Parallel Programming Models - II

Lecture 09

Outline

- Data (work) Distribution
  - 1D array
  - 2D array
- Information Exchange
  - Shared variables
  - Communication operations
- Summary

Data Distribution

- Parallel computing problems are commonly based on array of various dimensions

- Useful to study how to decompose the arrays for distribution on multiple processors
  - known as **data distribution / work distribution / decomposition / partitioning**

- For problems exhibiting data parallelism, data distribution can be used as a simple parallelization strategy

Data Distribution for 1D Arrays

- Assumptions for discussion:
  - **p** identical processors, P_1, P_2, .., P_p, and with processor rank i in {1, 2, .., p}
  - Array elements numbered from 1 to n

- Given a one dimensional array, common distribution patterns:
  - Blockwise data distribution
  - Cyclic data distribution

Blockwise and Cyclic Data Distribution

D														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Blockwise

P_1	1	2	3	4
P_2	5	6	7	8
P_3	9	10	11	12
P_4	13	14		

- Block size, B = $\left\lceil \frac{n}{p} \right\rceil$
- Pj takes elements $[(j-1) \times B + 1 \ldots j \times B]$

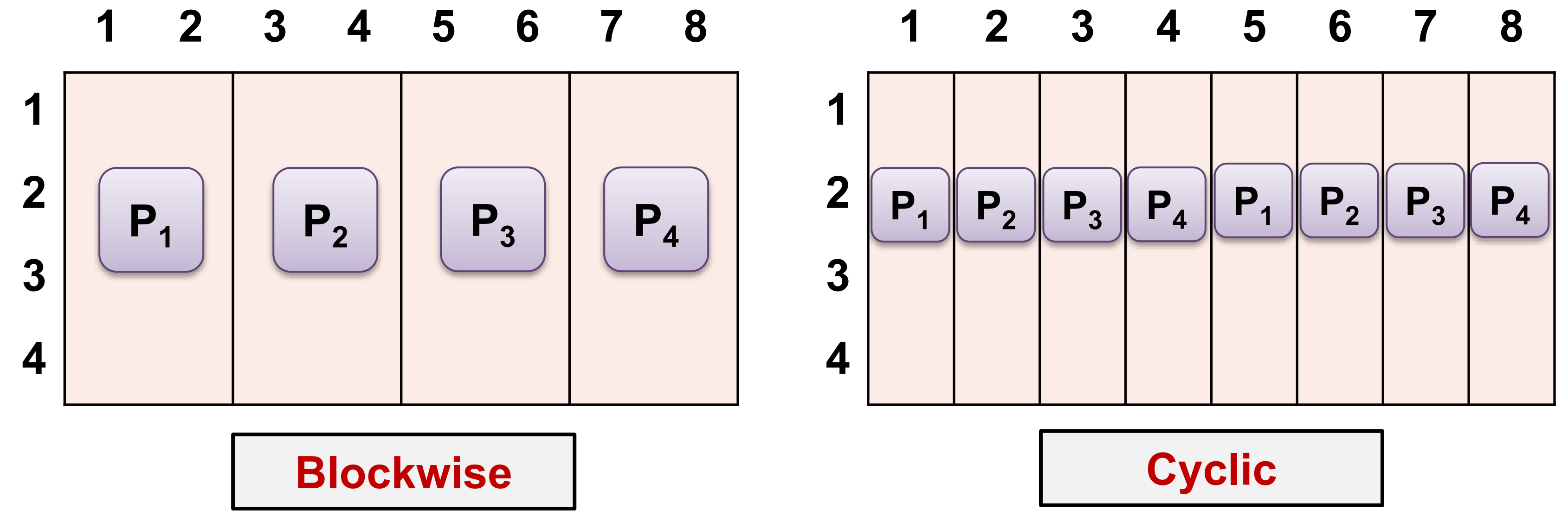
Cyclic

P_1	1	5	9	13
P_2	2	6	10	14
P_3	3	7	11	
P_4	4	8	12	

- Pj takes elements
  $[\mathrm{j}, \mathrm{j}+\mathrm{p}, \ldots, \mathrm{j}+(\left\lceil \frac{n}{p} \right\rceil - 1) \times p]$ if j $\leq$ $n \bmod p$
  $[\mathrm{j}, \mathrm{j}+\mathrm{p}, \ldots, \mathrm{j}+(\left\lceil \frac{n}{p} \right\rceil - 2) \times p]$ otherwise

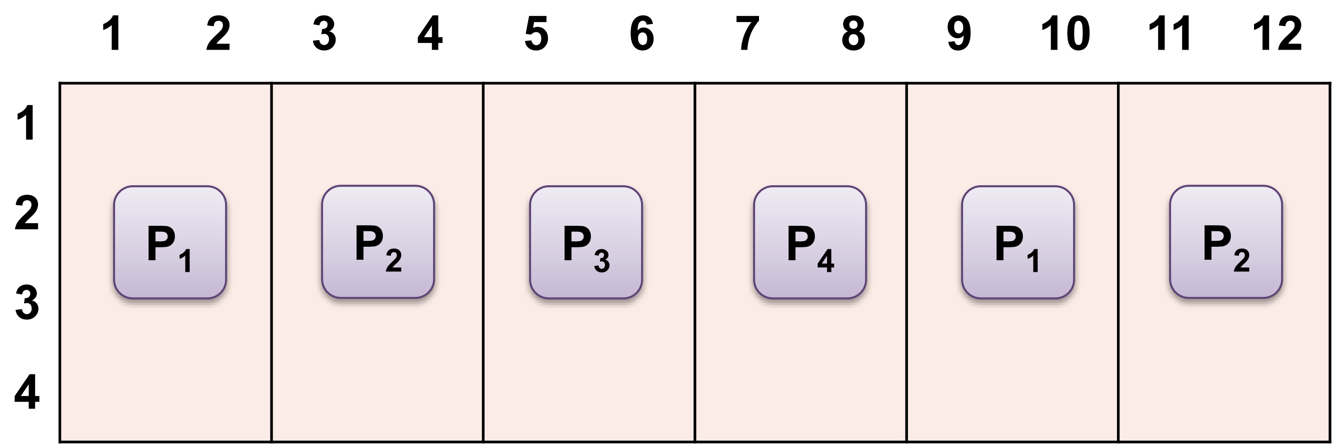
Data distribution for 2D Arrays

- Combination of blockwise / cyclic distribution in one or both dimensions can be used
- One-dimension distributions
  - Use the **column dimension** for illustration:



