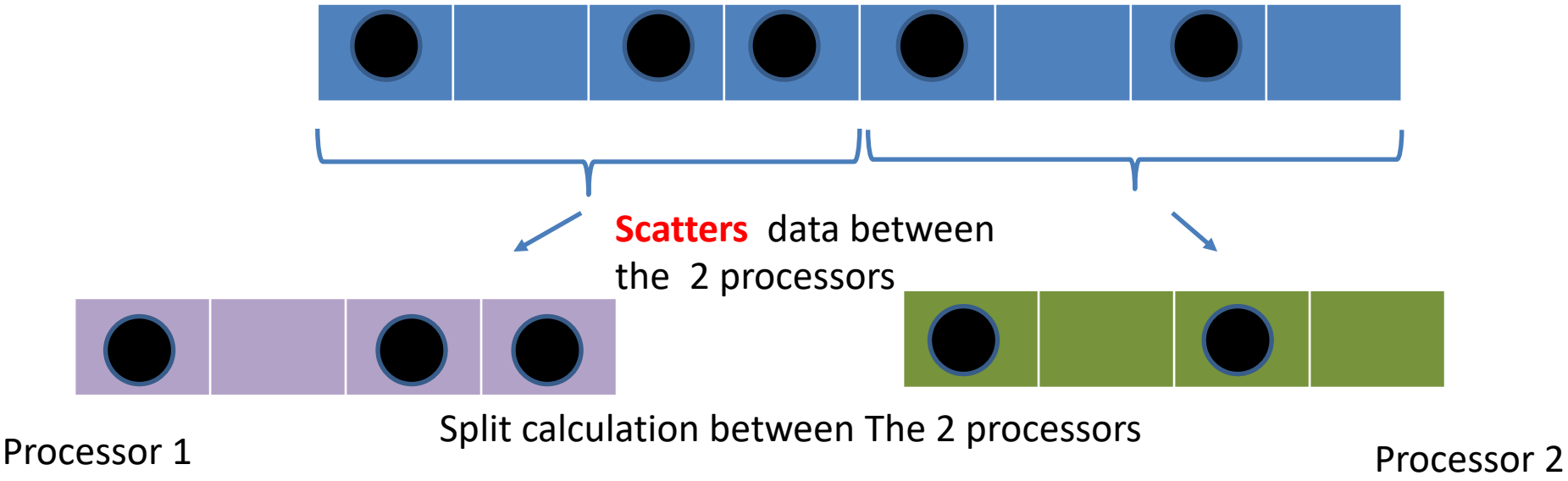


Solution 1



- **Globally resynchronise** all data after each move
  - A replicated data strategy
- Every Process stores the entire state of the calculation
  - Any process can compute the total number of moves

Solution 2



- Every must know which part of the data it is updating
- Synchronize at completion of each iteration and obtain that number of moves

