IN3200/IN4200: Chapter 10
Efficient MPI programming

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objectives

- Understand some of the performance characteristics of MPI communication
- Learn some guidelines for efficient MPI programming

How costly is a point-to-point message?

We learned in Chapter 4:

$$T = T_\ell + \frac{M}{B}$$

- M: size of the message [bytes]
- T_ℓ: latency
- B: maximum network point-to-point bandwith [bytes/sec]
- For simplicity, T_ℓ and B were considered as constants in Chapter 4, but they may not be in reality....

Two different protocols

Most MPI implementations switch between different variants, depending on the message size and other factors:

- **Eager protocol:** For short messages, the entire message may be sent and stored at the receiver side in some preallocated buffer space, without the receiver's cooperation.
  - Advantage: synchronization overhead (due to handshaking) is reduced.
  - Disadvantage: need a large amount of preallocated buffer space, too many eager messages may overflow those buffers and lead to contention or program crashes.
- **Rendezvous protocol:** For large messages, the message "envelope" is immediately delivered, but the actual message transfer blocks until the user's receive buffer is available.
  - Advantage: Extra data copies could be avoided, improving effective bandwidth.
  - Disadvantage: sender and receiver must synchronize.

Performance problems in form of synchronization, serialization, contention

Case study: linear shift communication pattern

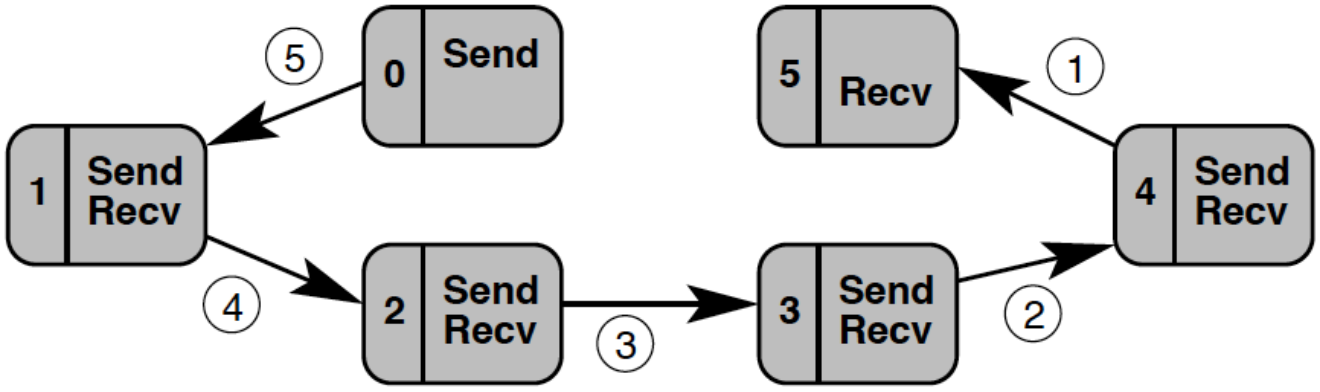


Figure 10.4: A linear shift communication pattern. Even with synchronous point-to-point communication, a deadlock will not occur, but all message transfers will be serialized in the order shown.

- Slightly different from the "ring shift" communication pattern
- "Linear shift" will not suffer from deadlock, but the performance depends on several factors

Linear shift: scenario 1

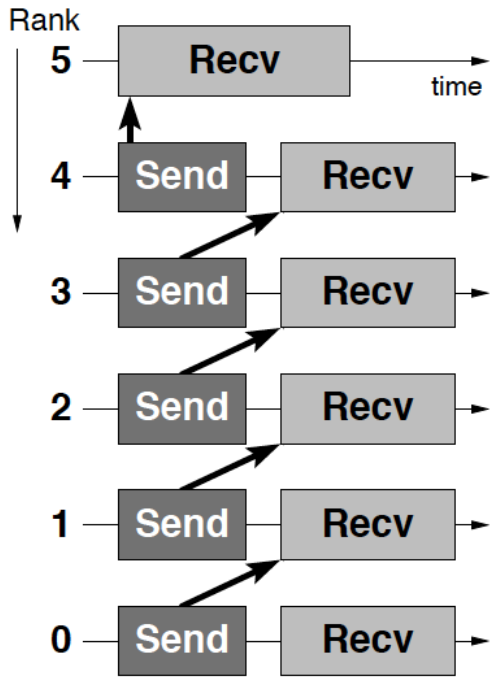


Figure 10.5: Timeline view of the linear shift (see Figure 10.4) with blocking (but not synchronous) sends and blocking receives, using eager delivery. Message transmissions can overlap, making use of a nonblocking network. Eager delivery allows a send to end before the corresponding receive is posted.

- Assuming `MPI_Send()` is not synchronous and "eager delivery";
- Message transfers can overlap if the network is nonblocking;
- All send operations terminate early, most of the time is spent receiving data.