Data distribution for 2D Arrays

- One-dimension distributions
  - **Block-Cyclic** is a new distribution pattern
  - Form blocks of size **b**, then perform cyclic (round robin) allocation



Block-Cyclic with ***b* = 2**

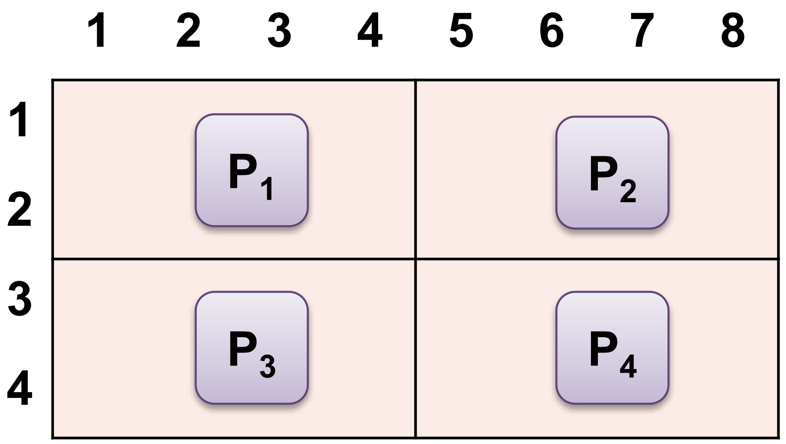
Two Dimensional Distributions

- Processors are **virtually organized into 2D mesh of R x C**:
  - i.e. each Processor now has a row and column number:

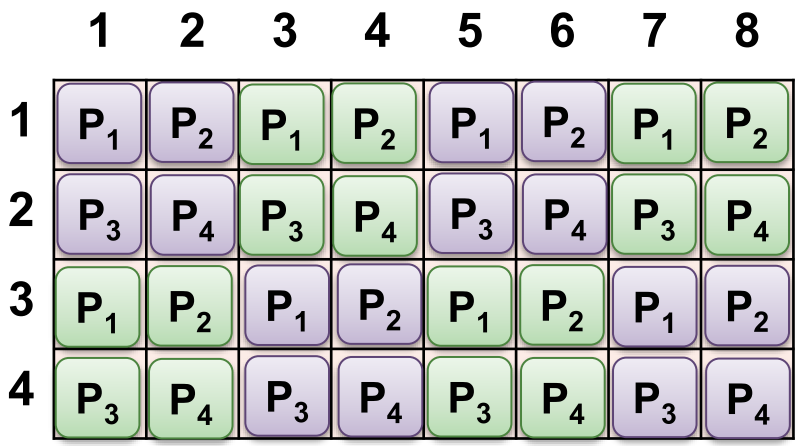
	1	2
1	P_1	P_2
2	P_3	P_4

  - **Checkerboard** distribution can then be applied:
    - **Blockwise**: elements split into blocks along both dimensions depending on R and C
    - **Cyclic**: cyclic assignment of elements according to processor mesh
    - **Block-Cyclic**: elements spilt into b1 x b2 size blocks, then cyclical assignment to processors

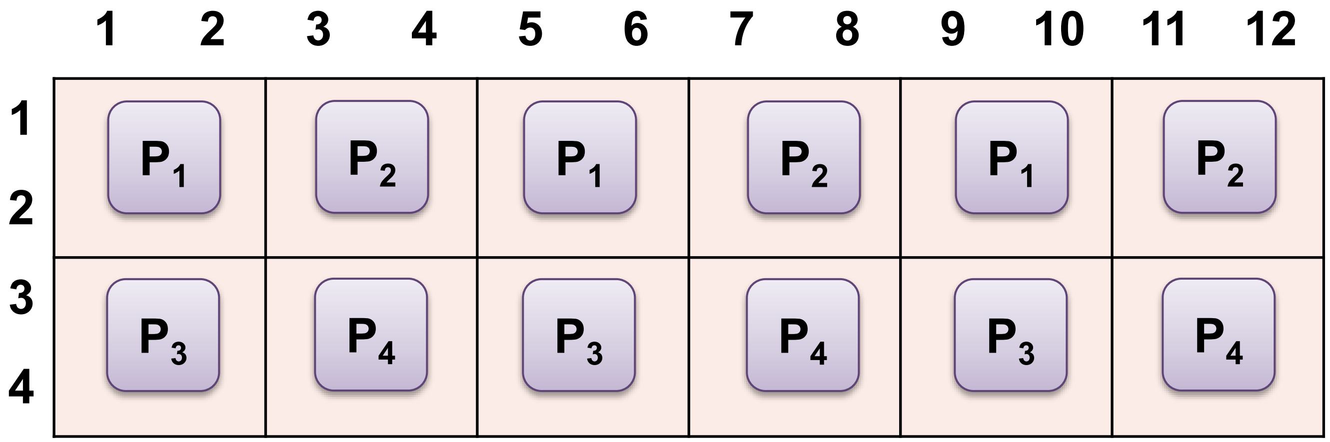
Checkerboard Distribution



Blockwise



Cyclic



Block-Cyclic with $b_1 = 2$, $b_2 = 2$

Exercise: Matrix Multiplication

- To illustrate the effect of data distribution on the computation

- Assume:
  - A x B = C, all matrices of N x N
  - There are p processors, p will be specified