Halo cells

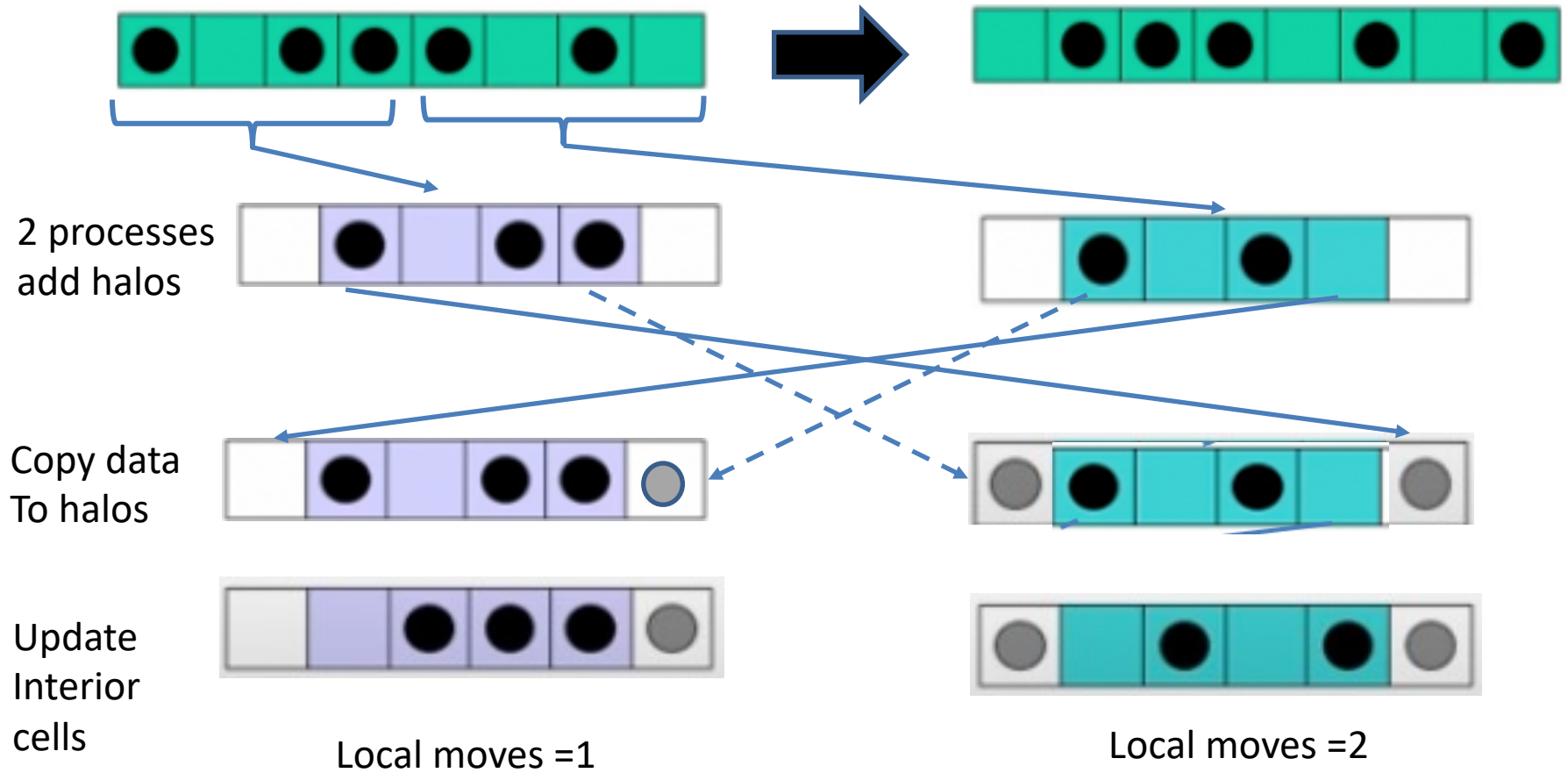
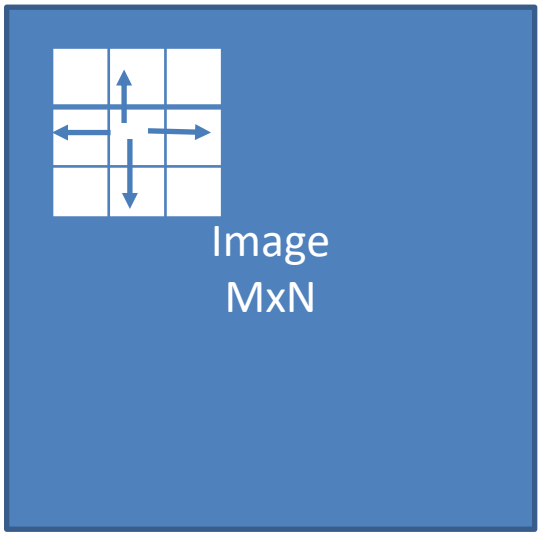

2 processes
add halos
Copy data
To halos
Update
Interior
cells
Local moves =1
Local moves =2

Image processing use case ..

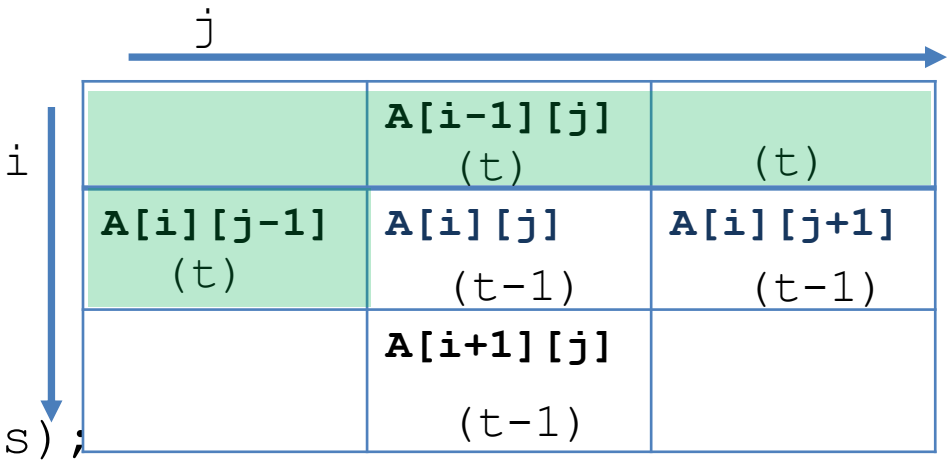


What is the best split ?

