MPI Programming Techniques

-
-

Young Won Lim
11/02/2012

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

This document was produced by using OpenOffice and Octave.

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

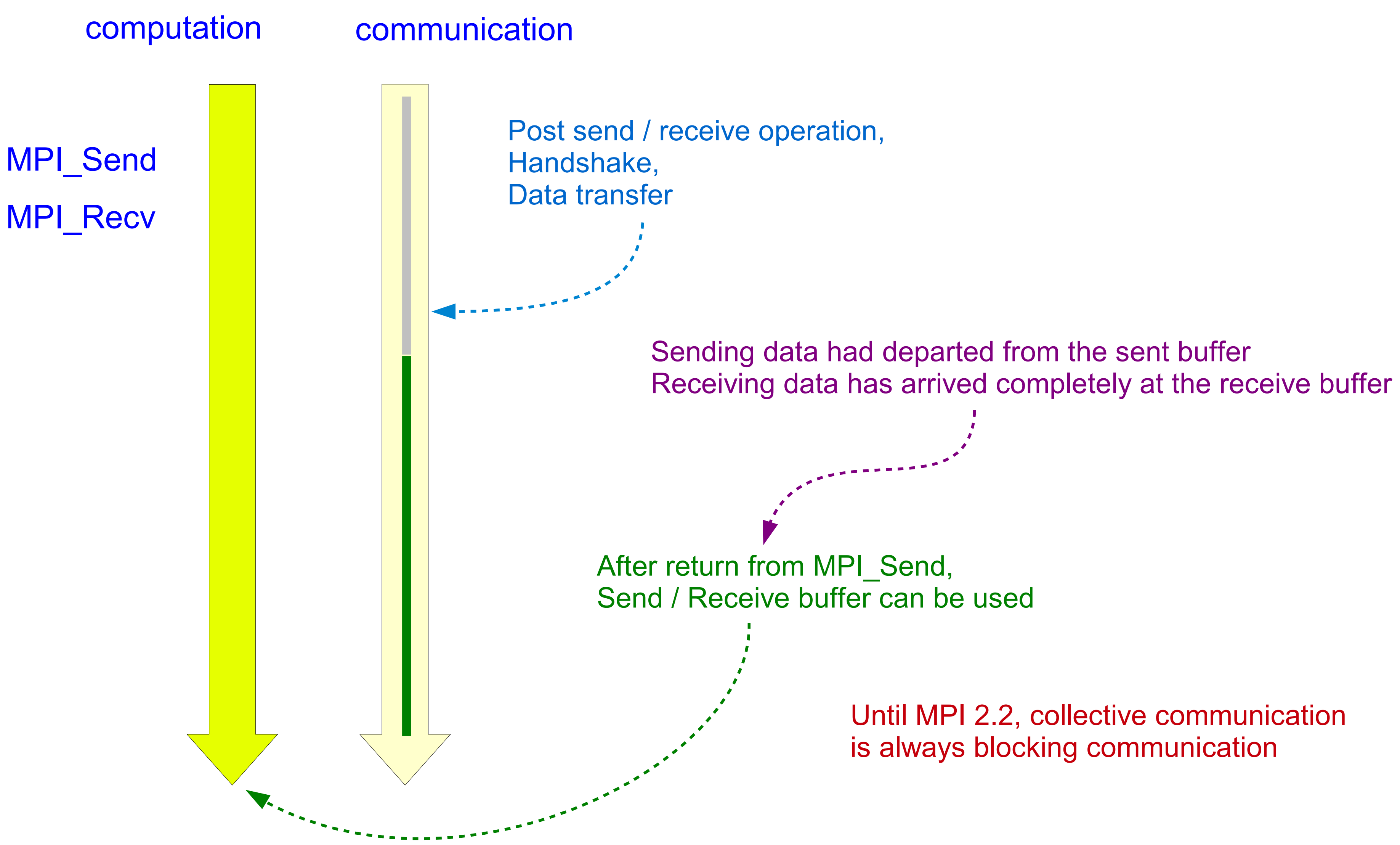
