

MPI

-
-

Copyright (c) 2012 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

The Butterfly Swap Operations

Communicators and Groups defines collection of processes that may communicate with each other.

Need to specify a communicator as an argument.

MPI_COMM_WORLD - predefined communicator that includes all of your MPI processes.

Within a communicator, every process has its own unique, integer identifier, called rank or “task ID”.

Used to specify the source and destination.
Also can be used in conditional statements.

MPI_Alltoall

MPI_Alltoall - Sends data from all to all processes

```
int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype,  
                void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm )
```

INPUT PARAMETERS

sendbuf - starting address of send buffer (choice)

sendcounts - integer array equal to the group size specifying the number of elements to send to each processor

sendtype - data type of send buffer elements (handle)

recvcounts - integer array equal to the group size specifying the maximum number of elements that can be received from each processor

recvtype - data type of receive buffer elements (handle)

comm - communicator (handle)

OUTPUT PARAMETERS

recvbuf - address of receive buffer (choice)

MPI_Alltoallv

MPI_Alltoallv - Sends data from all to all processes, with a displacement

```
int MPI_Alltoallv (void *sendbuf, int *sendcnts, int *sdispls, MPI_Datatype sendtype,
void *recvbuf, int *recvcnts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm )
```

INPUT PARAMETERS

sendbuf - starting address of send buffer (choice)

sendcounts - integer array equal to the group size specifying the number of elements to send to each processor

sdispls - integer array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j)

sendtype - data type of send buffer elements (handle)

recvcounts - integer array equal to the group size specifying the maximum number of elements that can be received from each processor

rdispls - integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i)

recvtype - data type of receive buffer elements (handle)

comm - communicator (handle)

OUTPUT PARAMETERS

recvbuf - address of receive buffer (choice)

MPI_Alltoallv

Alltoallv

flexibility in that the location of send data is specified by `sdispls` and the location of the placement of receive data is specified by `rdispls`.

The ***j*th block** sent from **process *i*** is received by **process *j*** and is placed in the ***ith* block**.

Need not be all the same size block

`sendcount[j]`, sendtype at **process *i***
`recvcount[i]`, recvtype at **process *j***.

The amount of data sent must be equal to the amount of data received, pairwise between every pair of processes.

Distinct type maps between sender and receiver are still allowed.

MPI_Alltoallw

ALLTOALLW in MPI-2.

Can specify separately count, displacement, and datatype.

The displacement of blocks is specified in bytes.

Can be seen as a generalization several MPI functions depending on the input arguments.

Communication Parameters

Point to point communication

Simple latency / bandwidth model

Not good for ping-pong benchmark data

MPI message transfer is complex

Message Envelope

supplementary information such as
length, sender, tag, etc

Eager Protocol

Rendezvous Protocol

Eager Protocol

For **short** messages

The message itself + supplementary information (**message envelope**)
may be sent and stored at the receiver side
without receiver's intervention

A **matching receiver** operation may **not be needed**
But afterward, the message in the **intermediate buffer**
must be copied to the **receive buffer**

- + **Synchronization** overhead is reduced
- May require large amount of preallocated **buffer space**
- **Flooding** a process with many eager messages may overflow → **contention**

Rendezvous Protocol

For **large** messages

Buffering the data is impossible

The **message envelope** is **immediately stored** at the receiver

The actual message transfer **blocks** until the user's **receive buffer** is available

Extra **data copy** could be avoided,
improving **effective bandwidth**,
but sender and receiver must **synchronize**.

Communication Modes

Blocking

- Standard
- Buffered
- Synchronous
- Ready

Immediate

- Standard
- Buffered
- Synchronous
- Ready

Blocking

does **not return until** the **message data** and **envelope** have been safely **stored** away so that the sender is free to **access** and **overwrite** the send buffer.

The message might be copied directly into the matching **receive buffer**, or it might be copied into a **temporary system buffer**.

Message buffering decouples the send and receive operations.

A blocking send can **complete** as soon as the message was **buffered**, even if no matching receive has been executed by the receiver.

On the other hand, message buffering can be **expensive**, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering.

MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

Standard Communication Mode

It is up to MPI to decide whether outgoing messages will be buffered.

- 1) MPI may **buffer** outgoing messages.
→ the send call may **complete before** a **matching** receive is invoked.
- 2) **Buffer space** may be **unavailable**, or
MPI may choose **not to buffer** outgoing messages, for performance reasons.
→ the send call will **not complete** until a **matching** receive has been posted,
and the data has been **moved** to the receiver.

Thus, a send in standard mode **can be started**
whether or not a **matching** receive has been posted.
It **may complete before** a matching receive is posted.

The standard mode send is **non-local**: successful completion of the send operation may depend on the occurrence of a **matching receive**.

Buffered Communication Mode

A buffered mode send operation **can be started** **whether or not** a **matching** receive has been posted.

It **may complete before** a **matching** receive is posted.

However, unlike the standard send, this operation is **local**, and its completion does **not depend** on the occurrence of a matching receive.

Thus, if a send is executed and no matching receive is posted, then MPI **must buffer** the outgoing message, so as to allow the send call to **complete**.

An **error** will occur if there is **insufficient buffer space**. The **amount** of available buffer space is controlled by the **user**.