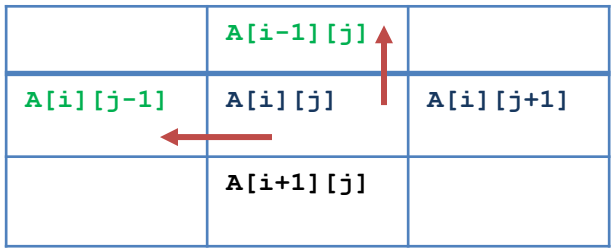
2-d array

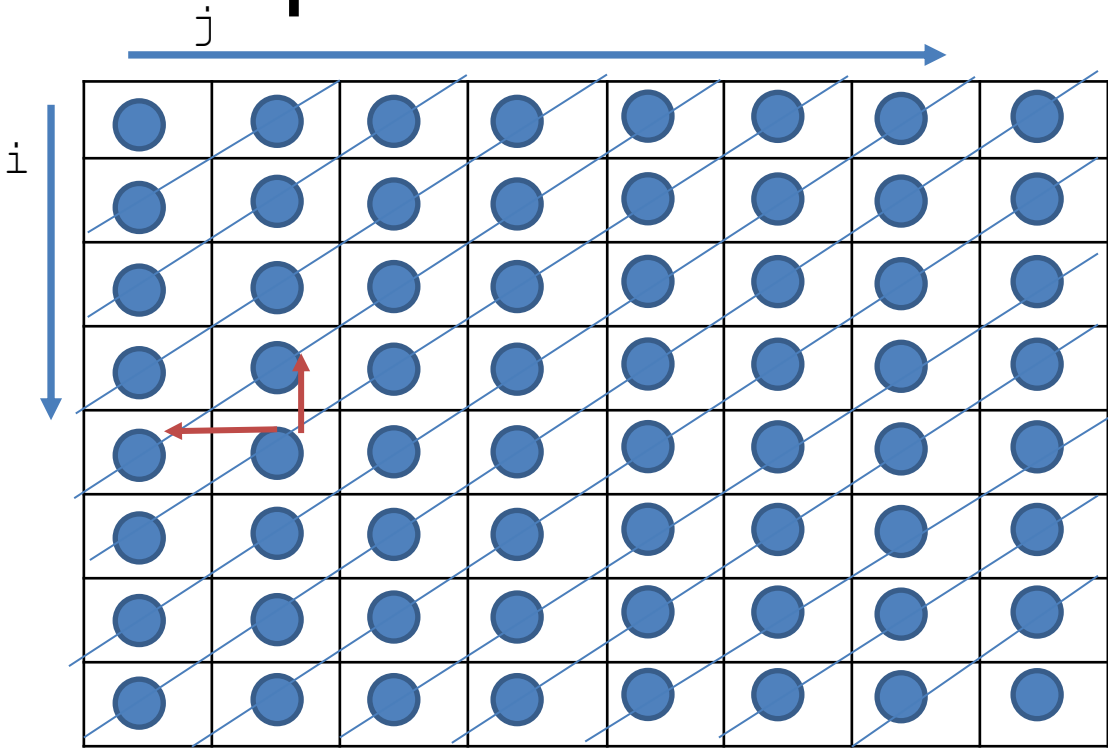
- Given a 2-d array of float values, repeatedly average each element with its immediate neighbours until the difference between the two iteration is less than some tolerance value

```
…
diff=0.0
tolerance
While(diff > tolerance)
for (i=0; i < n; i++)
  for (j=0; j < n; j++)
    temp = A[i][j]
    A[i][j]= average(neighbours);
    diff  += abs(A[i][j] – temp);
  }
}
```



Which domain decomposition is best?