  - For each value of p, suggest a data distribution pattern for the matrices A and B

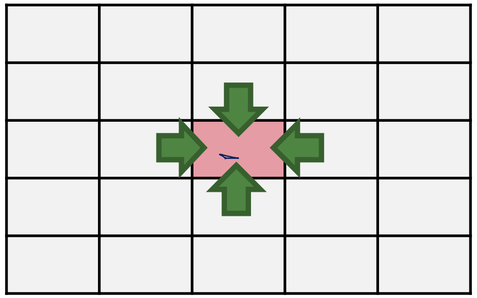
Exercise: Matrix Multiplication

1. $1 < p \leq N$, you can use $p = N$ as a start
   - A distributed as
   - B distributed as
   - Each processor calculate

2. $p = N^2$
   - A distributed as
   - B distributed as
   - Each processor calculate

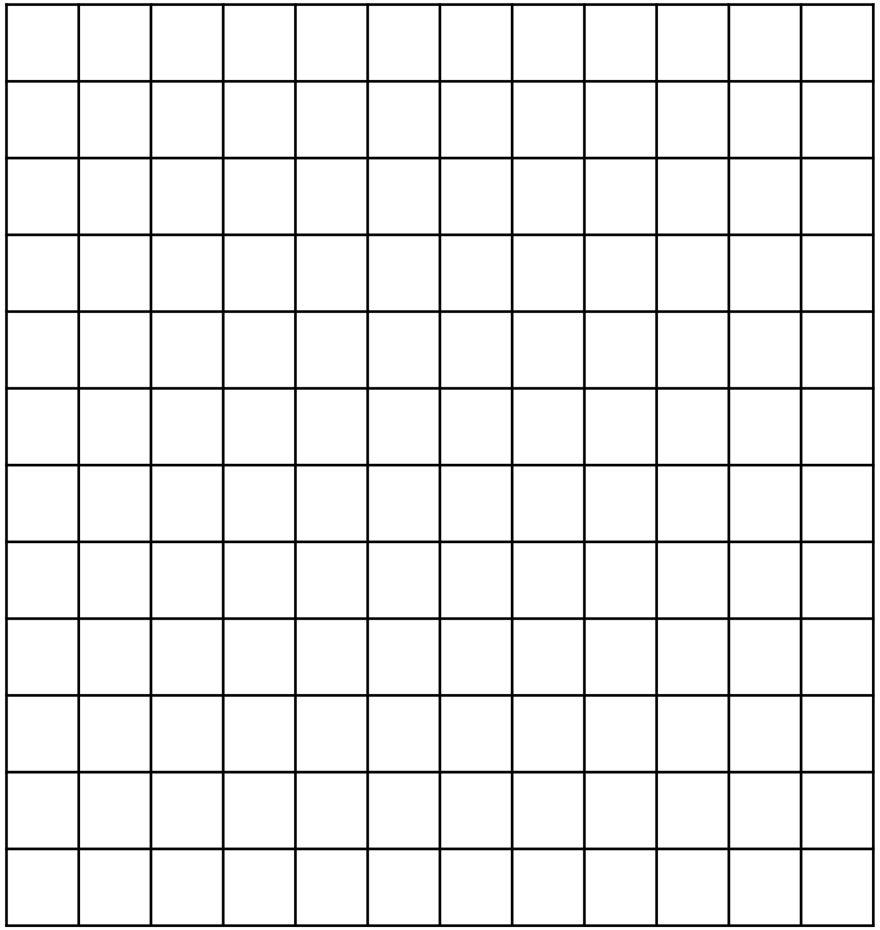
Exercise: Heat Transfer Simulation

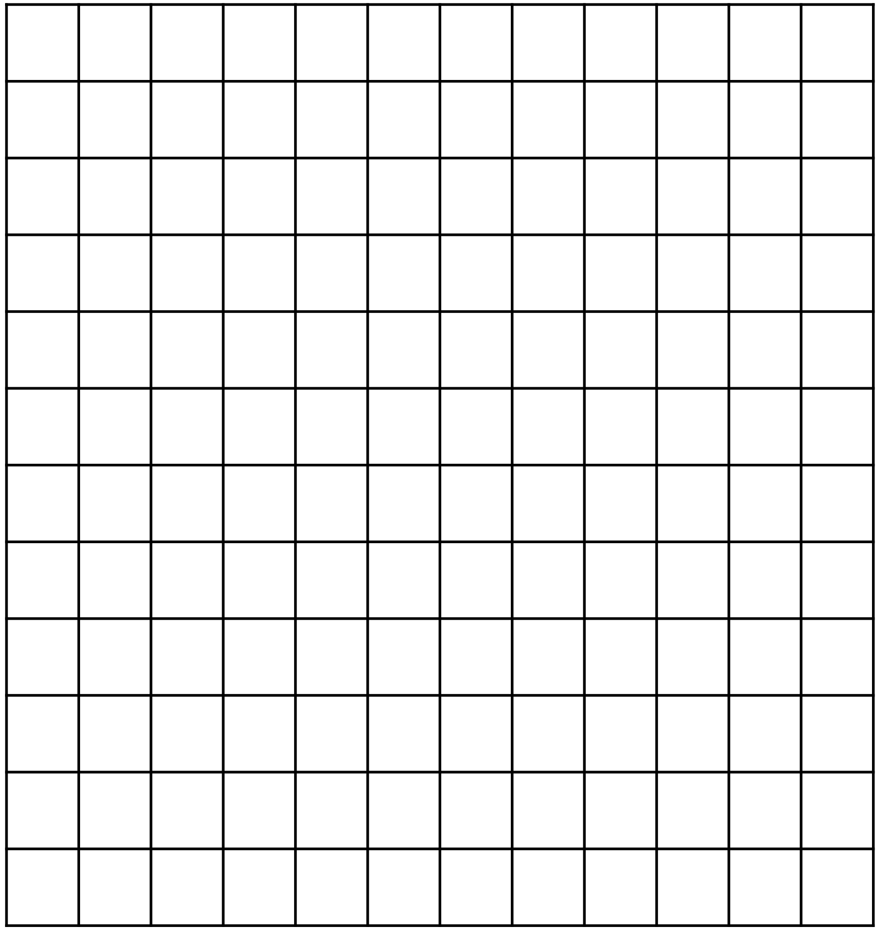
- A simplistic simulation of heat transfer on a metal plate
- The metal plate is modeled as:
  - 2D integer array
  - Each integer represent the temperature of a "point" on the plate
- The temperature is calculated iteratively:
  - Temperature of a point = Average of the top, left, right and down points