For large messages

Buffering the data is impossible

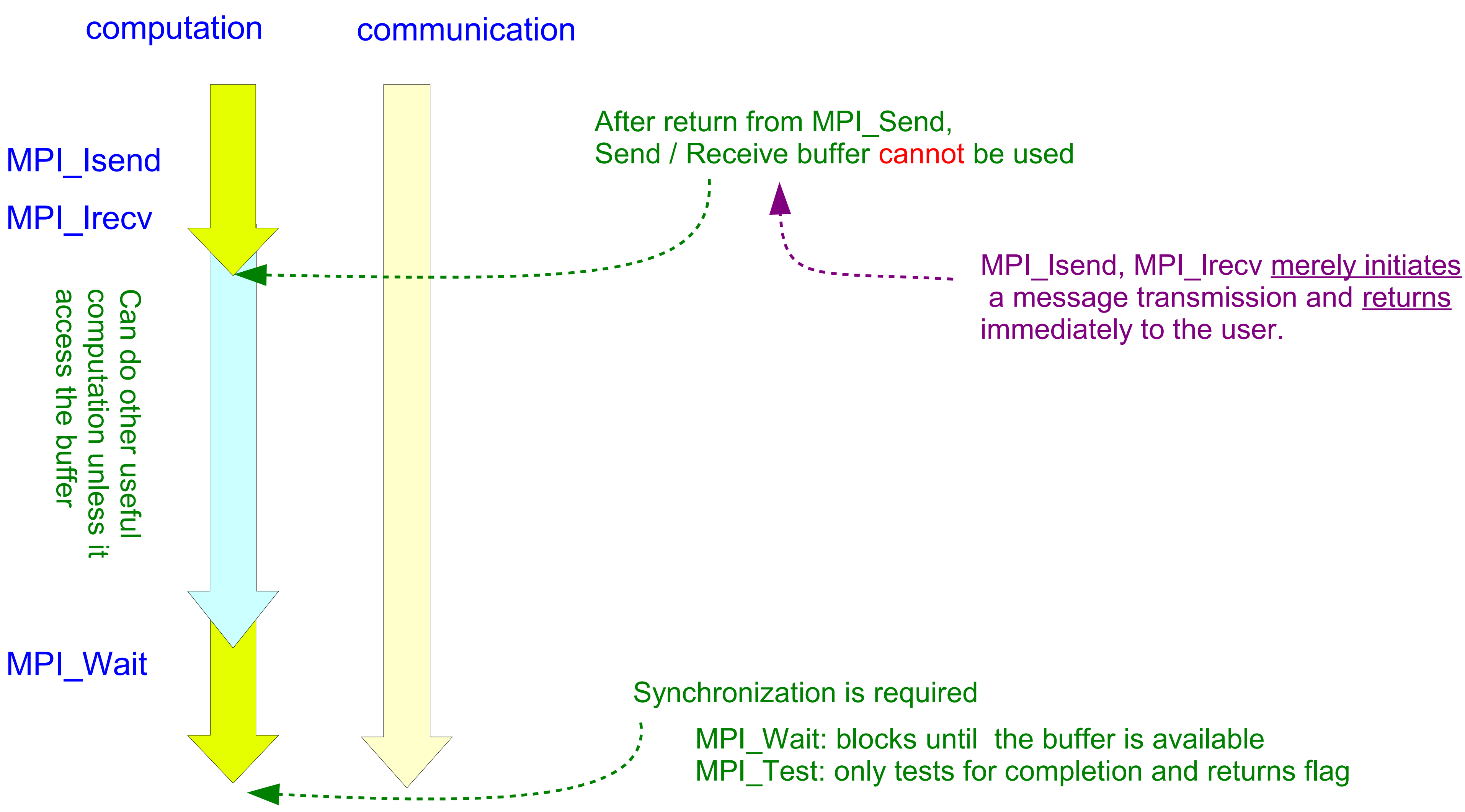
The message envelope is immediately stored at the receiver
The actual message transfer blocks until the user's receive buffer is available