Buffer **allocation** by the **user** may be required for the buffered mode to be effective.

Synchronous Communication Mode

A send that uses the synchronous mode **can be started whether or not** a **matching** receive was posted.

However, the send will **complete** successfully **only if** a matching receive is **posted**, **and** the **receive** operation has **started** to receive the message sent by the synchronous send.

Thus, the **completion** of a synchronous send not only indicates that the **send buffer** can be **reused**, but also indicates that the **receiver** has reached a certain point in its execution, namely that it **has started** executing the matching receive.

If both sends and receives are **blocking operations** then the use of the **synchronous mode** provides **synchronous communication** semantics: a communication does **not complete** at either end **before both** processes **rendezvous** at the communication.

A send executed in this mode is **non-local**.

Ready Communication Mode

A send that uses the ready communication mode **may be started only if the matching** receive is already **posted**.
Otherwise, the operation is **erroneous** and its outcome is **undefined**.

On some systems, this allows the **removal of a hand-shake operation** that is otherwise required and results in improved **performance**.

The **completion** of the send operation does **not depend** on the status of a **matching** receive, and merely indicates that the **send buffer** can be **reused**.

A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation;

it is merely that the sender provides **additional information** to the system (namely that a **matching** receive is **already posted**), that can save some overhead.

In a correct program, therefore, a ready send could be replaced by a **standard** send with no effect on the behavior of the program **other than performance**.

NonBlocking Communication (1)

overlapping communication and computation
light-weight threads vs nonblocking communication.

A nonblocking send (receive) start call **initiates** the send (receive) operation, but does **not complete** it.

The send (receive) start call will **return before** the message was copied out of (into) the send (receiver) buffer.

A **separate** send (receive) **complete call** is needed to **complete** the communication, i.e., to **verify** that the data has been copied out of the send buffer (received into the receive buffer) .

With suitable hardware, the **transfer** of data out of the sender (receiver) memory may **proceed concurrently** with **computations** done at the sender (receiver) after the send (receive) was initiated and before it completed.

The use of **nonblocking receives** may also **avoid system buffering** and **memory-to-memory copying**, as information is provided early on the location of the receive buffer.

NonBlocking Communication (2)

Nonblocking send start calls can use the same four modes as blocking sends: **standard**, **buffered**, **synchronous** and **ready**.

Sends of all modes, **ready** excepted, can be **started whether or not a matching** receive has been posted ;
a nonblocking **ready** send can be started only if a **matching** receive is posted.

In all cases, the send start call is **local**:
it **returns immediately**, irrespective of the status of other processes.

If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in "pathological" cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

The **send-complete call returns** when data has been **copied** out of the send buffer. It may carry **additional meaning**, depending on the **send mode**.

NonBlocking Communication (3)

If the send mode is **synchronous**, then the send can **complete only if a matching receive has started**. That is, a receive has been **posted**, and has been **matched** with the send. In this case, the send-complete call is **non-local**. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender ``knows" the transfer will complete, but before the receiver ``knows" the transfer will complete.)

If the send mode is **buffered** then the message **must be buffered if there is no pending receive**. In this case, the send-complete call is **local**, and must succeed irrespective of the status of a matching receive.

If the send mode is **standard** then the **send-complete** call may **return before a matching** receive occurred, **if the message is buffered**. On the other hand, the **send-complete** may not complete **until a matching** receive occurred, and the message was **copied** into the receive buffer.

Nonblocking sends can be matched with **blocking receives**, and vice-versa.

Send Modes (1)

MPI_Send

MPI_Send will **not return until you can use the send buffer**.
It may or may not block
(it is allowed **to buffer**, either on the sender or receiver side,
or **to wait** for the matching receive).

MPI_Bsend

May **buffer**;
returns immediately and you can **use the send buffer**.
A late add-on to the MPI specification.
Should be used only when absolutely necessary.

MPI_Ssend

will **not return until matching receive posted**

MPI_Rsend

May be **used ONLY if matching receive already posted**.
User responsible for writing a correct program.

Send Modes (2)

MPI_Isend

Nonblocking send. But not necessarily asynchronous.

You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free).

Note also that while the I refers to **immediate**, there is no performance requirement on MPI_Isend. An immediate send **must return** to the user **without requiring a matching receive** at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does **not block waiting for a matching receive**. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.