Exercise: Heat Transfer Simulation

- If we have a N x N metal plate and p processor, where $p < N$:
  - Suggest at least two data distribution patterns and discuss pro/cons





I'll trade my B for your A

INFORMATION EXCHANGE

Information Exchange

- **Purpose**
  - Information exchange between the executing processors is necessary for controlling the coordination of different parts of a parallel program execution

- **Shared address space**
  - use **Shared variables**

- **Distributed address space**
  - use **Communication operations**

Shared Variables

- Shared memory programming models assume a global memory accessible by all processors
  - Information exchange through shared variables
  - Need synchronization operations for safe concurrent access

- Flow of control abstractions

  ➔ processes or threads

- Each thread:
  - Executed by one processor or one core in multicore processors
  - Have shared variables and may have private variables

Synchronized Access

- **Data race:** multiple threads accessing (read and write) the same shared variable
  - Computation result depends on the execution order of threads **(race condition)**
  - May lead to **non-deterministic** behavior
  - Can be avoided using a **critical section** mechanism

- Critical section:
  - A program part in which concurrent access should be avoided
    - i.e. only one thread can execution at any point in time
  - Use **mutual exclusion** (**mutex**) to provide critical section

Example: **OpenMP**

- Data race

(might produce a race condition):

```
void main () {
  int count = 0;
  #pragma omp parallel
  {
    count = count + 1; // race
  }
  printf("count = %d\n", count);
}
```

- Mutual exclusion:

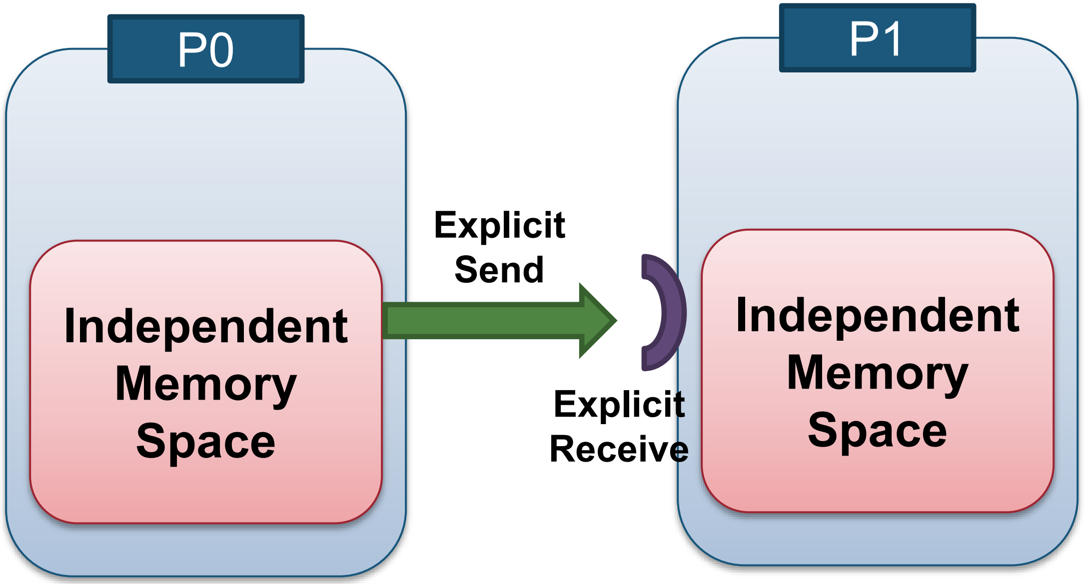
```
void main () {
  int count = 0;
  omp_lock_t lock;
  omp_init_lock(&lock);

  #pragma omp parallel
  {
    omp_set_lock(&lock);
    count = count + 1;
    omp_unset_lock(&lock);
  }
  printf("count = %d\n", count);
}
```

Communication Operations

- Distributed memory programming models assume disjoint memory space:
  - Exchange of data between processors through **dedicated communication operations**
- One common communication model send / receive messages between participating processors:
  - known as **message-passing programming model**
- Two main types of data exchange:
  - **point-to-point** and global communication

Principles of Message Passing Model



- Data explicitly partitioned for each process
- All interaction requires both parties to participate
- ➔ The programmer has to explicitly express parallelism

Principles of Message Passing Model

- **Loosely synchronous** paradigm:
  - Tasks or subsets of tasks synchronize to perform interactions
  - Between these interactions, tasks execute completely **asynchronously**

So, you talk, I talk?