Linear shift: scenario 2

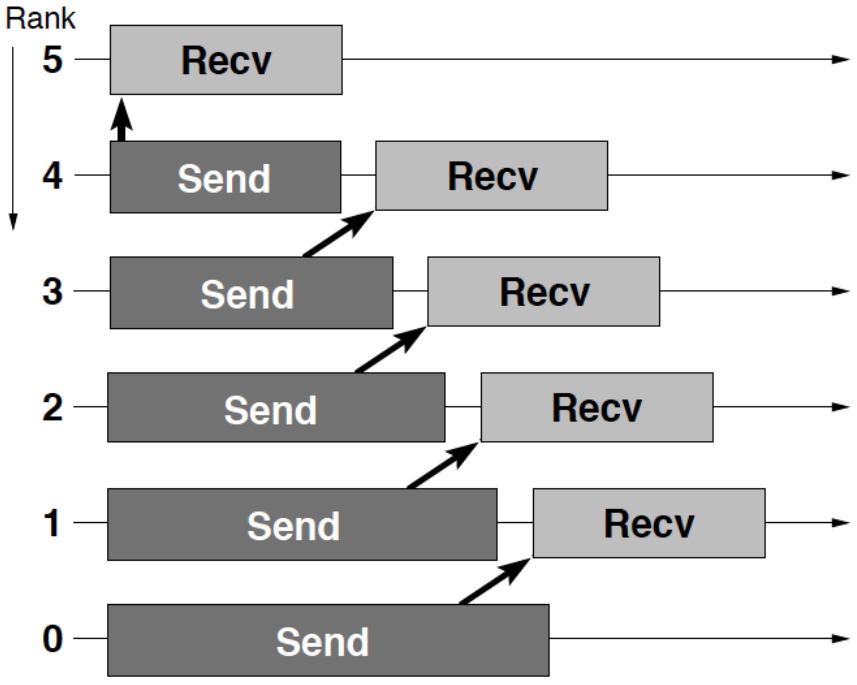


Figure 10.6: Timeline view of the linear shift (see Figure 10.4) with blocking synchronous sends and blocking receives, using eager delivery. The message transfers (arrows) might overlap perfectly, but a send can only finish just after its matching receive is posted.

- The length of the message is large such that `MPI_Send()` is actually executed as `MPI_Ssend()`;
- `MPI_Ssend()` does not return to the user code before a matching receive is posted;
- Message delivery still follows the eager protocol;
- Some performance penalty compared with scenario 1.

Linear shift: scenario 3

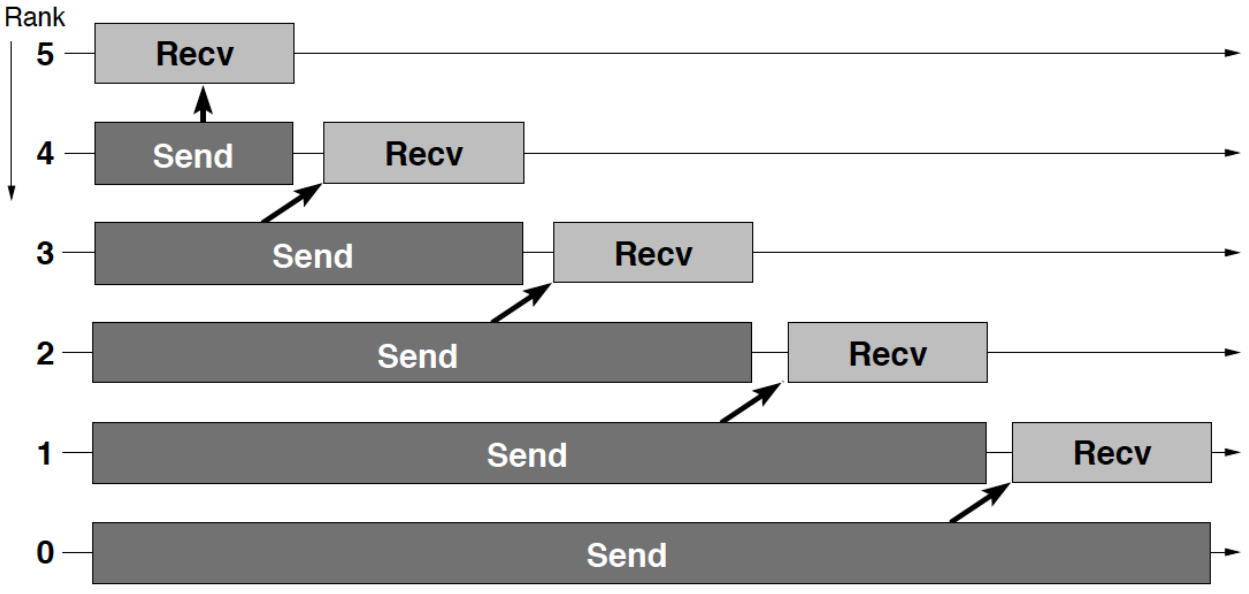


Figure 10.7: Timeline view of the linear shift (see Figure 10.4) with blocking sends and blocking receives, using the rendezvous protocol. The messages (arrows) are transmitted in serial because buffering is ruled out.

- Messages are transmitted according to the rendezvous protocol;
- Sender and receiver must synchronize;
- For the linear shift pattern, messages will be transmitted in serial, one after the other $\rightarrow$ **implicit serialization**

Avoiding implicit serialization

Implicit serialization should be avoided because it is not only a source of additional communication overhead but can also lead to load imbalance.