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

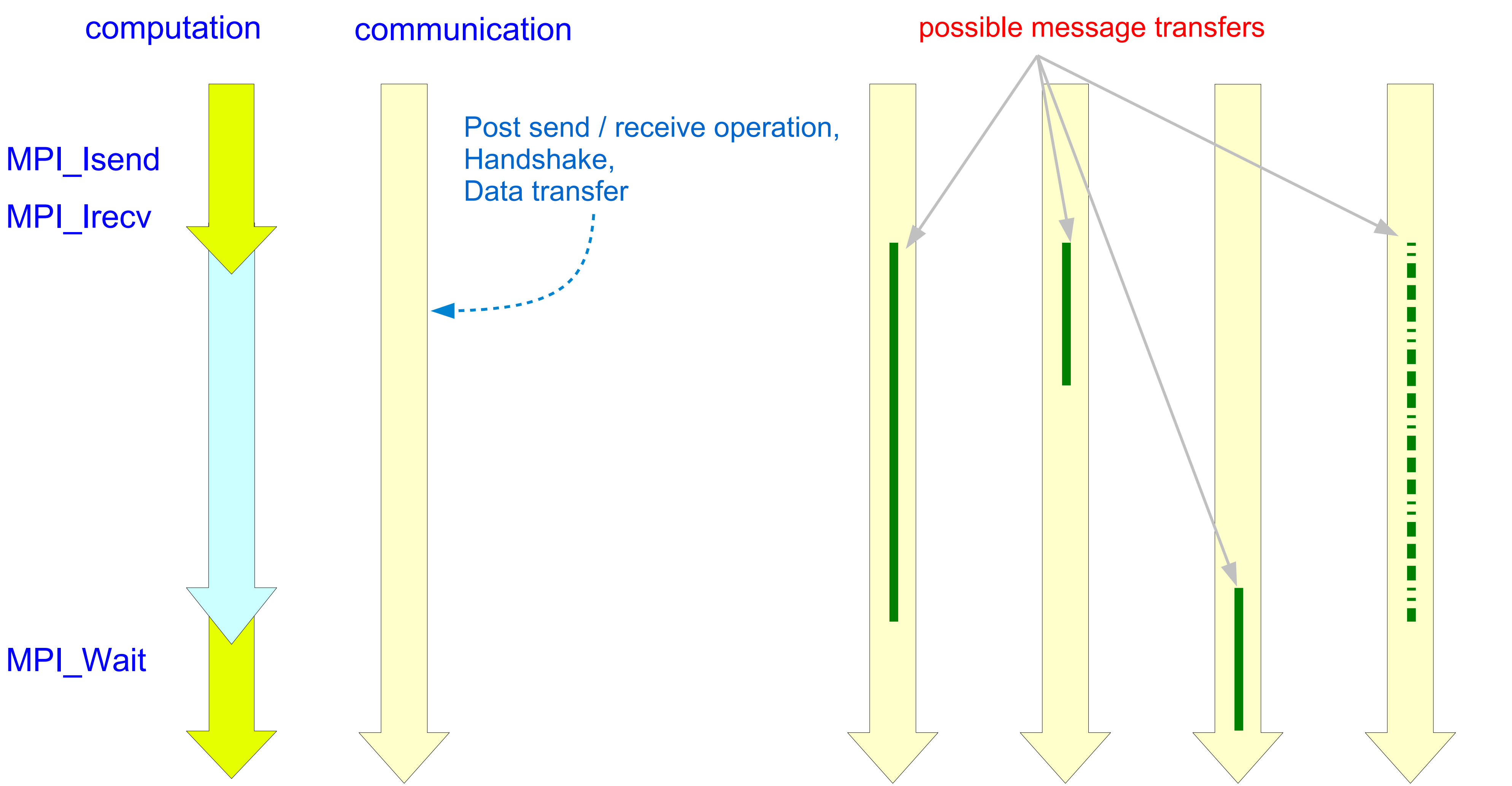
Blocking Communication



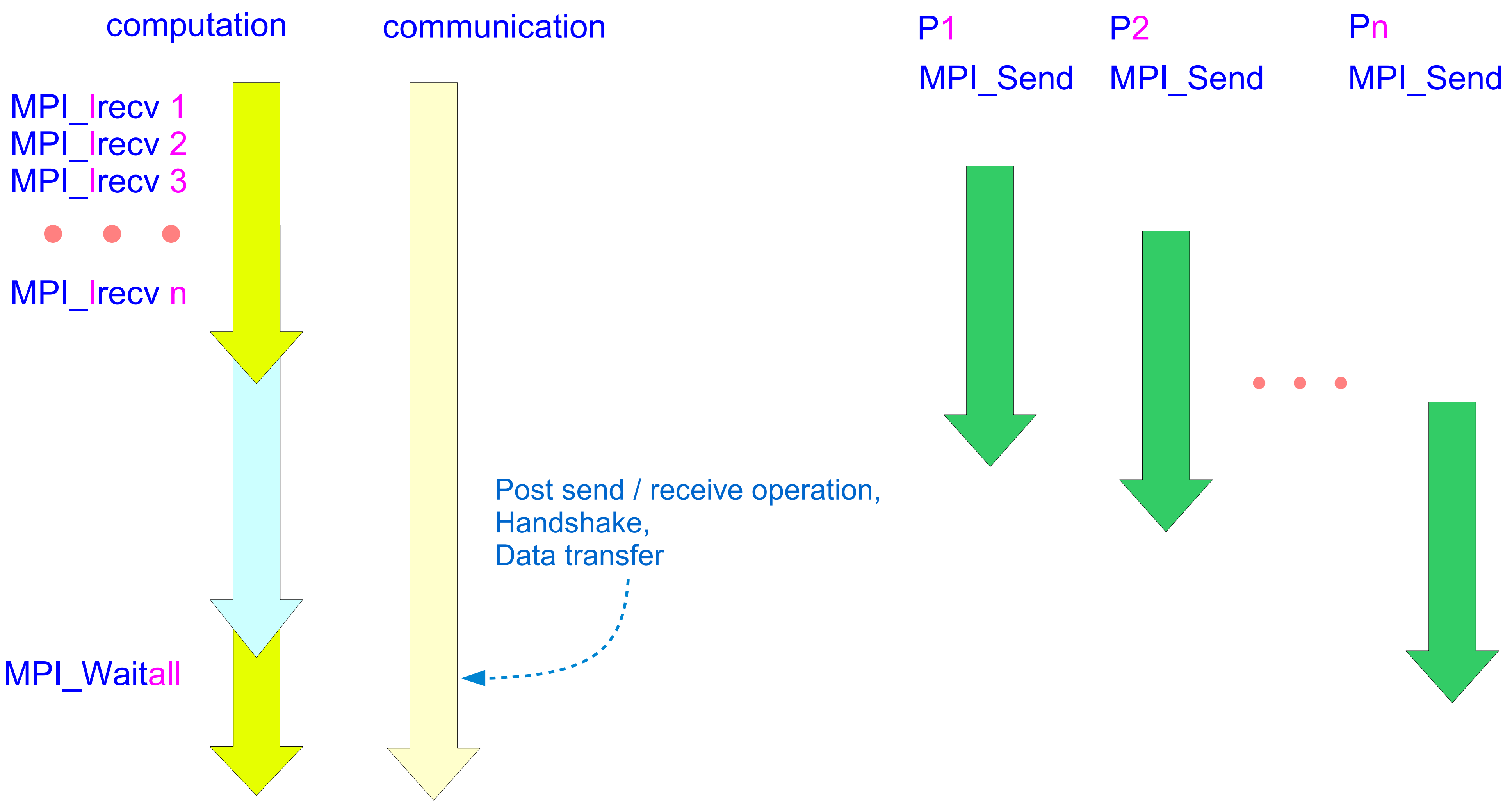
Non-Blocking Communication (1)