COMMUNICATION PROTOCOLS

Send and Receive Operations

```
a = 100;
send(&a, P1);
a = 0;
```

Process P0

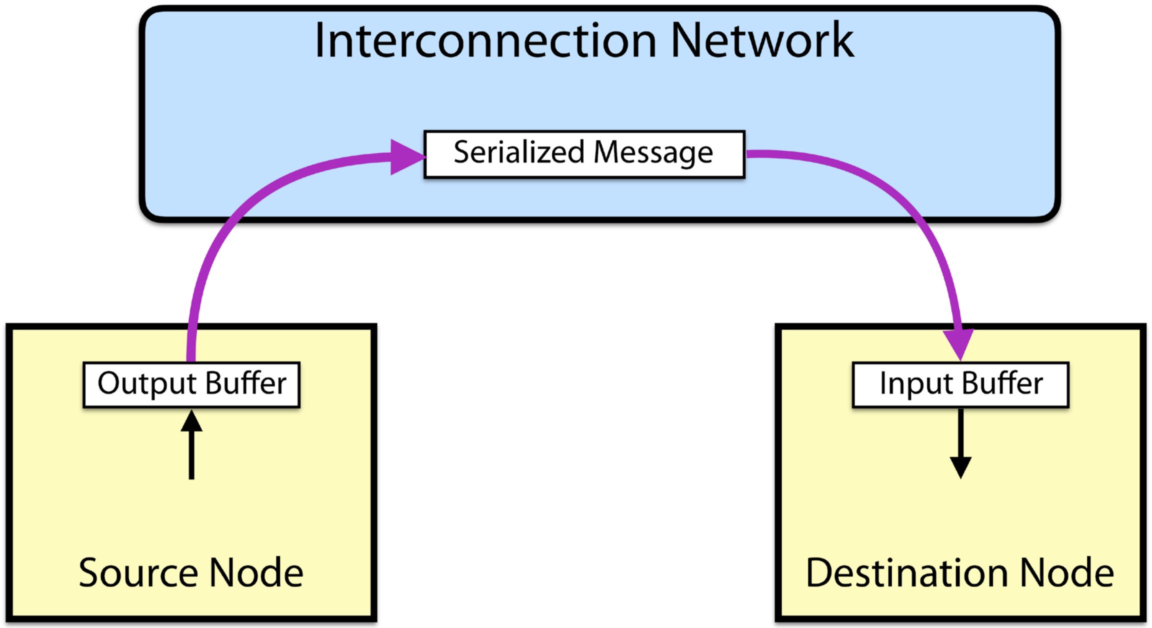
```
receive(&a, P0);
printf("%d\n", a);
```

Process P1

- Semantic of the send():
  - The value received by P1 should be 100
- Motivates the design of the underlying communication protocols

Point-to-point Communication (Buffered)

- In a distributed memory system, over a network
  - One-way transfer



Send and Receive Protocols Possibilities

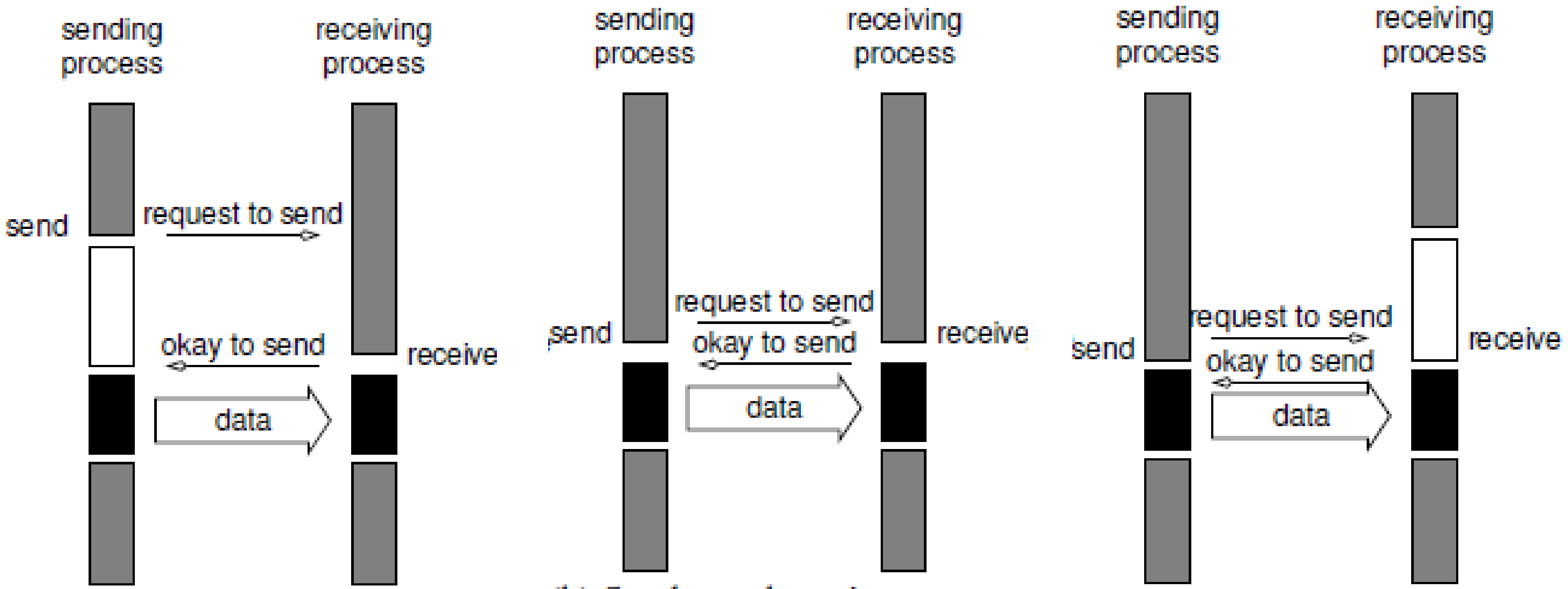
	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer	Sending process returns after initiating the transfer to buffer. This operation might not be completed on return.
Non-buffered	Sending process blocks until matching receive operation has been encountered.	
	Send and receives semantics assured by corresponding operation.	Programmer must explicitly ensure completion of the operation by polling.