- Change the order of sends and receives on, e.g., all odd-numbered processes;
- Use nonblocking functions, which have the additional benefit that multiple outstanding communications can be handled by the MPI library in a (hopefully) optimal order. Moreover they provide an opportunity for *asynchronous* communication, where auxiliary threads and/or hardware mechanisms transfer data even while a process is executing user code.
- Use blocking point-to-point functions that are guaranteed not to deadlock, such as `MPI_Sendrecv()` or `MPI_Sendrecv_replace()`.

Example of changing the order of sends/receives

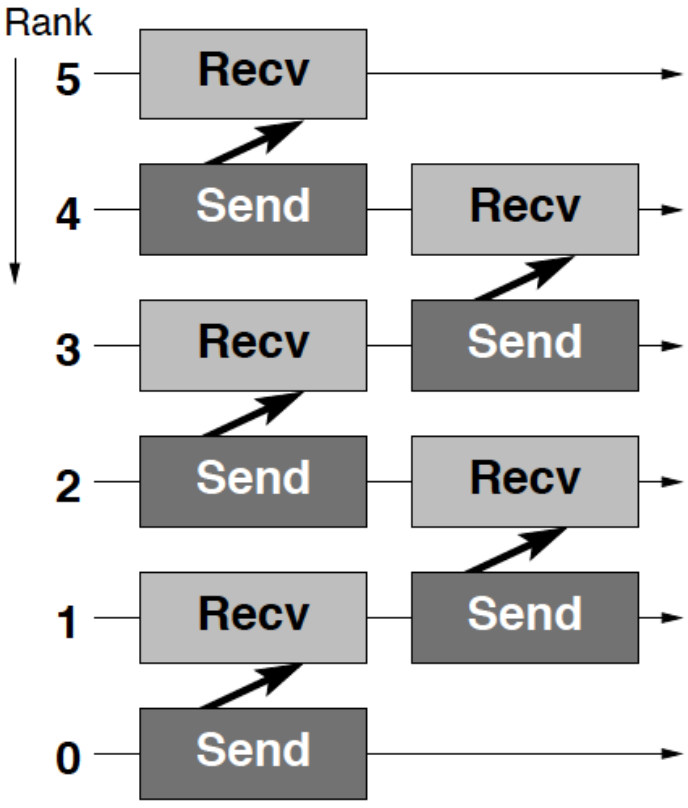


Figure 10.8: Even if sends are synchronous and the rendezvous protocol is used, exchanging the order of sends and receives on all odd-numbered ranks exposes some parallelism in communication.

Network contention

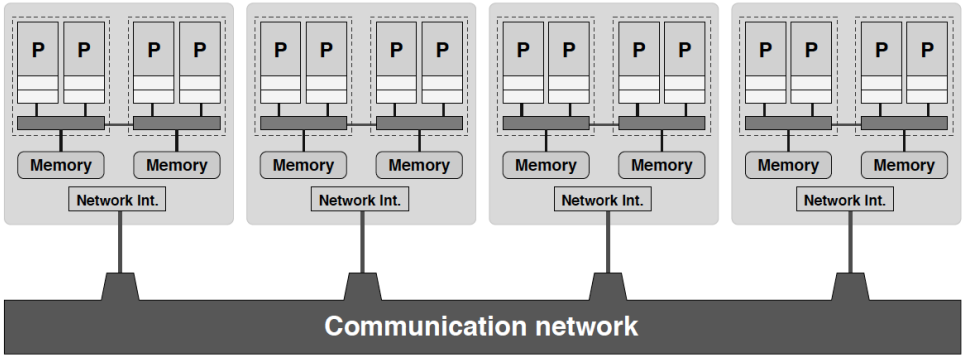


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance "sweet spot" and are thus found in many commodity clusters.

On a typical hybrid (hierarchical) parallel computer:

- Multiple threads or processes on a node may issue communication requests to other nodes. The available bandwidth per connection will typically go down;
- The network topology may not be fully nonblocking, i.e., the bisection bandwidth may be lower than the product of the number of nodes and the single-connection bandwidth;
- Even if bisection bandwidth is optimal, static routing can lead to contention for certain communication patterns.

More on network contention

In general, network contention of some kind is hard to avoid.

- Some communication patterns are especially prone to causing contention, like all-to-all message transmission where every process sends to every other process; MPI's `MPI_Alltoall()` function is a special form of this.
- Any optimization that reduces communication overhead and message transfer volume will most probably also reduce contention.
- Even if there is no way to lessen the amount of message data, it may be possible to rearrange communication calls so that contention is minimized.

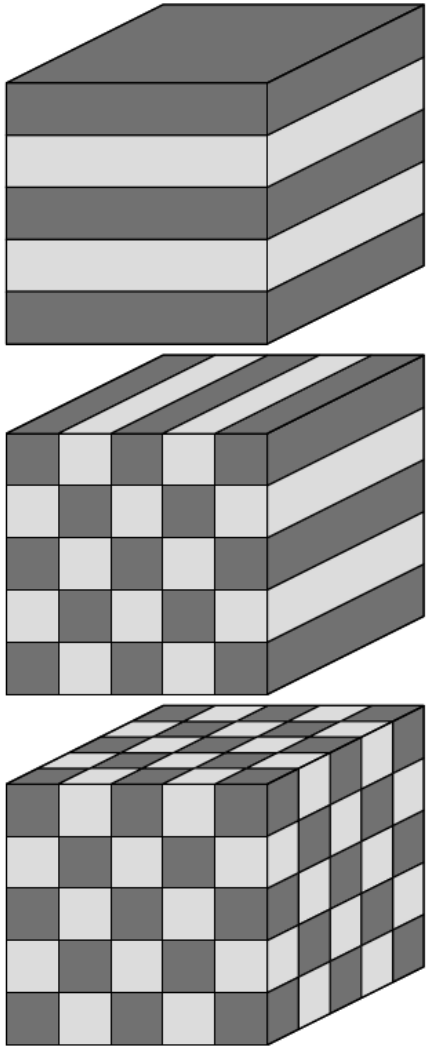
Reducing communication overhead

Domain decomposition

Domain decomposition is one of the most important implementations of data parallelism:

- We want to understand the impact of domain decomposition on communication overhead.
- We want to avoid “wrong” decompositions.

Example: decomposing a 3D domain



"Slabs"

$$c_{1d}(L,N) = L \cdot L \cdot w \cdot 2$$
$$= 2wL^2$$

"Poles"

$$c_{2d}(L,N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2+2)$$
$$= 4wL^2N^{-1/2}$$

"Cubes"

$$c_{3d}(L,N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2+2+2)$$
$$= 6wL^2N^{-2/3}$$

Figure 10.9: 3D domain decomposition of a cubic domain of size L^3 (strong scaling) and periodic boundary conditions: Per-process communication volume $c(L,N)$ for a single-site data volume w (in bytes) on N processes when cutting in one (top), two (middle), or all three (bottom) dimensions.

The amount of communication depends on decomposition

The cubic domain is of size L^3, which has periodic boundary conditions, is decomposed into N subdomains:

- The domain cuts can be performed in one, two, or all three dimensions;
- The number of elements that must be communicated by one process with its neighbors, $c(L, N)$, can be found as simple formulas;

Decomposing as "slabs"

- 1D cuts

$$c_{1\mathrm{d}}(L, N) = L \cdot L \cdot w \cdot 2 = 2wL^2$$

- Should not be used in general because they incur a much larger and, more importantly, N-independent overhead (doesn't decrease when N grows).
- A constant cost of communication per subdomain will greatly harm strong scalability, because performance saturates at a lower level, determined by the message size (i.e., the slab area) instead of the latency.

Decomposing as "poles"

- 2D cuts

$$c_{2\mathrm{d}}(L, N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2 + 2) = 4w\frac{L^2}{\sqrt{N}}$$

- The volume of communication per process decreases with increasing N;
- However, the surface-to-volume ratio will grow with N.

Decomposing as "cubes"

- 3D cuts

$$c_{3\mathrm{d}}(L,N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2+2+2) = 6w\frac{L^2}{\left(\sqrt[3]{N}\right)^2}$$

- Smaller communication volume per process than 2D decomposition (as long as $\sqrt[6]{N} > \frac{3}{2}$);
- Still has the same problems as with 2D decomposition;
- If latency dominates the overhead, "optimal" 3D decomposition may even be counterproductive because of the six neighbors for each domain.
- **Another factor for consideration:** cost of message packing & unpacking

Mapping issues

- Modern parallel computers are inevitably of the hierarchical type. They all consist of shared-memory multiprocessor "nodes" coupled via some network;
- The simplest way to use this kind of hardware is to run one MPI process per core;
- Assuming that any point-to-point MPI communication between two cores located on the same node is much faster than between cores on different nodes, the **mapping of computational subdomains to cores** has a large impact on communication overhead;
- `MPI_Cart_create()` with rank reordering may help, but the interconnect topology may **not** be avaiable for `MPI_Cart_create()`.

An example of 2D decomposition for a 2D domain

Specific case: $N = 32$ MPI processes that form a virtual 4×8 Cartesian grid.
Hardware: 4 nodes each with 8 CPU cores.