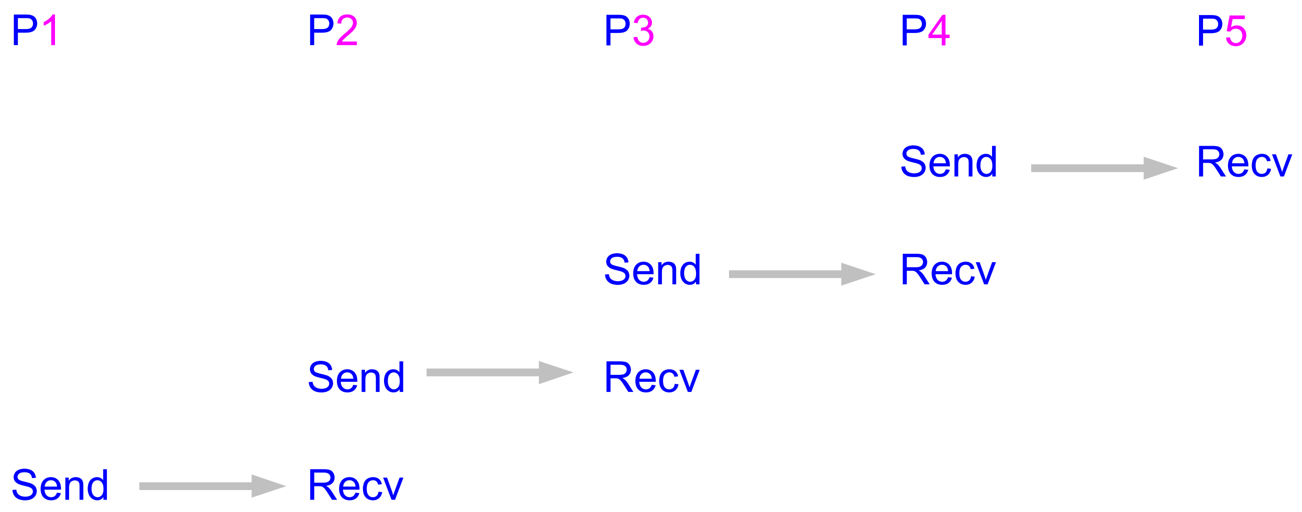
Non-Blocking Communication (2)