MPI_IbSEND

buffered nonblocking

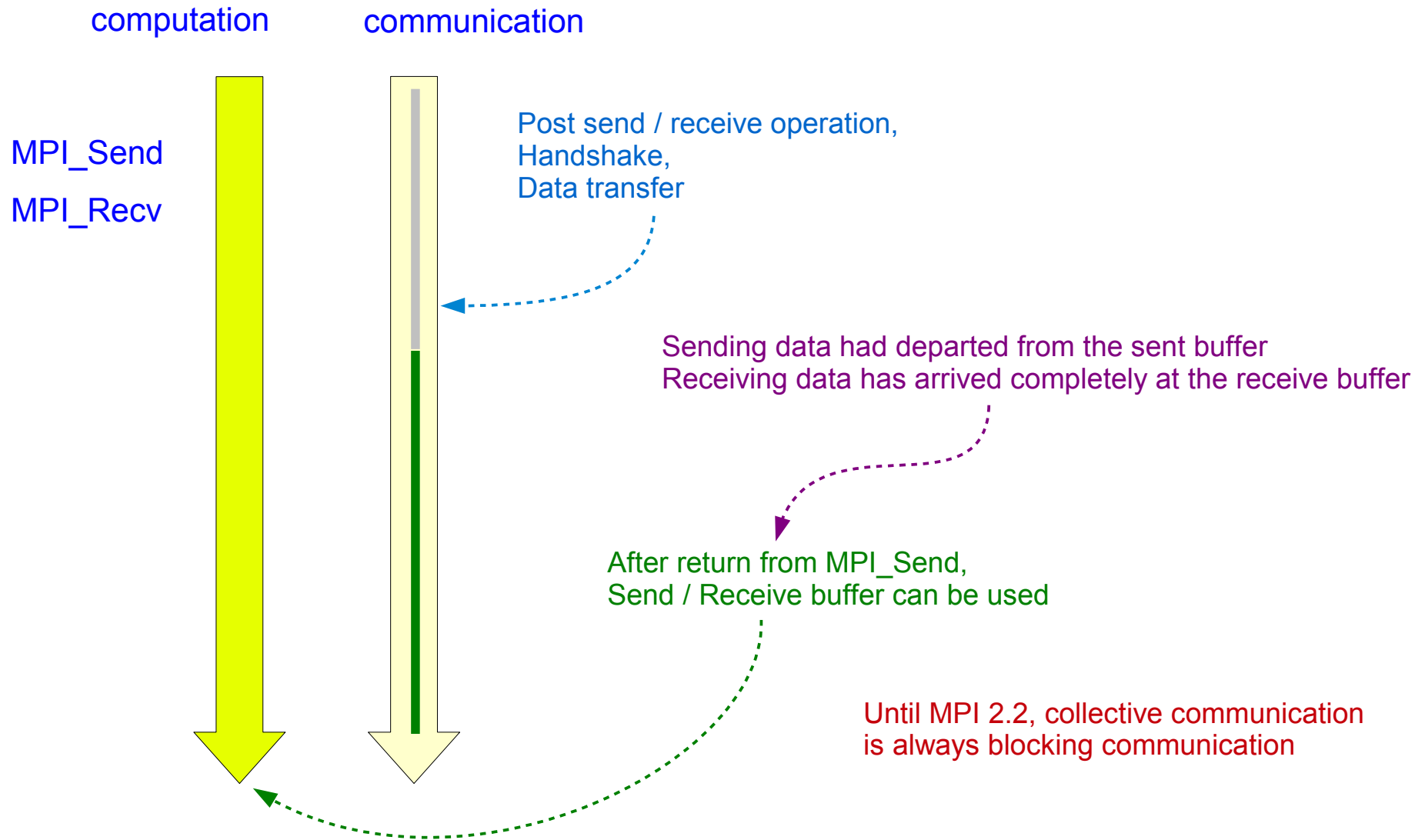
MPI_Issend

Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.