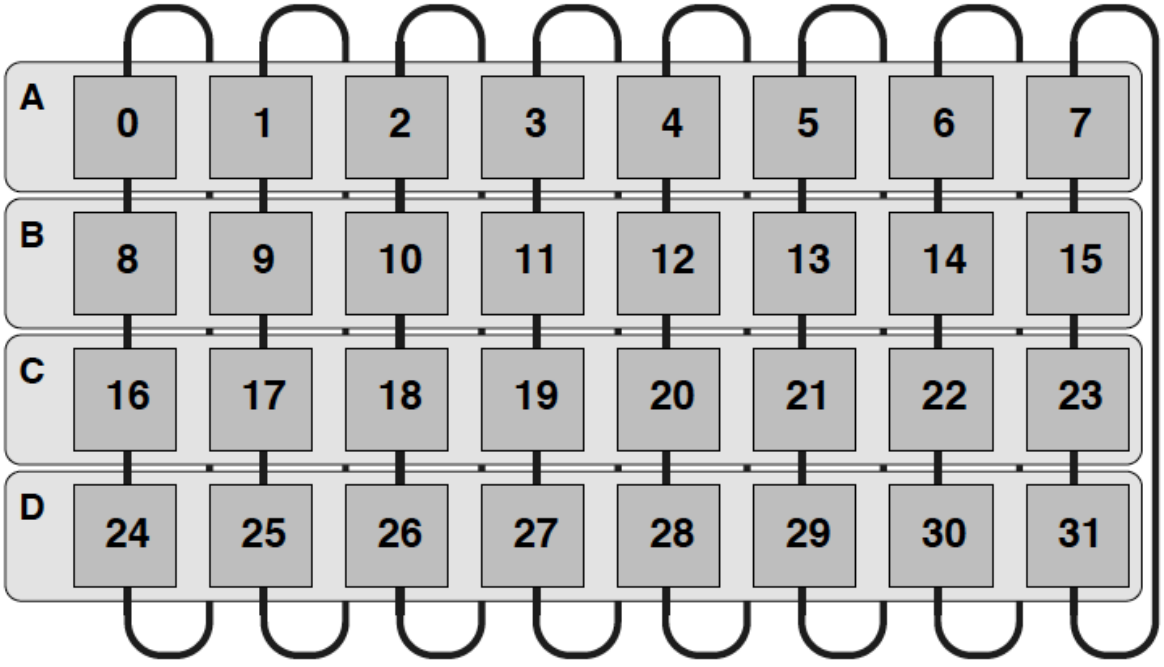


Figure 10.10: A typical default mapping of MPI ranks (numbers) to sub-domains (squares) and cluster nodes (letters) for a two-dimensional 4×8 periodic domain decomposition. Each node has 16 connections to other nodes. Intranode connections are omitted.

- A straightforward mapping will lead to 16 inter-node "connections" per node.

If a different numbering of the subdomains is chosen

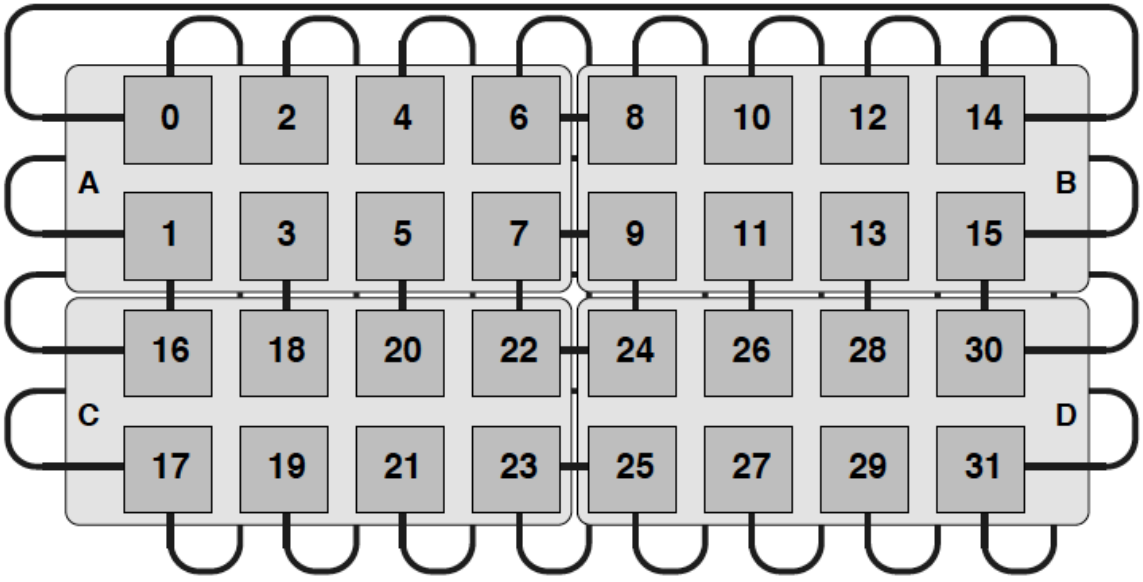


Figure 10.11: A "perfect" mapping of MPI ranks and subdomains to nodes. Each node has 12 connections to other nodes.

- Now, the number of inter-node "connections" is reduced to 12 per node;
- A 25% reduction in inter-node communication volume.