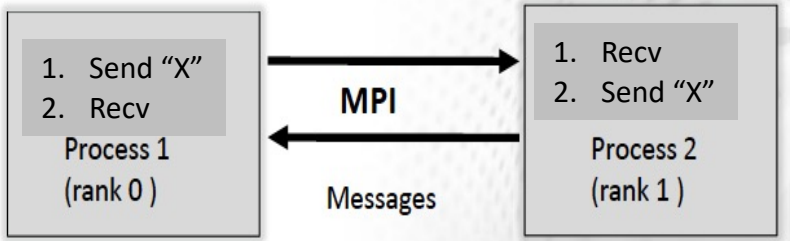


Decomposition	Parallelism	synchronisation
1-per-compute node.	$n^{2.}$	north, west
Anti-Diagonal.	2n-1.	One anti-diagonal

Point to point Communications

Blocking Routines

```
1   #include "mpi.h"
2   #include <stdio.h>
3
4   main(int argc, char *argv[])  {
5   int numtasks, rank, dest, source, rc, count, tag=1;
6   char inmsg, outmsg='x';
7   MPI_Status Stat;   // required variable for receive routines
8
9   MPI_Init(&argc,&argv);
10  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
11  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13  // task 0 sends to task 1 and waits to receive a return message
14  if (rank == 0) {
15    dest = 1;
16    source = 1;
17    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
18    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
19    }
20
21  // task 1 waits for task 0 message then returns a message
22  else if (rank == 1) {
23    dest = 0;
24    source = 0;
25    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
26    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
27    }
28
29  // query recieve Stat variable and print message details
30  MPI_Get_count(&Stat, MPI_CHAR, &count);
31  printf("Task %d: Received %d char(s) from task %d with tag %d \n
32         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
33
34  MPI_Finalize();
35  }
```

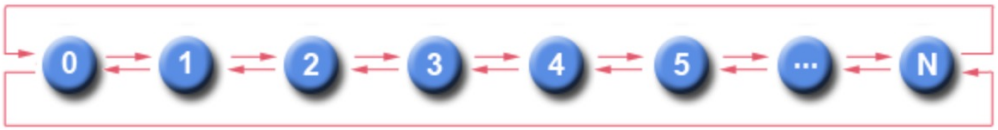


https://computing.llnl.gov/tutorials/mpi/

Point to point Communications

Non-Blocking Routines

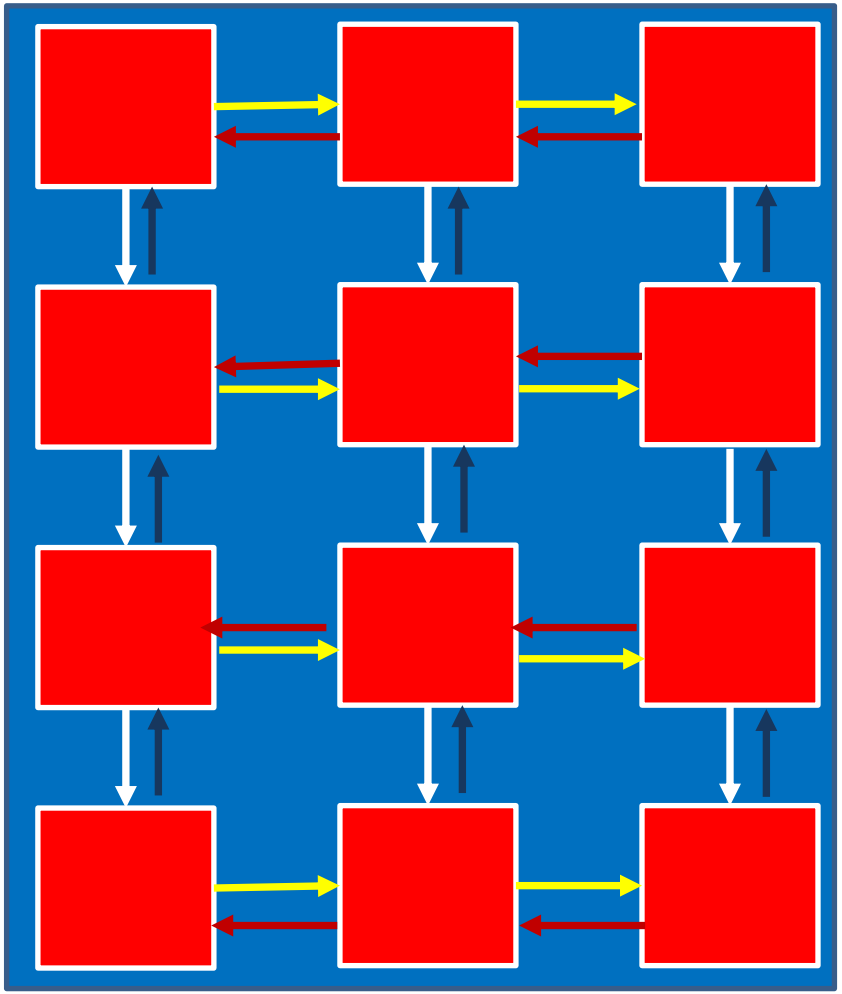
```
1   #include "mpi.h"
2   #include <stdio.h>
3
4   main(int argc, char *argv[])  {
5   int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
6   MPI_Request reqs[4];   // required variable for non-blocking calls
7   MPI_Status stats[4];   // required variable for Waitall routine
8
9   MPI_Init(&argc,&argv);
10  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
11  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13  // determine left and right neighbors
14  prev = rank-1;
15  next = rank+1;
16  if (rank == 0)  prev = numtasks - 1;
17  if (rank == (numtasks - 1))  next = 0;
18
19  // post non-blocking receives and sends for neighbors
20  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
21  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
22
23  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
24  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
25
26     // do some work while sends/receives progress in background
27
28  // wait for all non-blocking operations to complete
29  MPI_Waitall(4, reqs, stats);
30
31     // continue - do more work
32
33  MPI_Finalize();
34  }
```