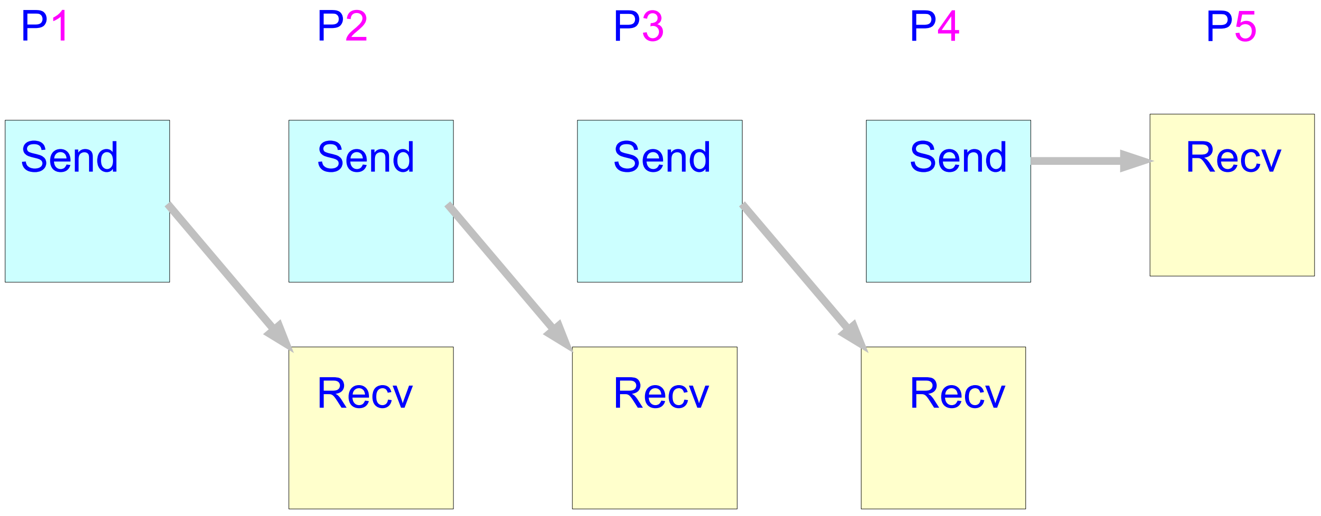
Multiple Request



Implicit Serialization and Synchronization



Implicit Serialization and Synchronization

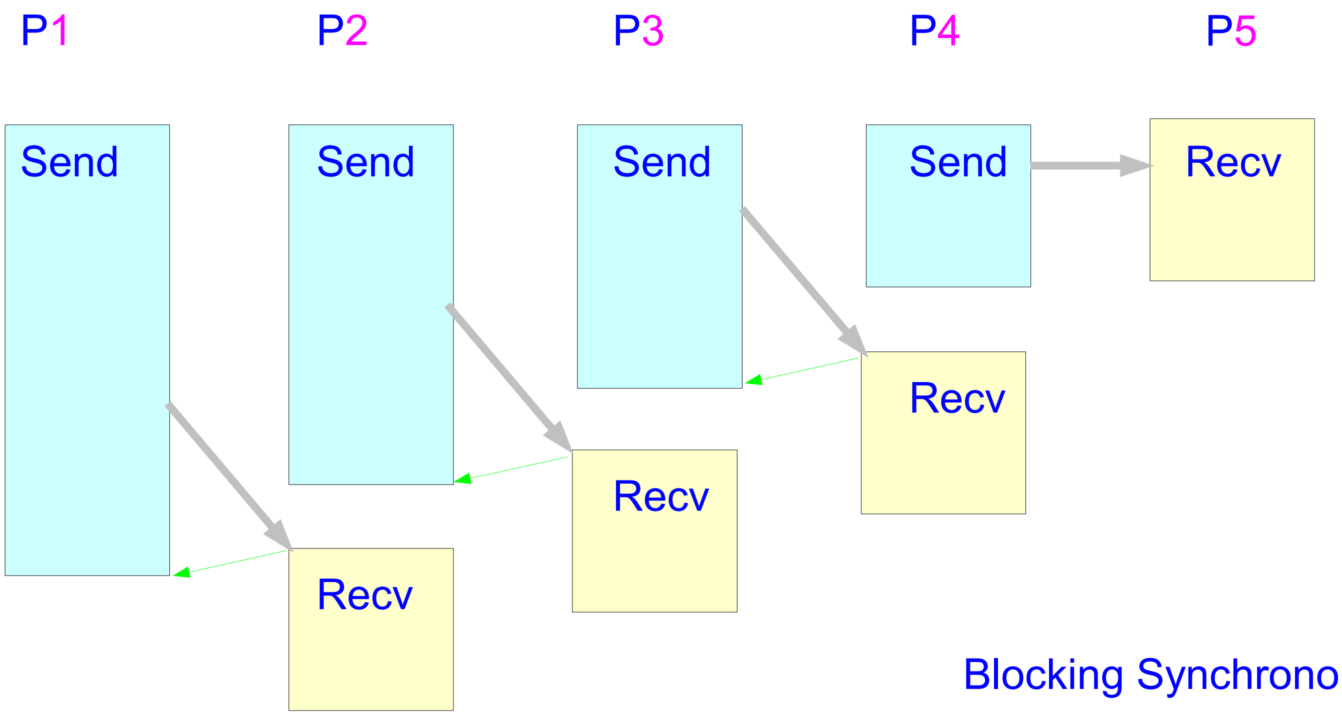


Blocking but not Synchronous Send

Blocking Recv

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

Implicit Serialization and Synchronization

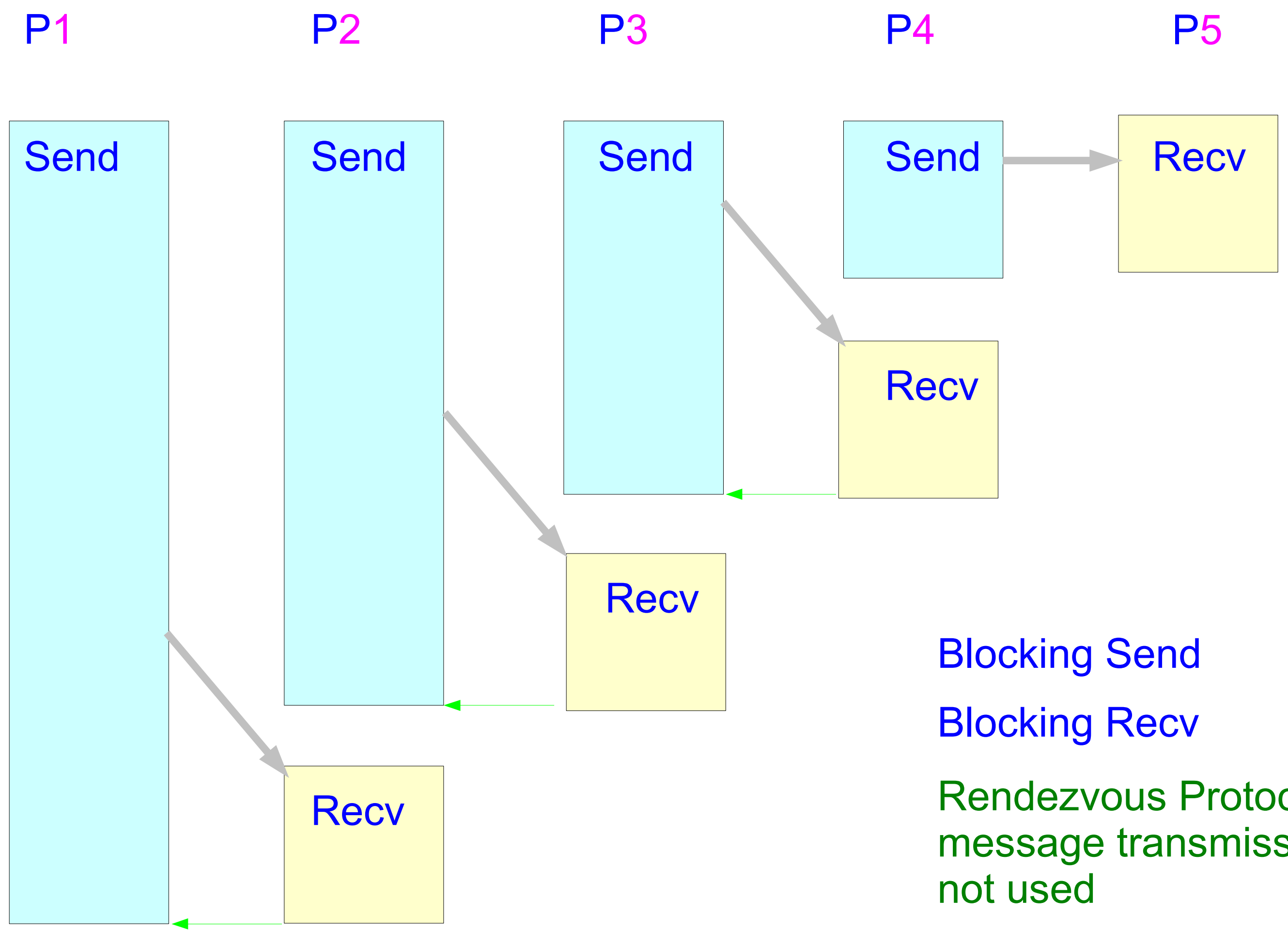


Blocking Synchronous Send

Blocking Recv

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

Implicit Serialization and Synchronization



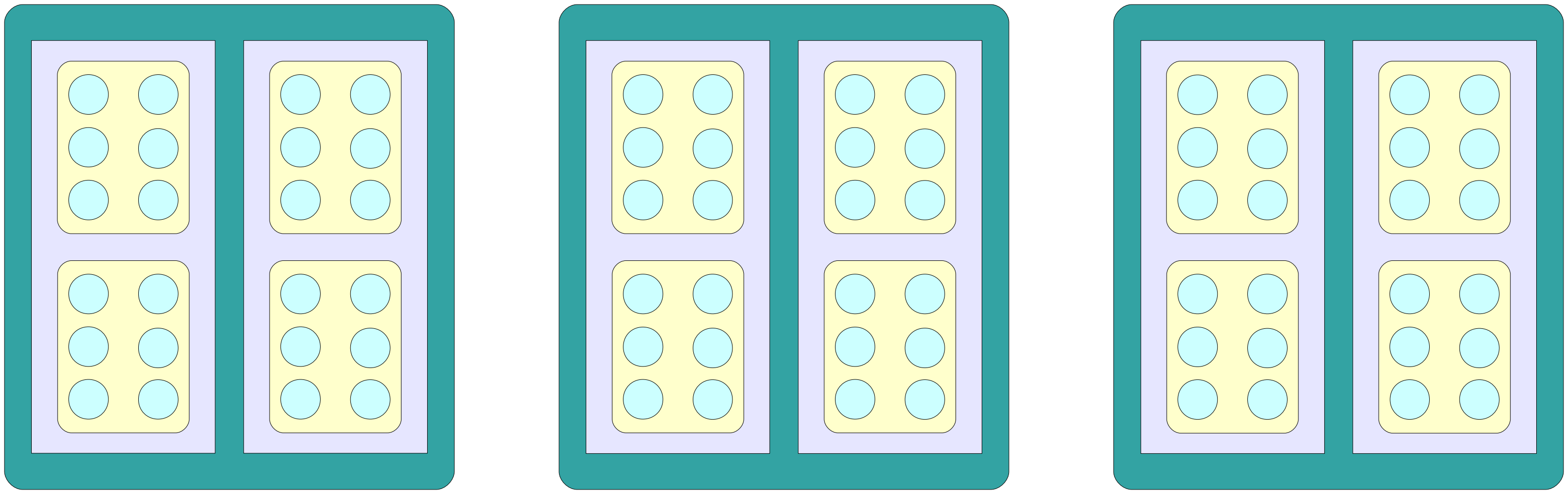
Blocking Send

Blocking Recv

Rendezvous Protocol – forces a serial message transmission because buffering is not used

Rank Reorder

PE 0, 1, 2, 3, 4, 5, 6, ….



Ex) MPICH on CrayPAT

MPICH_RANK_REORDER_METHOD

ROUND-ROBIN One rank per node, wrap around
Sequential ranks are placed on the next node