A bad mapping example

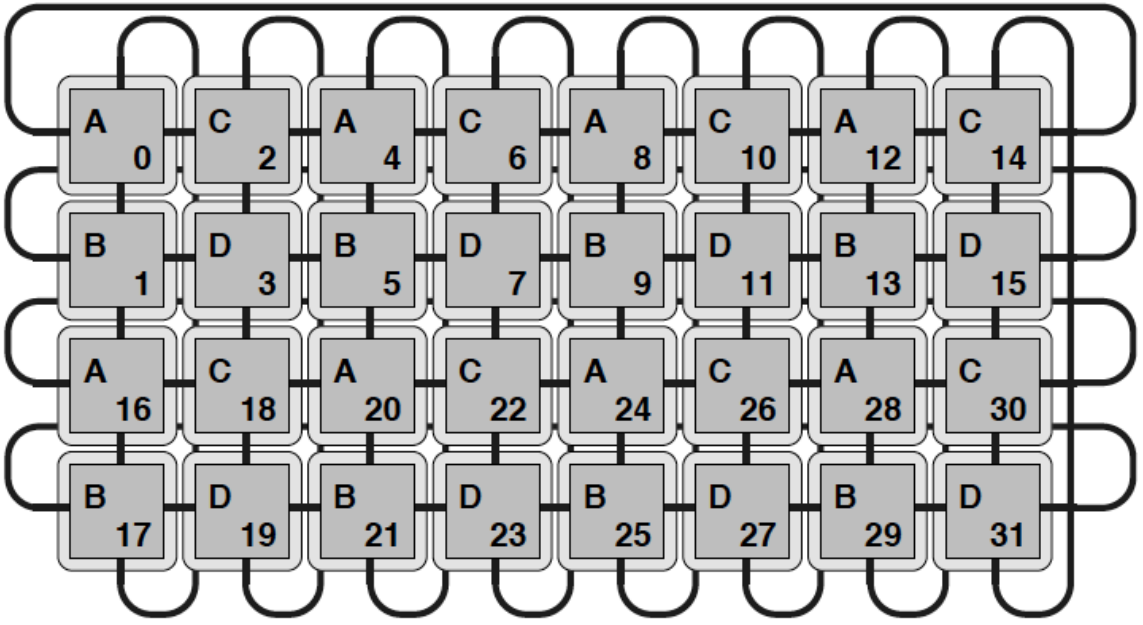


Figure 10.12: The same rank-to-subdomain mapping as in Figure 10.11, but with ranks assigned to nodes in a "round-robin" way, i.e., successive ranks run on different nodes. This leads to 32 internode connections per node.

- In the previous two cases, we have assumed that successsive MPI ranks are mapped to the same node.
- In case the MPI ranks are mapped to the nodes in a *round-robin* distribution, then the inter-node connections per node can be very bad: **32**

Aggregating messages

- If a parallel algorithm requires transmission of a lot of small messages between processes, communication becomes latency-bound because each message incurs latency.
- Hence, small messages should be aggregated into contiguous buffers and sent in larger chunks so that the "latency penalty" must only be paid once, and effective communication bandwidth is good.
- This advantage pertains to point-to-point and collective communication alike.

Things to consider...

Aggregation will only pay off if the additional time for copying the messages to a contiguous buffer does not outweigh the "latency penalty" for separate sends:

$$(m-1)\,T_\ell > \frac{mL}{B_c}$$

- m is the number of messags
- L is the message length
- B_c is the bandwidth for memory-to-memory copies
- For simplicity we assume that all messages have the same length, and that latency for memory copies is negligible.

The actual advantage of message aggregation

The actual advantage depends on the raw network bandwidth B_n as well.

- T_s: "serialized" communication time

$$T_s = m\left(T_\ell + \frac{L}{B_n}\right)$$

- T_a: "aggregated" communicatin time

$$T_a = \frac{mL}{B_c} + T_\ell + \frac{mL}{B_n}$$

- On a slow network, where B_n is very low, the $\frac{T_s}{T_a}$ ratio will be close one, that is, aggregation will not be beneficial.

Need for message aggregation

- A typical case for message aggregation comes up when separate, i.e., noncontiguous data items must be transferred between processes.
- Example: x-direction communication related to 2D domain decomposition—need to send and receive "columns".

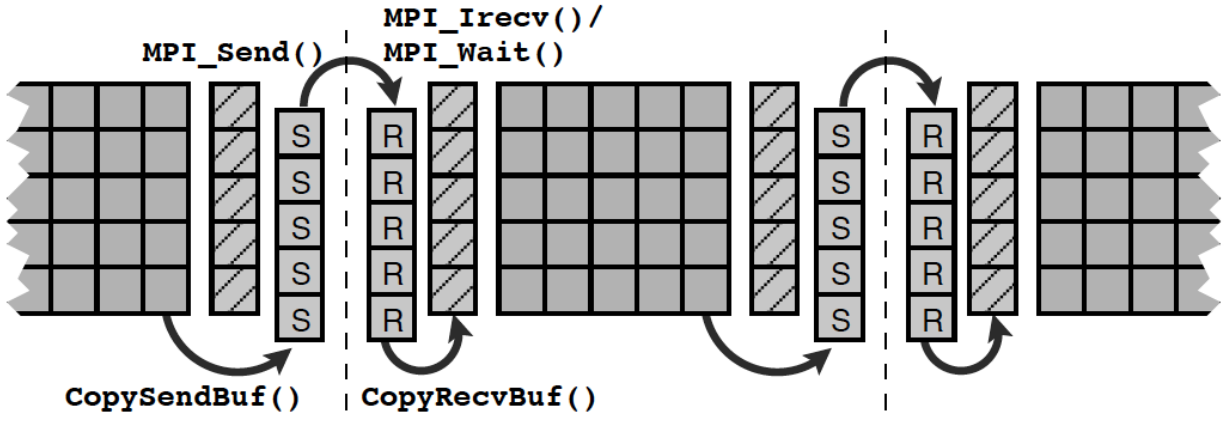


Figure 9.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled "R" ("S") belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

MPI's derived data types

- MPI's derived data types can help to simplify message packing/unpacking.
- The programmer can introduce new datatypes beyond the built-in ones, and use them in communication calls.
- There is a variety of choices for defining new types.
- The new type must first be defined using `MPI_Type_XXXXX()`, where "`XXXXX`" designates one of the variants.