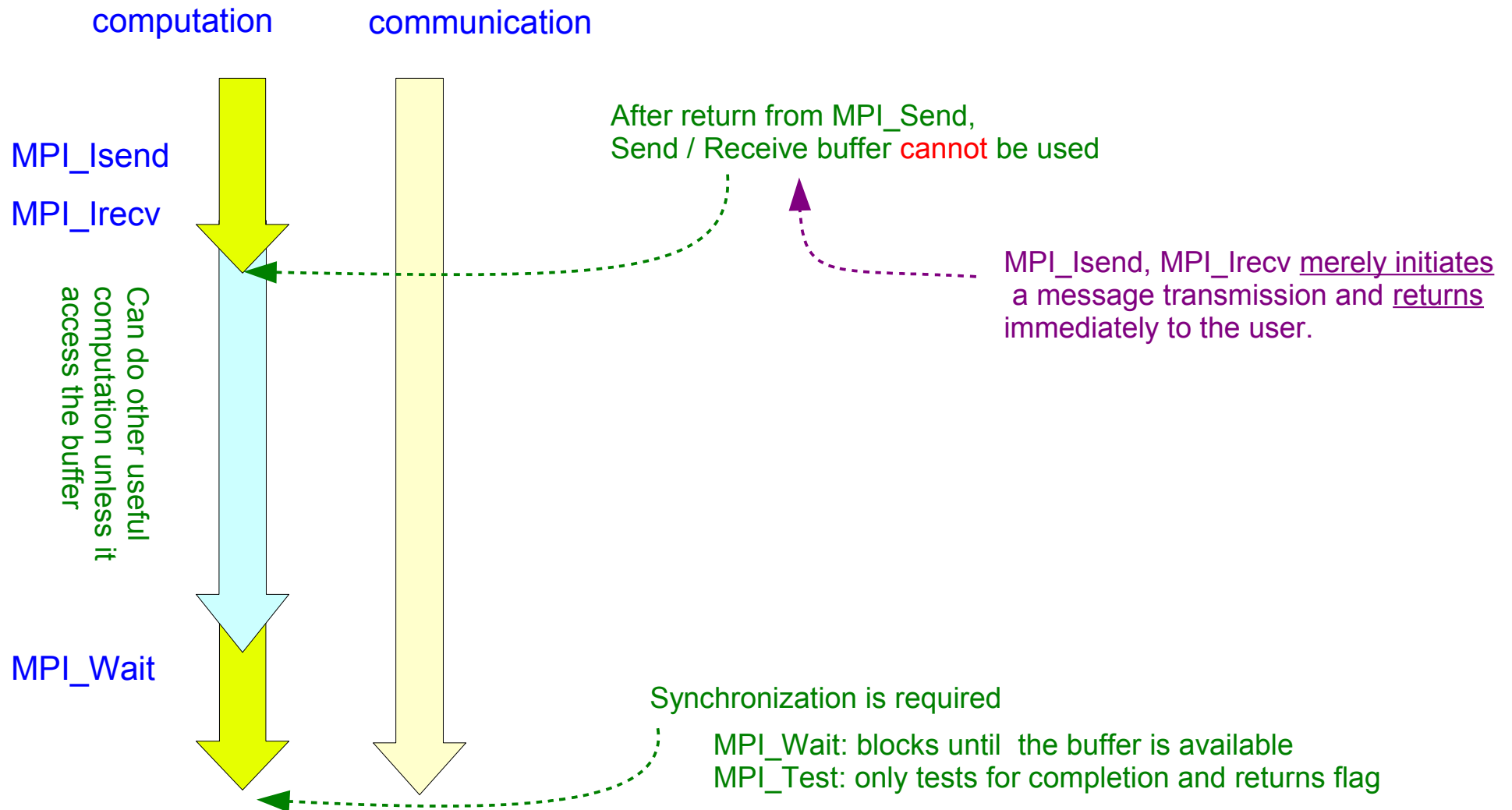
MPI_IrSEND

As with MPI_Rsend, but **nonblocking**.