SMP-STYLE Fill up one node before going to next
All cores from all nodes are allocated in a sequential order

FOLDED RANK 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
```

Nonblocking s. Asynchronous Communication

Nonblocking :

implies that the message buffer cannot be used
after the call has returned from the MPI library.

It depends on the implementation
whether data transfer (MPI progress) takes place
outside MPI while user code is being executed

```
T = MPI_Wtime()

    MPI_Irecv(...);
    do_work( delay );
    MPI_Wait(...);

T = MPI_Wtime() - T;
```

If MPI_Irecv() triggers a truly asynchronous data transfer,

the measured overall time will stay constant with increasing delay until the delay equals the message transfer time. Beyond this point, there will be a linear rise in execution time.

If MPI progress occurs only inside the MPI library
(which means, in this example, within MPI_Wait()),

the time for data transfer and the time for executing do_work() will always add up
and there will be linear rise of overall execution time starting from zero delay

Nonblocking s. Asynchronous Communication

If MPI_Irecv() triggers a truly asynchronous data transfer,

the measured overall time will stay constant with increasing delay until the delay equals the message transfer time. Beyond this point, there will be a linear rise in execution time.

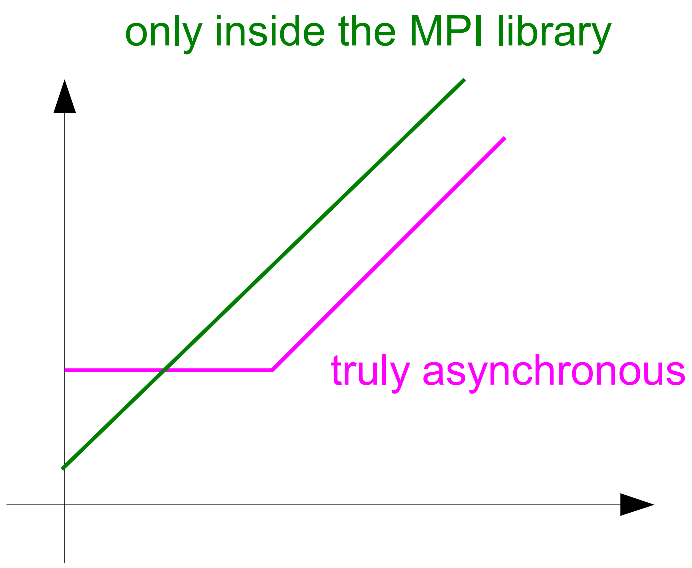
If MPI progress occurs only inside the MPI library (which means, in this example, within MPI_Wait()),

the time for data transfer and the time for executing do_work() will always add up
and there will be linear rise of overall execution time starting from zero delay