MPI_Type_vector

```
int MPI_Type_vector(
  int count,
  int blocklength,
  int stride,
  MPI_Datatype old_type,
  MPI_Datatype *newtype_p
);
```

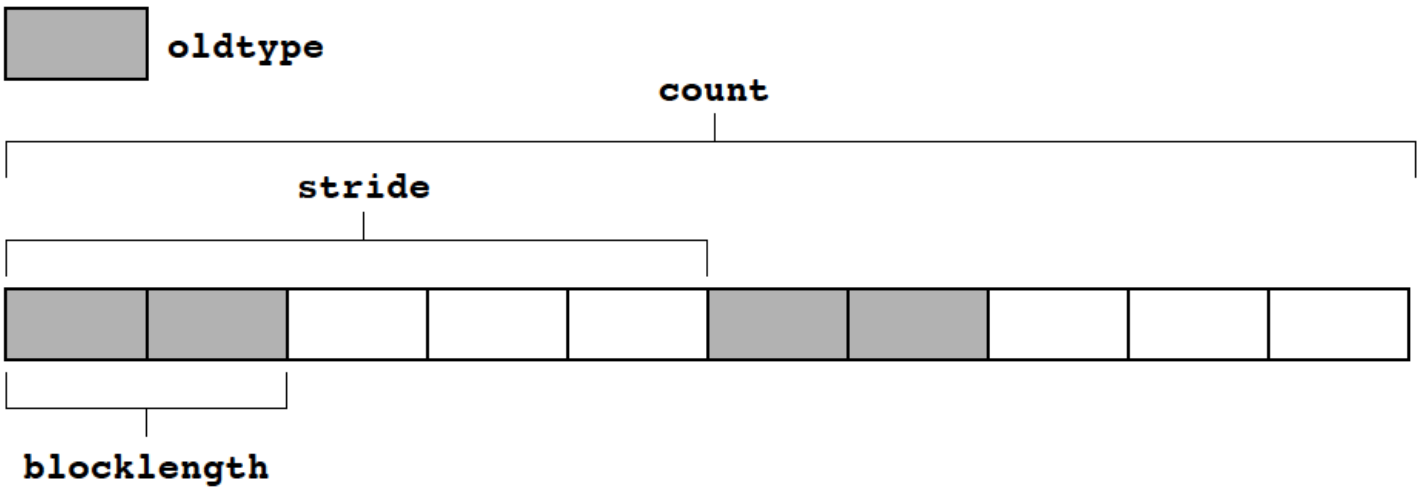


Figure 10.13: Required parameters for `MPI_Type_vector`. Here `blocklength`=2, `stride`=5, and `count`=2.

Example of using MPI_Type_vector

x-direction communication for 2D domain decomposition, each process with arrays of dimension YMAX$\times$XMAX (including the ghost layer)

```
MPI_Datatype new_type;
MPI_Type_vector(YMAX-2,1,XMAX,MPI_DOUBLE,&new_type);
MPI_Type_commit(&new_type);

....

MPI_Send(&(array[1][1]),1,new_type,left_dest,tag0,MPI_COMM_WORLD);
MPI_Send(&(array[1][XMAX-2]),1,new_type,right_dest,tag1,MPI_COMM_WORLD);

....

MPI_Type_free(&new_type);
```

Asynchronous communication

- A further chance for increasing efficiency of parallel programs is overlapping communication and computation.
- Nonblocking point-to-point communication is the straightforward way to achieve this.
- Example of 2D domain decomposition for Jacobi computation:

1. Compute only the rows `phi_new[1][...]`, `phi_new[YMAX-2][...]` and columns `phi_new[...][1]`, `phi_new[...][XMAX-2]`
2. Initiate sending/receiving of all the messages by `MPI_Isend` and `MPI_Irecv`
3. Compute the "interior" of the interior points (hopefully MPI messages are being transferred at the same time)
4. Finish all MPI calls by `MPI_Wait`

Difference between nonblocking and asynchornous communciation

Very important: One must strictly differentiate between non-blocking and truly asynchronous communication. Nonblocking semantics, according to the MPI standard, merely implies that the message buffer cannot be used after the call has returned from the MPI library; **In other words**, it is entirely up to the implementation whether data transfer, i.e., MPI progress, takes place while user code is being executed outside MPI.

Revisit of a naive implementation of numerical integration

Listing 9.3: Program fragment for parallel integration in MPI.