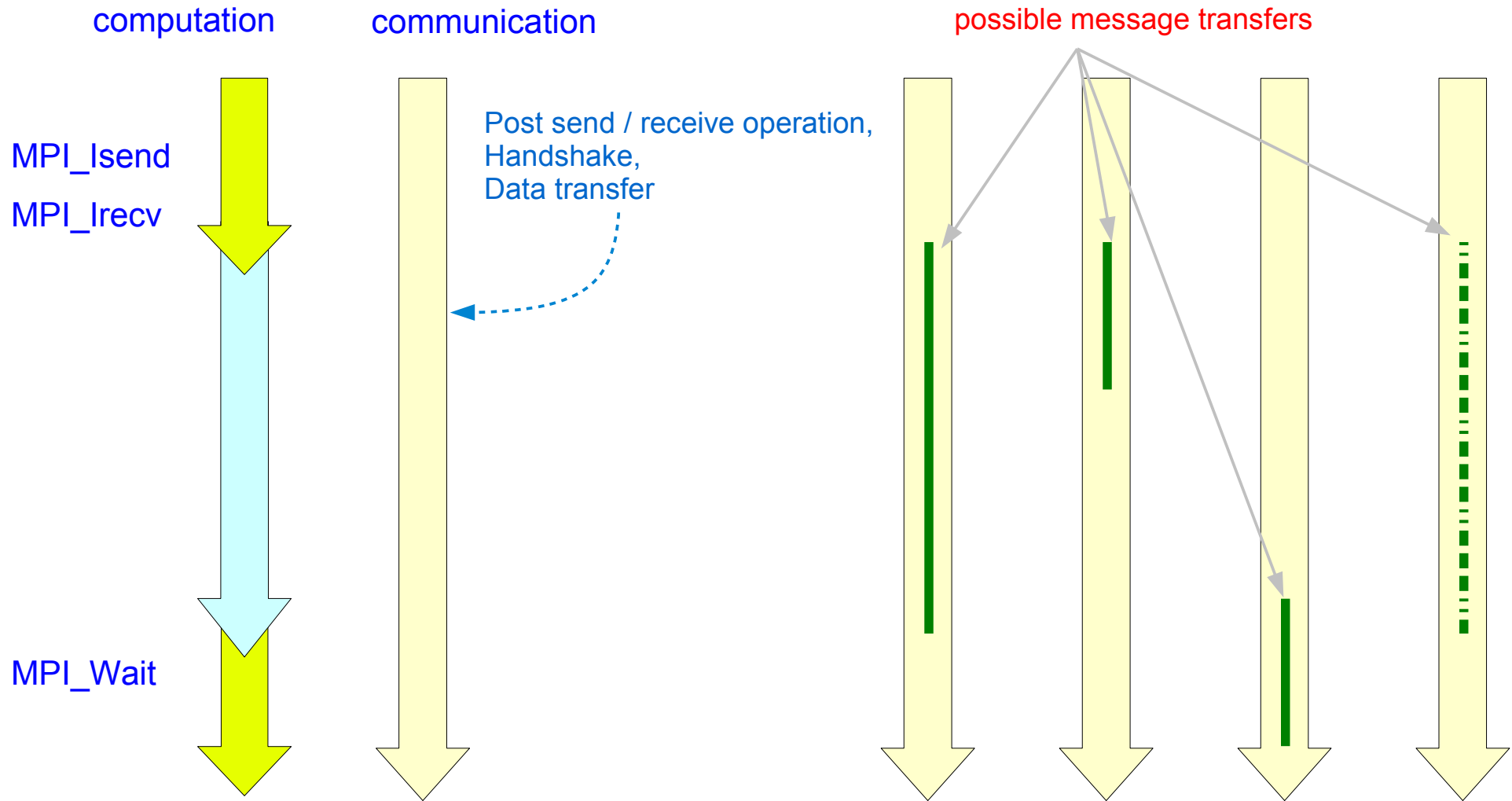
Blocking Communication



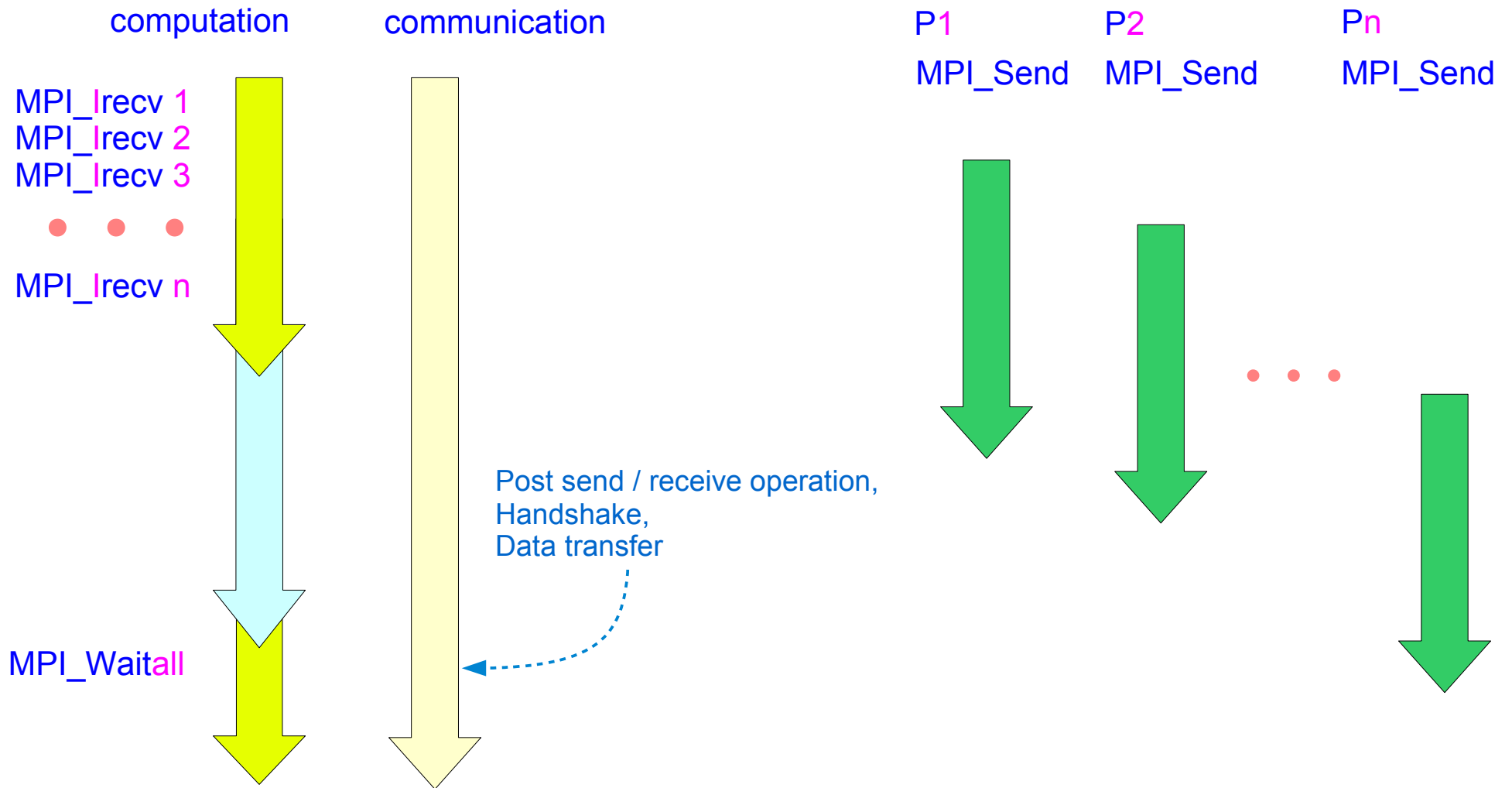
Non-Blocking Communication (1)