Blocking Send

- Send operation blocks until it is **safe** to reuse the input buffer
  - "Safe" refers to the integrity of the data to be sent

- Non-buffered blocking send:
  - The operation blocks until the matching receive has been performed by the receiving process
  - Idling and deadlocks are major issues with non-buffered blocking sends

Non-Buffered + Blocking Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

- Considerable idling overheads
  - Due to the mismatch in timing between sender and receiver

Buffered + **Blocking** Operations

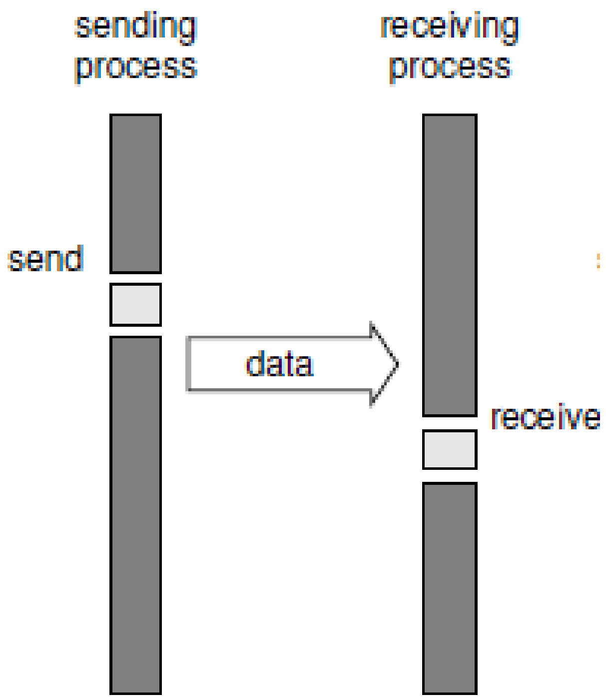
- To reduce idling overhead:
  - Utilize **buffers** at both ends

- **Sender** simply copies the data into the designated buffer and **returns** after the copy operation has been completed

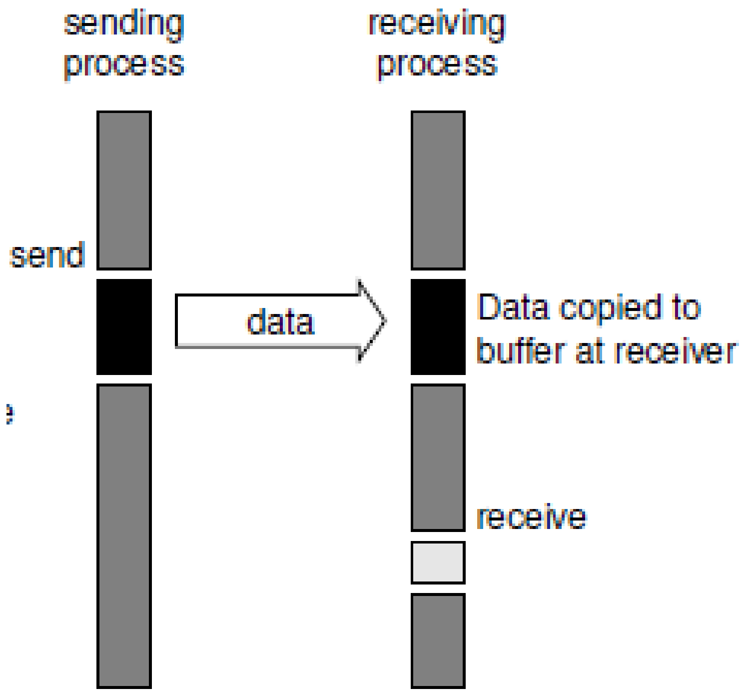
- **Receiver** similarly buffered the incoming data

- Buffering trades off **idling overhead** for **buffer copying overhead**

Buffered + Blocking Operations



(a) Using communication hardware with buffers at send and receive ends



(b) Without communication hardware, sender interrupts receiver and deposits data in buffer at receiver end

Blocking buffered transfer protocols

Bounded Buffer Size: Impact

```
for (i = 0; i < 1000; i++){
    produce(&a);
    send(&a, P1);
}
```

Process P0

```
for (i = 0; i < 1000; i++){
    receive(&a, P0);
    consume(&a);
}
```

Process P1

- What if consumer was much slower than producer?
  - Think "behind the scene"….

Deadlock

- Deadlocks are still possible with buffering since receive operations block:

```
receive(&a, P1);
send(&b, P1);
```