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   do i=1,size-1
      call MPI_Recv(tmp, &    ! receive buffer
                    1,   &    ! array length
                         ! data type
                    MPI_DOUBLE_PRECISION,&
                    i,   &    ! rank of source
                    0,   &    ! tag (unused here)
                    MPI_COMM_WORLD,& ! communicator
                    status,& ! status array (msg info)
                    ierror)
      res=res+tmp
   enddo
   write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum,    &    ! send buffer
                 1,       &    ! message length
                 MPI_DOUBLE_PRECISION,&
                 0,       &    ! rank of destination
                 0,       &    ! tag (unused here)
                 MPI_COMM_WORLD,ierror)
endif
```

Beware of single-process becoming the bottleneck

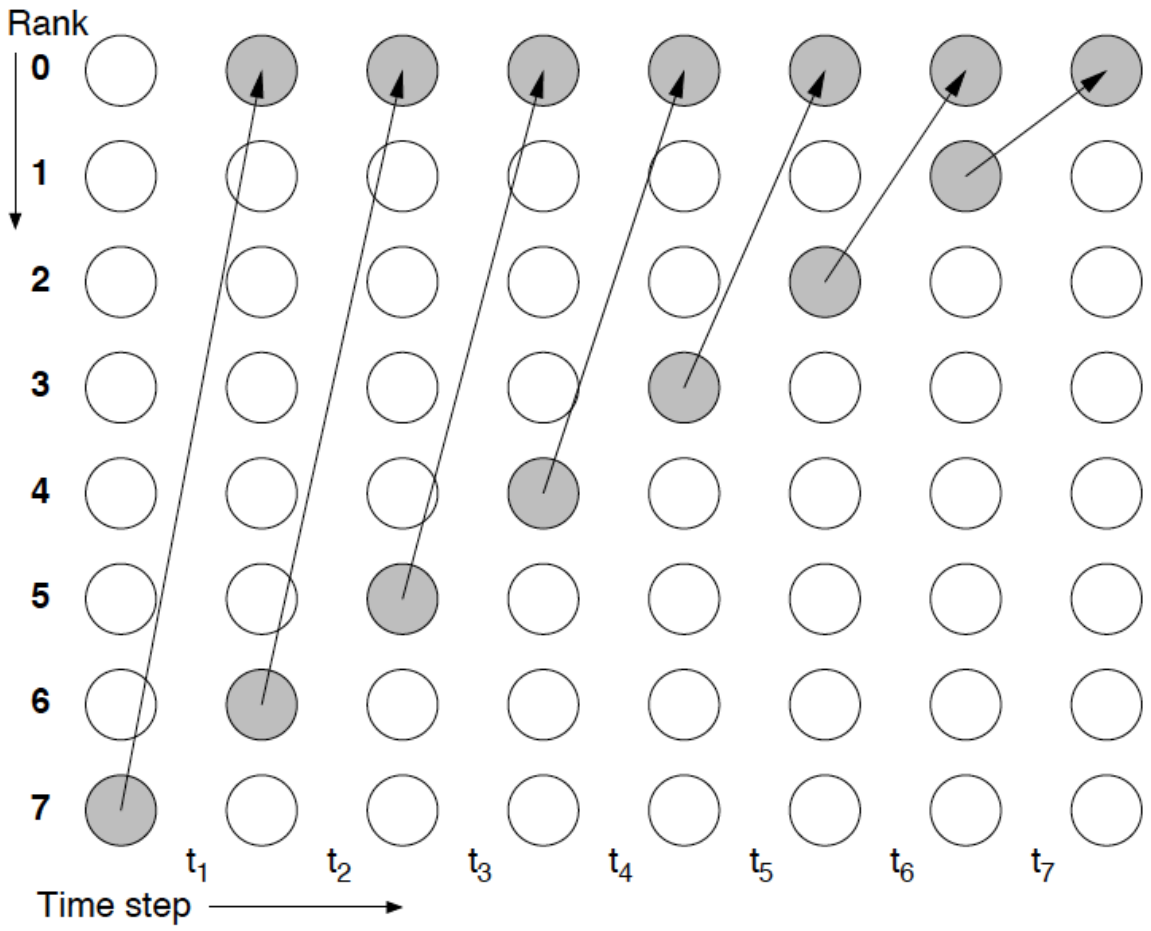


Figure 10.15: A global reduction with communication overhead being linear in the number of processes, as implemented in the integration example (Listing 9.3). Each arrow represents a message to the receiver process. Processes that communicate during a time step are shaded.

Collective communication bears optimization potential

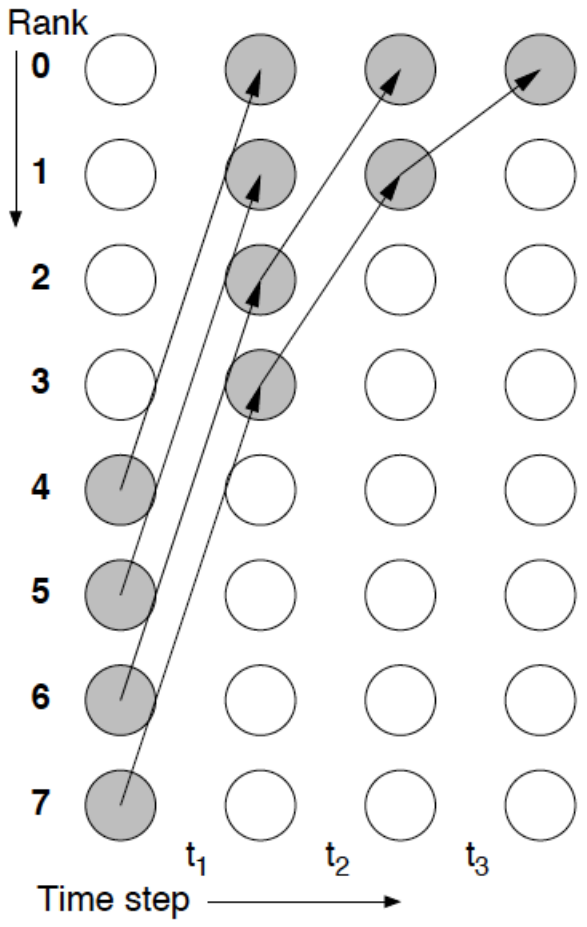


Figure 10.16: With a tree-like, hierarchical reduction pattern, communication overhead is logarithmic in the number of processes because communication during each time step is concurrent.