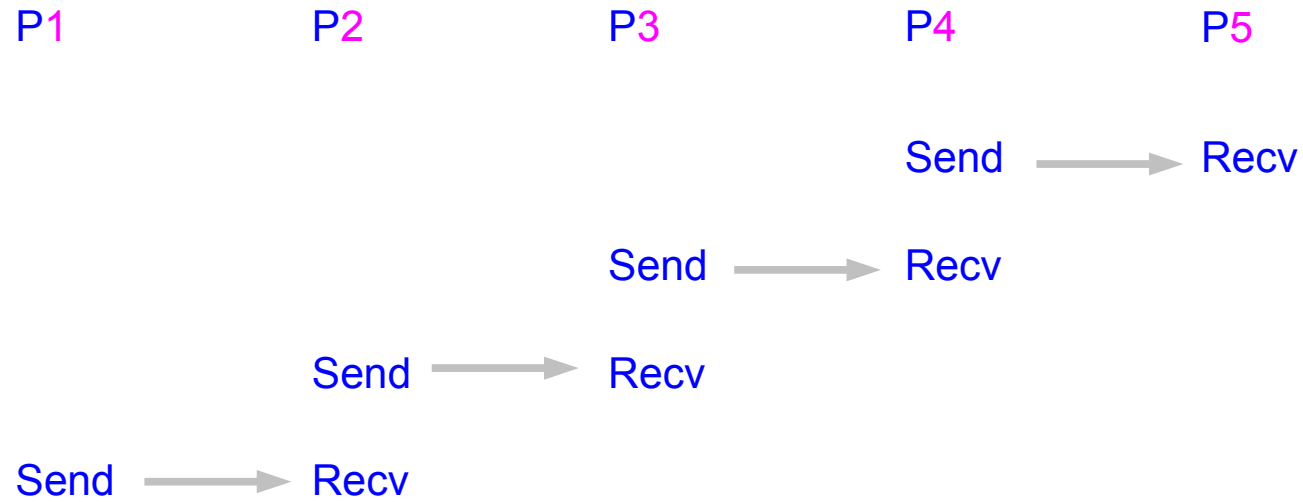
Non-Blocking Communication (2)