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

Mesh Exchange

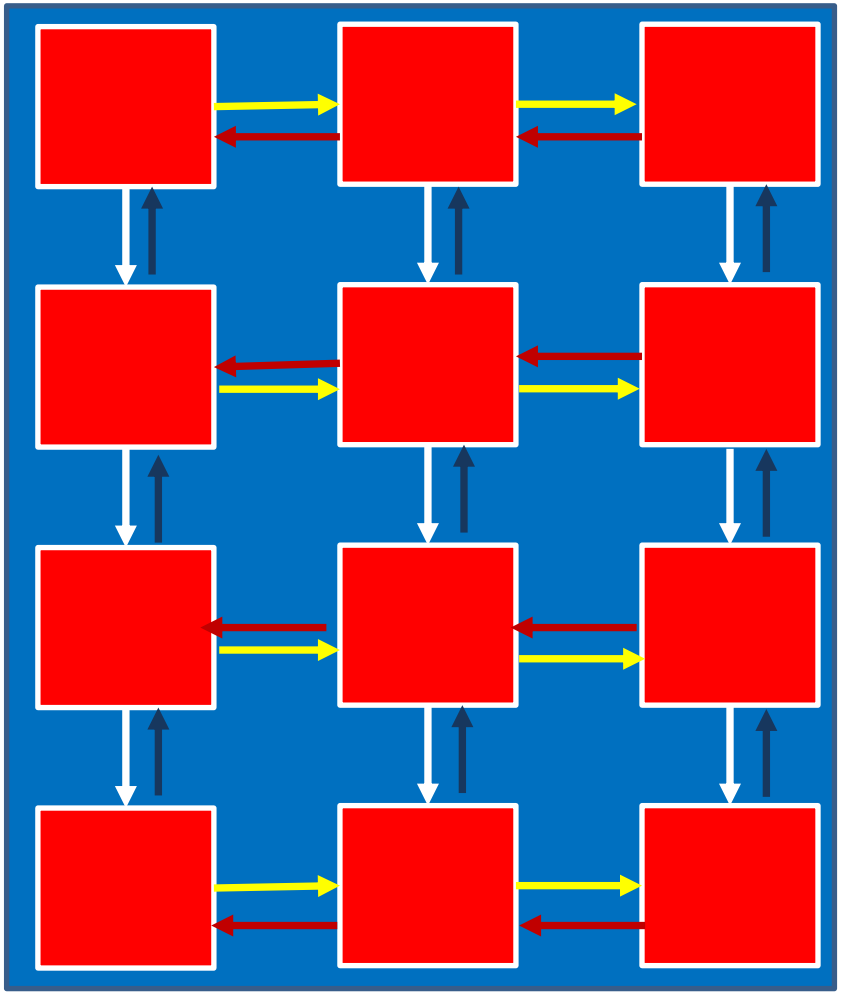
```
Do i = 1, n_neighbours

  Call MPI_send(edge,
       len, MPI_REAL,
       nbr(i),
       tag, comm,
       ierr)
enddo


Do i = 1, n_neighbours

  Call MPI_Recv(edge,
                len, MPI_REAL,
                nbr(i),
                tag, comm,
                status, ierr)
enddo
```