Process P0

```
receive(&a, P0);
send(&b, P0);
```

Process P1

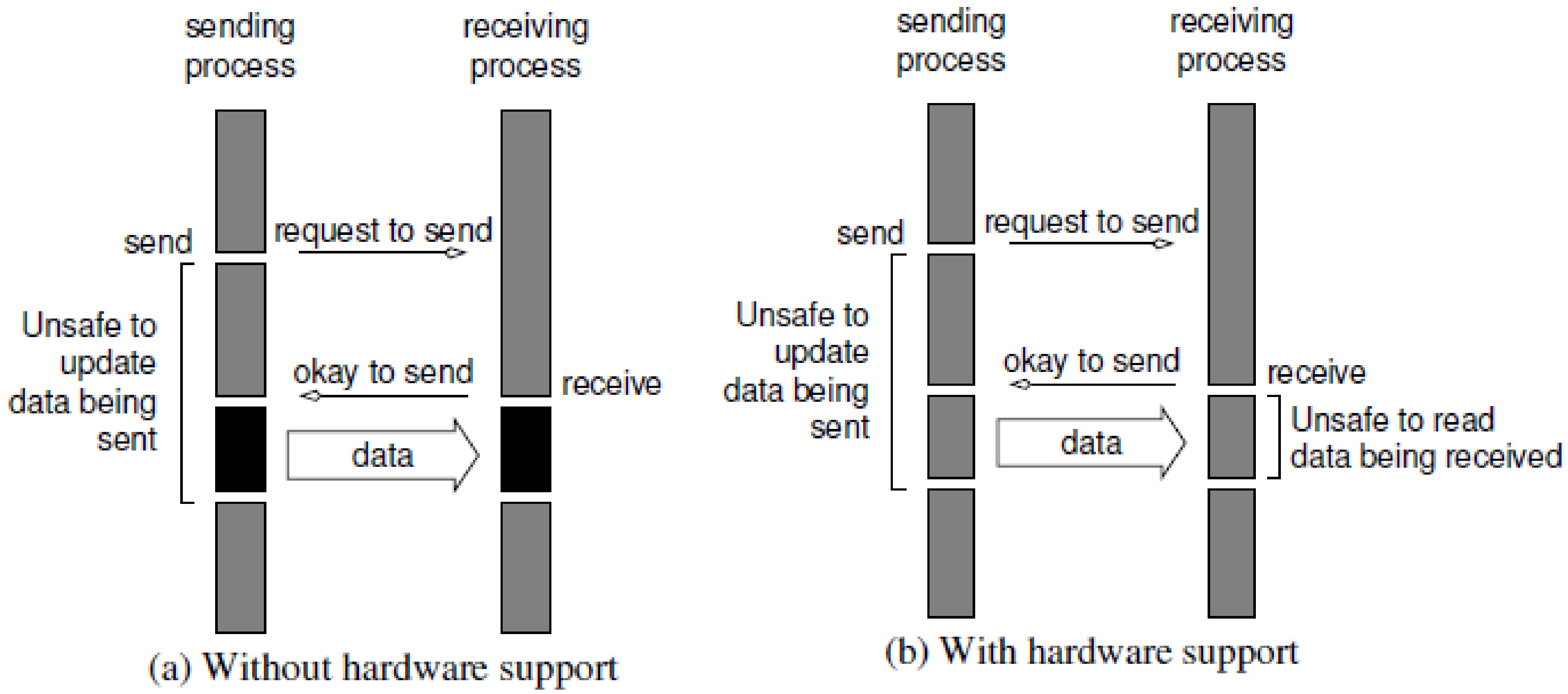
Non-Blocking Operations

- Send / Receive returns before it is semantically safe to use the data transferred
  - Non-blocking operations are generally accompanied by a ***check-status*** operation
  - The programmer must ensure the semantics of the operations

- When used correctly, these primitives are capable of overlapping communication overheads with useful computations

- Message passing libraries typically provide both blocking and non-blocking primitives

Non-Blocking + Non-Buffered Operations



Non-blocking non-buffered send and receive operations

Semantics of Send/Receive Operations

Local view	Global view
Blocking Return from a library call indicates the user is allowed to reuse resources specified in the call	**Synchronous** Communication operation does not complete before both processes have started their communication operation
Non-blocking A procedure may return before the operation completes, and before the user is allowed to reuse resources specified in the call	**Asynchronous** Sender can execute its communication operation without any coordination with the receiver

Implementation Options

- Synchronous:
  - Send completes after matching receive and source data sent
  - Receive completes after data transfer completed from matching send
- Asynchronous:
  - Send completes after input buffer may be reused