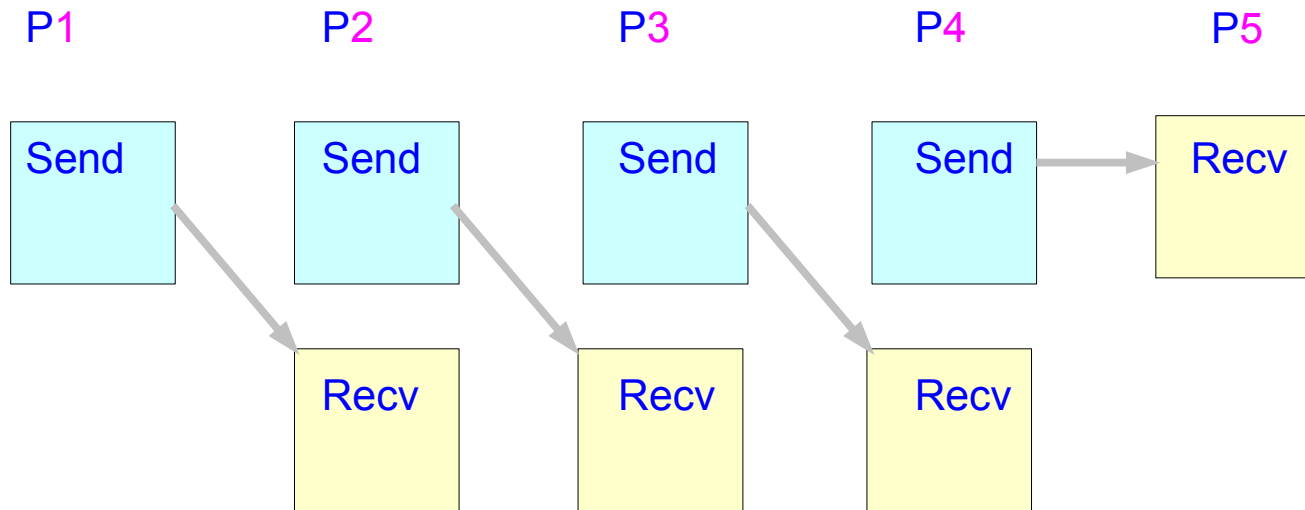
Multiple Request



Implicit Serialization and Synchronization



Implicit Serialization and Synchronization



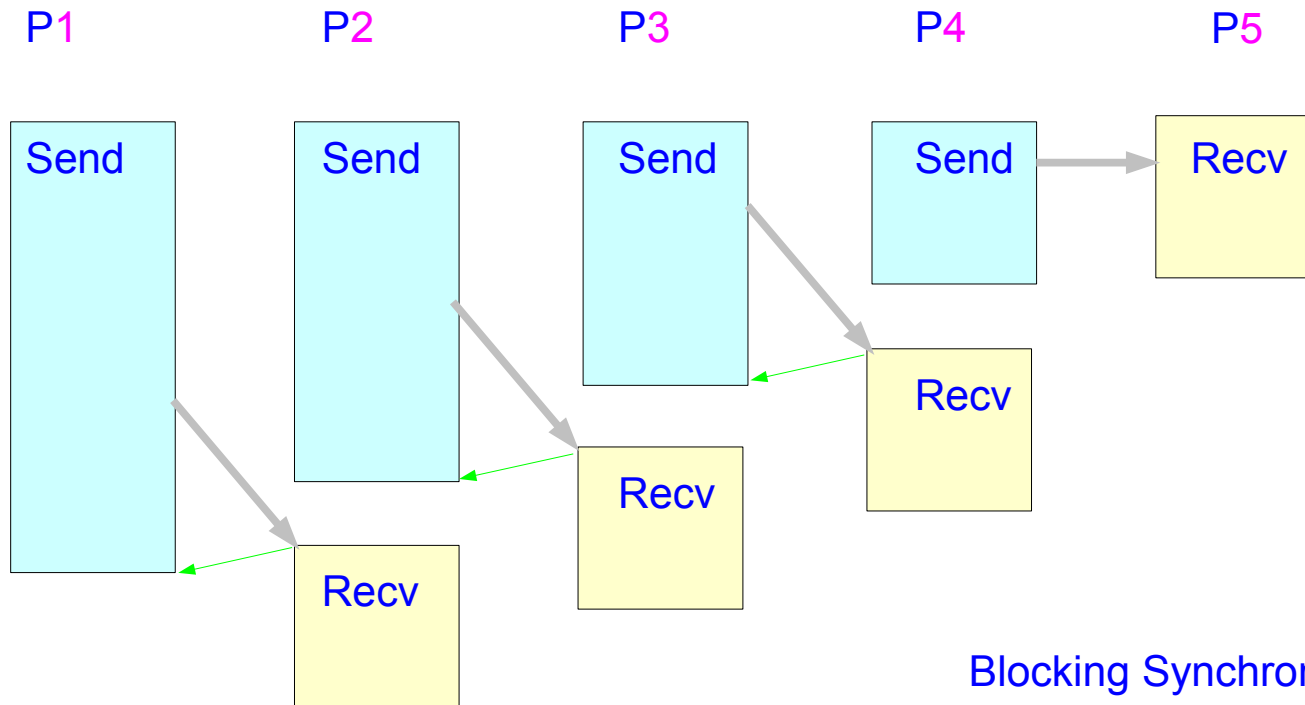
Blocking but not Synchronous Send

Blocking Recv

Eager Delivery

Not Synchronous – enables a send to end before the corresponding recv is posted

Implicit Serialization and Synchronization



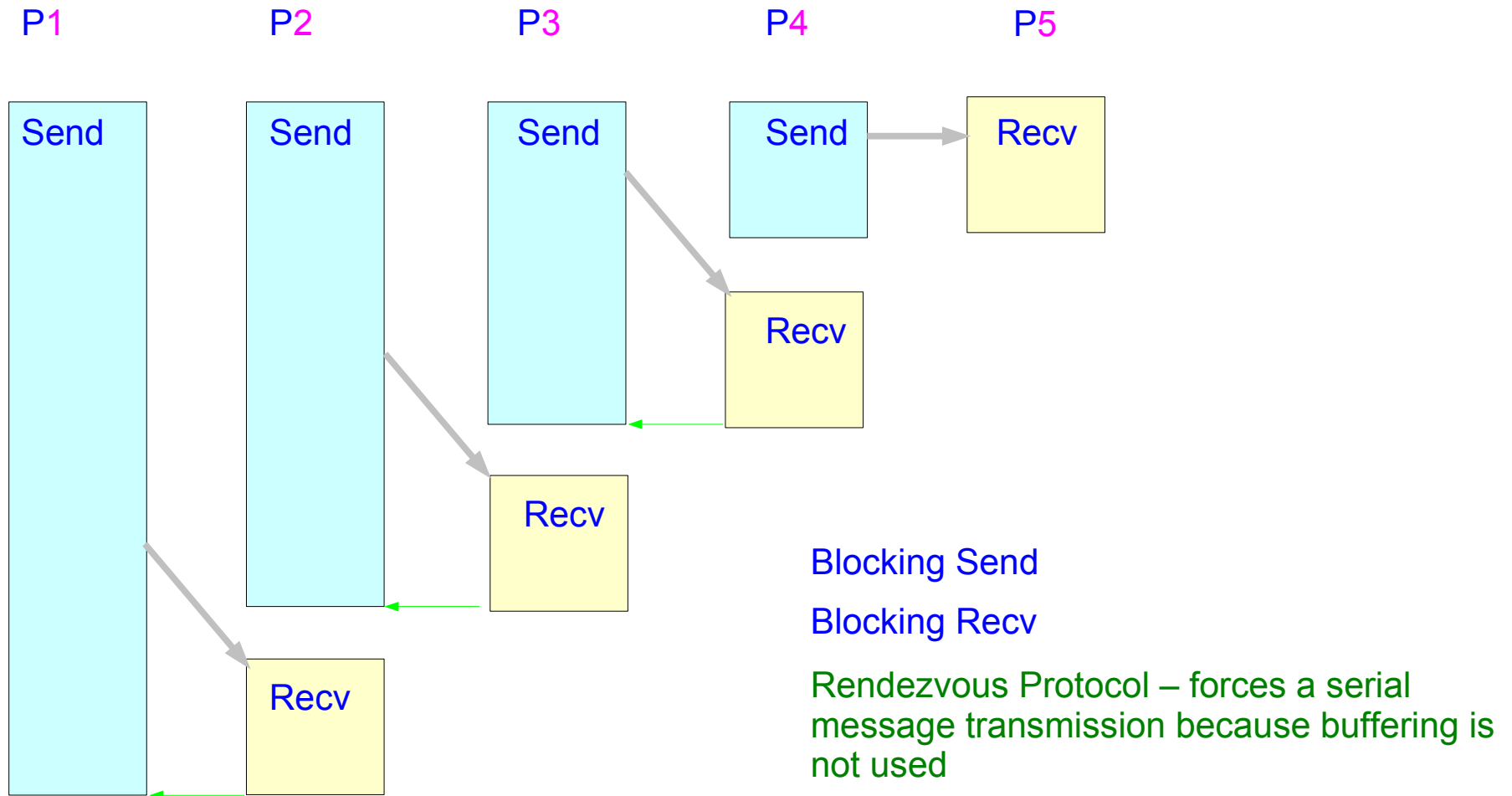
Blocking Synchronous Send

Blocking Recv

Eager Delivery

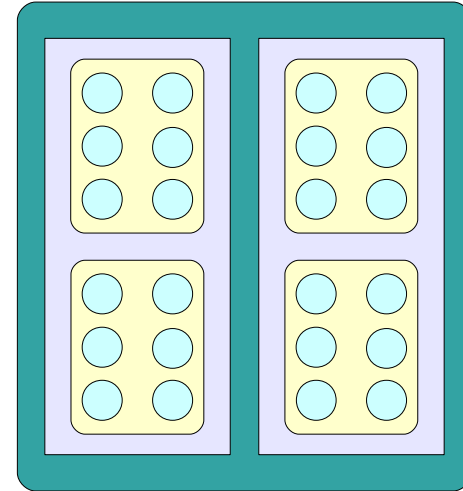
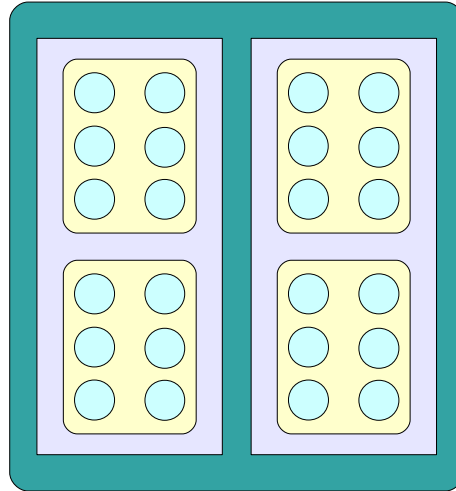
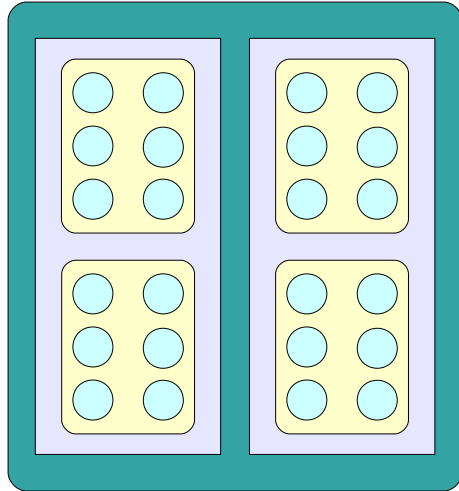
Synchronous – forces a send finish only after it's matching recv is posted

Implicit Serialization and Synchronization



Rank Reorder

PE 0, 1, 2, 3, 4, 5, 6,



Ex) MPICH on CrayPAT

ROUND-ROBIN

SMP-STYLE

FOLDED RANK

MPICH_RANK_REORDER_METHOD

One rank per node, wrap around

Sequential ranks are placed on the next node

Fill up one node before going to next

All cores from all nodes are allocated in a sequential order

One rank per node, wrap back

Rank

- MPI_Comm_size** : Determines the size of the group associated with a communicator
- MPI_Comm_rank** : Determines the rank of the calling process in the communicator
- MPI_Cart_create** : Makes a new communicator to which **topology information** has been attached
- MPI_Dims_create** : Creates a division of processors in a cartesian grid