Deadlocks ! All of the sends block, waiting for a matching receive

Mesh Exchange

```
Do i = 1, n_neighbours
    if (i has down nbr)
        Call MPI_send(edge,
                      len, MPI_REAL,
                      down, tag, comm, ierr)
Enddo

Do i = 1, n_neighbours
    if (i has up nbr)
        Call MPI_Recv(edge,
                      len, MPI_REAL,
                      up, tag, comm, status ierr)
 Enddo

…
```



Performance issues !!!! Sequentializes (all except the bottom process blocks

Fix 1: Use Irecv

```
Do i = 1, n_neighbours
   Call MPI_Irecv(edge, len,
                  MPI_REAL,
                   nbr(i),
                  tag, comm,
                  request(i), ierr)
Enddo

Do i = 1, n_neighbours
     Call MPI_send(edge, len,
                   MPI_REAL,
                   nbr(i),
                   tag, comm, ierr)

Enddo

Call MPI_Waitall(n_neighbours, request
statues, ierr)
```

Six steps !

Timing Model

- Sends interleave
- Sends block (only if data larger than buffering)
- Sends control timing
- Receives do not interfere with Sends

Fix 2: Use Isend and Irecv

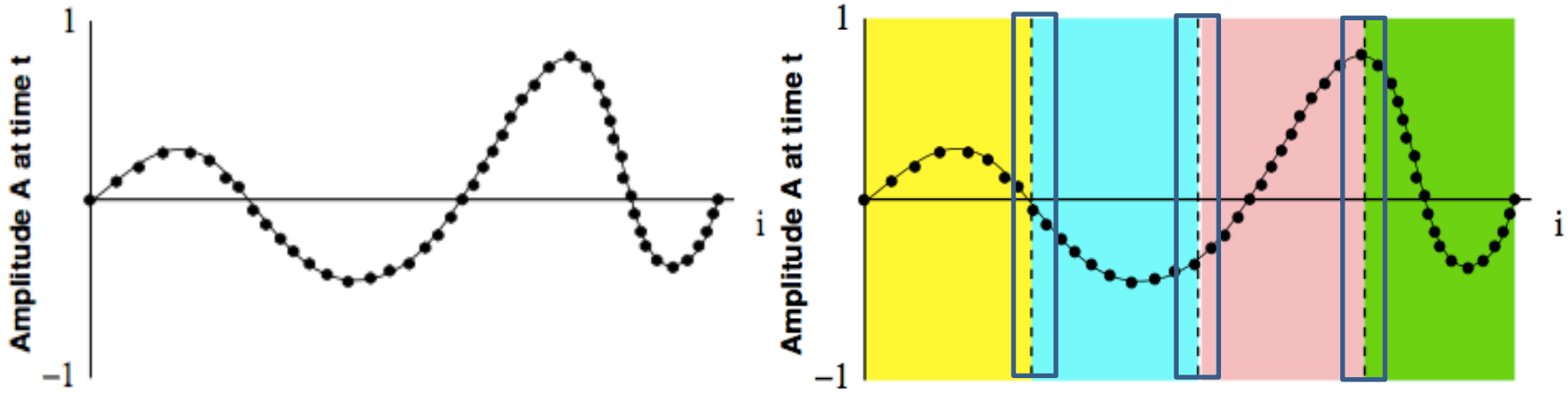
```
Do i=1,n_neighbours
   Call MPI_Irecv(edge,len,MPI_REAL,
                  nbr(i),tag,
                  comm,request(i),ierr)
Enddo

Do i=1,n_neighbours
   Call MPI_Isend(edge, len, MPI_REAL,
                  nbr(i), tag,comm,
                  request(n_neighbors+i),
                  ierr)
Enddo

Call MPI_Waitall(2*n_neighbors, request,
                 statuses, ierr)
```

4 steps !