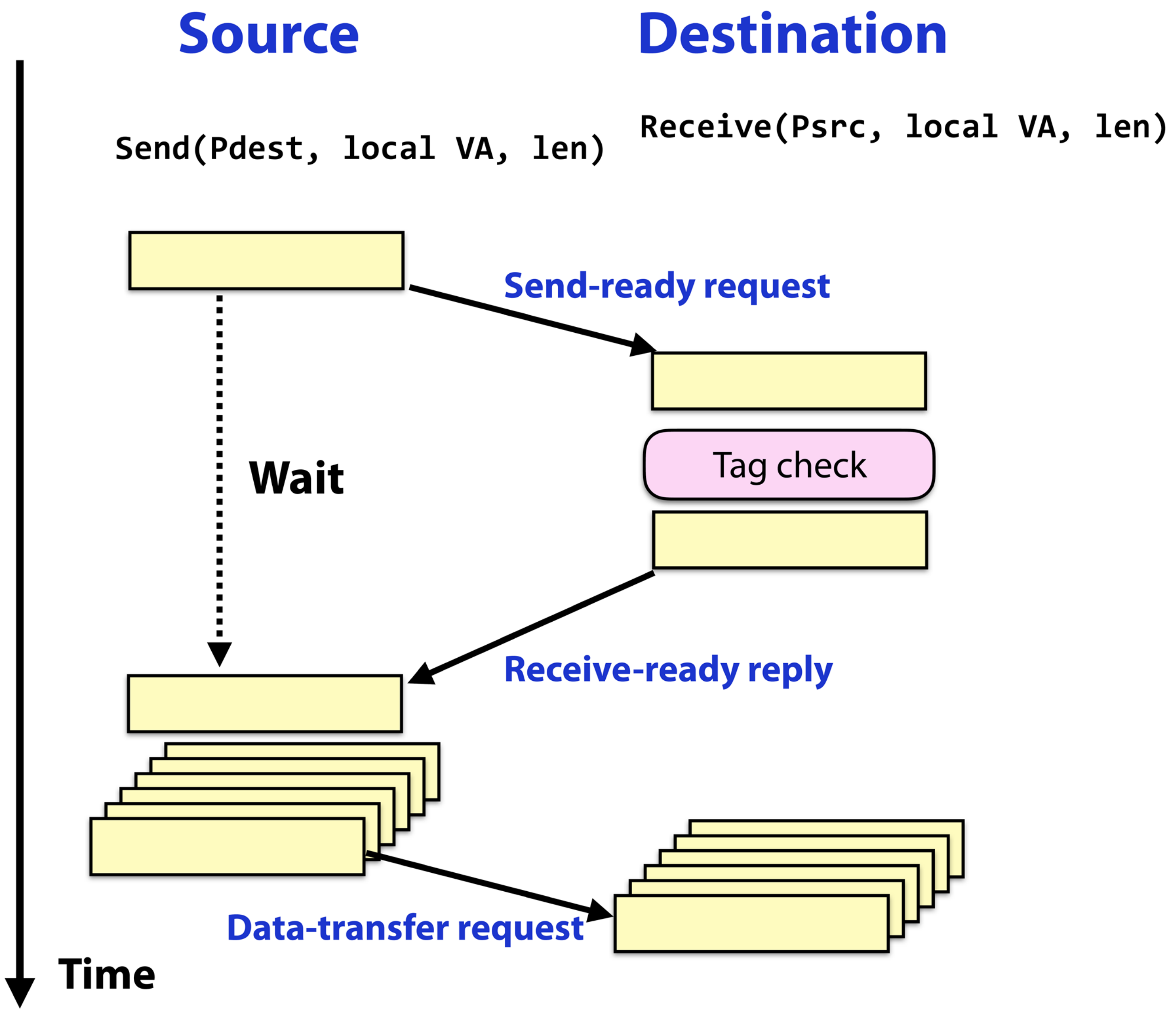
Synchronous Communication

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) Send-ready request

(5) Remote check for posted receive (assume success)

(6) Reply transaction

(7) Bulk data transfer
Source VA —> Dest VA



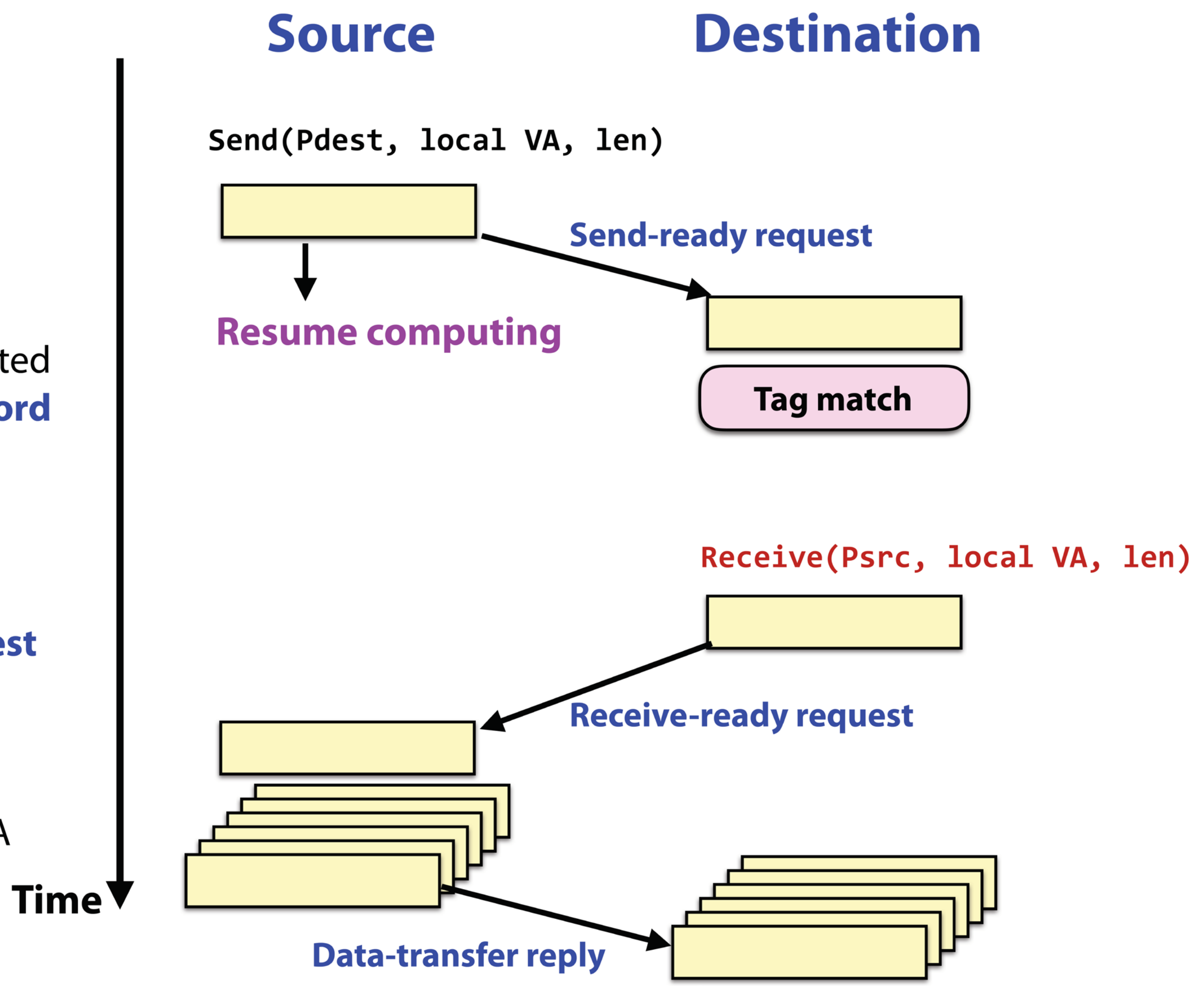
Asynchronous Communication

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) Send-ready request

(5) Remote check for posted receive (assume fail); **record send-ready**

(6) Receive-ready request

(7) Bulk data reply
Source VA —> Dest VA



Summary of Communication Protocols

1. **Overhead** of guaranteeing semantic correctness:
   - **Idling** (non-buffered)
   - **Buffer management** (buffered)
2. **Side effect:**
   - Safe and easier programming (**blocking**)
   - Hide communication overhead (**non-blocking**)
3. **Local or global view:**
   - Synchronous vs. asynchronous communication

Summary

- Data distribution
- Information exchange for shared and distributed address space