```
T = MPI_Wtime()

      MPI_Irecv(...);
      do_work( delay );
      MPI_Wait(...);

T = MPI_Wtime() - T;
```



Intranode point-to-point communication (1)

Cray XT5 system

One XT5 node
– 2 AMD Opteron chips
    With a 2MB quad-core L3 group each
These nodes are connected via 3D torus network

Different Level of
point-to-point communication characteristics

Intranode intrasocket : inside an L3 group
Intranode intersocket : between core on different sockets
Internode : between different nodes

Internode ↔ Intranode : large difference
Intersocket ↔ Intrasocket : similar

Intranode point-to-point communication (2)

False Assumption:
Any intranode MPI communication is infinitely fast.

Depends on the MPI implementation

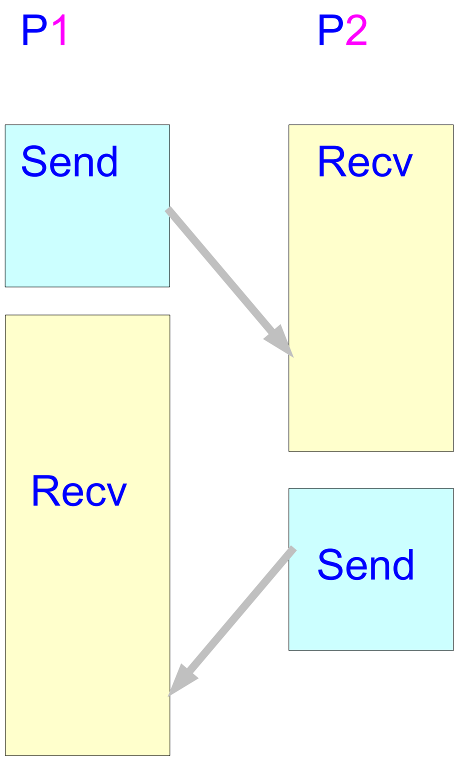
When the MPI library is not aware of intranode communication, relatively slow network protocols are used instead of memory-to-memory copies

Nontemporal stores or cache line zero
Depending on message size and cache sizes
Large message / No shared cache : avoid the write allocate

Single copy (simple block copy command)
From send buf to recv buf
(synchronizing randezvous protocol)
Intermediate buffer (additional copy)

Hardware support for intranode memory-to-memory copy

Ping-Pong Benchmark (1)



Message Size

$$T = T_l + \frac{N}{B}$$

Latency Max Bandwidth

$$B_{eff} = \frac{N}{T} = \frac{N}{T_l + N/B}$$

A multicore processor with a shared cache
- fit into the cache

IMB (Intel Benchmarks)

Ping-Pong Benchmark (2)

$$T = T_l + \frac{N}{B}$$

Small sized message : latency dominating

$$T \approx T_l$$

$$B_{eff} = \frac{N}{T} = \frac{N}{T_l + N/B}$$

Large sized message : effective bandwidth saturating

$$B_{eff} \approx B$$

Measured latency with N=0

May be inaccurate because of the followings:

All protocols have some overhead (headers)

Some protocols have min message size > 1 byte

Involves multiple software layers (added latencies)

May not have optimized low-latency I/O

Different buffering algorithms at a certain message size

Extremely large message must be split into smaller chunks

Ping-Pong Benchmark (3)

$$T = T_l + \frac{N}{B}$$

$$B_{eff} = \frac{N}{T} = \frac{N}{T_l + N/B} = \frac{B}{2}$$

$$T_l = \frac{N_{1/2}}{B} \qquad B T_l = N_{1/2}$$

$$B_{eff}(B, T_l) = \frac{N}{T_l + N/B}$$

$$B_{eff}(\beta B, T_l) = \frac{N}{T_l + N/\beta B}$$

$$\frac{B_{eff}(\beta B, T_l)}{B_{eff}(B, T_l)} = \frac{T_l + N/B}{T_l + N/\beta B} = \frac{1 + N/B T_l}{1 + N/\beta B T_l} = \frac{1 + N/N_{1/2}}{1 + N/\beta N_{1/2}}$$

Whether an increase in maximum network bandwidth by a factor of β is really beneficial for all messages?

Message Aggregation

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