- Exchange can be done in 4 steps (down, right, up, left)

1-D wave equation

- 1-D wave equation:



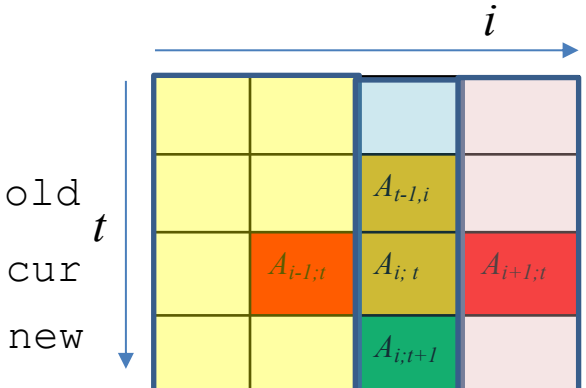
$$A_{i;t+1} = 2x A_{i;t} - A_{i;t-1} + cx(A_{i-1;t} - (2x A_{i;t} - A_{i+1;t}))$$

Amplitude $A_{t+1;i}$ depends on

- Amplitude at neighboring points
- Amplitude at previous time steps

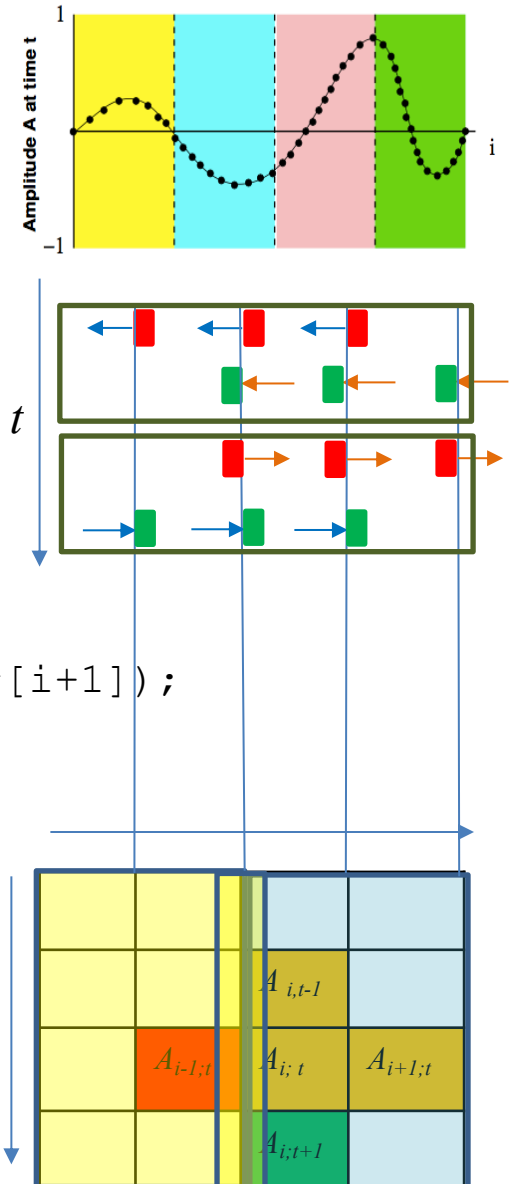
Parallel pseudocode: initialization

```
1.    localSize = npoints / num_tasks
2.    double old[npoints+2], new[npoints+2], cur[npoints+2];
3.    myLeft = myRank - 1;            // special case boundaries left out
4.    myRight = myRank + 1;
5.    if (myRank==MASTER) {          //we assume MASTER=0
6.      for (p = 1 .. num_tasks-1) {
7.        init (cur[1..localSize]); send(p, cur[1..localSize]);
8.        init (old[1..localSize]); send(p, old[1..localSize]);
9.      }
10.     init(cur[1..localSize]);
11.     init(old[1..localSize]);
12.   }
13.   else {                         // I am worker
14.     receive (MASTER, cur[1..localSize]);
15.     receive (MASTER, old[1..localSize]);
•     }
```



Parallel pseudocode: core

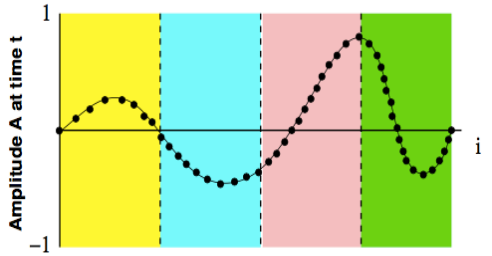
```
1.    myLeft = rank - 1;
2.    myRight = rank + 1;
3.     …
4.     for (t = 1 .. nsteps) {
5.       send(myLeft, cur[1]);
6.       receive(myRight, cur[localSize +1]);

7.       send(myRight, cur[localSize]);
8.       receive(myLeft, cur[0]);

9.       for (i= 1.. localSize) {
10.        new[i] = f(old[i], cur[i], cur[i-1], cur[i+1]);
11.      }
12.      old = cur;
13.      cur = new;
14.    }
```