- MPI_Cart_coords** : Determines **process coords** in cartesian topology given rank in group
- MPI_Cart_rank** : Determines **process rank** in communicator given Cartesian location
- MPI_Cart_shift** : Returns the **shifted source and destination ranks**, given a shift Direction and amount

Rank

int **MPI_Comm_size** (MPI_Comm comm, int *size)

int **MPI_Comm_rank** (MPI_Comm comm, int *rank)

int **MPI_Cart_create** (MPI_Comm comm_old, int ndims, int *dims, int *periods,
int reorder, MPI_Comm *comm_cart)

int **MPI_Dims_create** (int nnodes, int ndims, int *dims)

int **MPI_Cart_coords** (MPI_Comm comm, int rank, int maxdims, int *coords)

int **MPI_Cart_rank** (MPI_Comm comm, int *coords, int *rank)

int **MPI_Cart_shift** (MPI_Comm comm, int direction, int displ, int *source, int *dest)

Rank

```
int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods,  
                    int reorder, MPI_Comm *comm_cart)
```

```
MPI_Cart_create (MPI_COMM_WORLD, // standard communicator  
                2, // two dimensions
```

References

- [1] <http://en.wikipedia.org/>
- [2] http://static.msi.umn.edu/tutorial/scicomp/general/MPI/mpi_coll_new.html
- [3] <https://computing.llnl.gov/tutorials/mpi/>
- [4] <https://computing.llnl.gov/tutorials/mpi/>
- [5] Hager & Wellein, Introduction to High Performance Computing for Scientists and Engineers
- [6] <http://www.mpi-forum.org/docs/mpi-11-html>