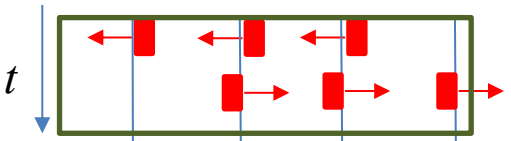


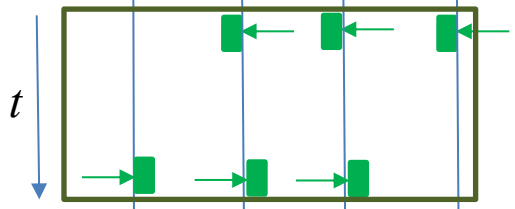
Performance: Postpone the calculation of the points in the helo regions to the last (for loop should start from 2 to localsize -1

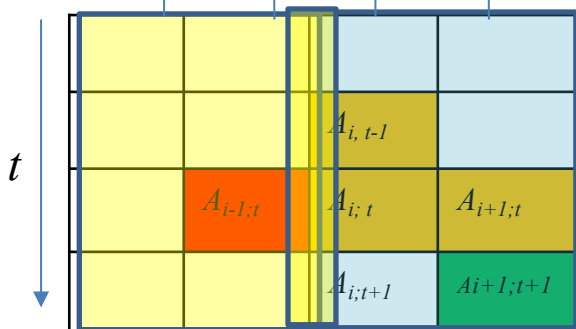
Parallel pseudocode: improvement?

```
1.    …
2.     myLeft = rank - 1;
3.     myRight = rank + 1;
4.     …
5.     for (t = 1 .. nsteps) {
6.        send(myLeft, cur[1]);
7.        send(myRight, cur[localSize]);

8.        for (i= 2.. localSize-1) {
9.          new[i] = f(old[i], cur[i], cur[i-1], cur[i+1]);
10.       }
11.       receive(myRight, cur[localSize +1]);
12.       new[localSize] = …

13.       receive(myLeft, cur[0]);
14.       new[1] = …
15.
16.     old = cur; cur = new;
17.   }
```









Solution: Postpone the calculation of the points in the helo region to the last (line 11-12, and line 13-14)

Parallel pseudocode: **finalization**

```
1. …
2.    if (myRank==MASTER) {              //we assume MASTER=0
3.      write(file, cur[1..localSize]);
4.      for (p = 1 .. num_tasks) {
5.        receive(p, cur[1..localSize]);
6.        write(file, cur[1..localSize]);
7.      }
8.    }
9.    else {                             // I am worker
10.     send (MASTER, cur[1..localSize]);
11.   }
```

Takehome message

- Standard for large scale (distributed)parallel machines
  - Process-based
  - Explicit communication and synchronisation
- Typical computation models
  - SPMD
  - Explict data (domain) decomposition
- Typical organization
  - Master distributes works & starts work session
  - Every process (master inclusded) contribute a share of the workload
  - Processs communicate (intermediate) results
  - Master computes (and prints) final results

- MPI is a low-level programming model
- MPI implementations are available on all supercomputer
- Most application are easier to implement using higher-level models
  - Provides much simpler front-end construct
  - Implict data distribution easier computation models, no explicite communication
  - Use MPI as back-end

- **Example**
  - **MapREduce**
  - **Spark**

Other/advanced MPI topics

Groups vs communicators

A **group** is an ordered set of processes.

- Each process in a group is associated with a unique integer rank.
- Rank values start at zero and go to N-1, where N is the number of processes in the group.
- A group is accessible by a "handle".
- A group is always associated with a communicator.

A **communicator** encompasses groups of processes that communicate w/each other.

- All MPI messages must specify a communicator.
- the communicator is an extra "tag" that must be included with MPI calls.
- communicators are accessible through "handles".

programmer's perspective,

- a group and a communicator are one.
- The group routines are used to specify which processes should be used to construct a communicator.

Primary Purposes of Group

- Allow you to organize tasks, based upon function, into task groups.
  - Enable Collective Communications operations across a subset of related tasks.
  - Provide basis for implementing user defined virtual topologies
  - Provide for safe communications

Programming Considerations and Restrictions

- Groups/communicators are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
- MPI provides over 40 routines on groups, communicators, and virtual topologies.