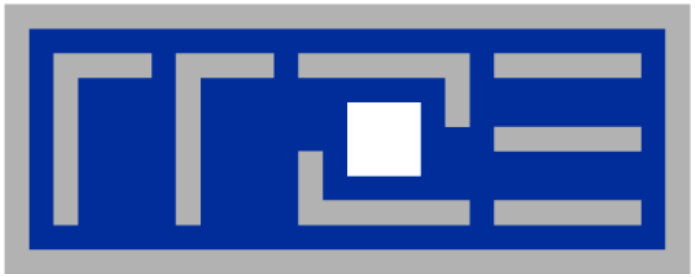


Programming Techniques for Supercomputers:

Distributed memory parallel processing with MPI
Jacobi 3D – a MPI case study

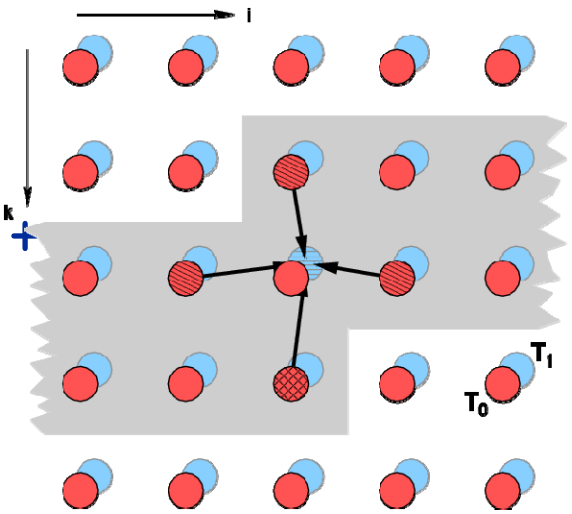
Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a] , J. Habich[a]

[a]HPC Services – Regionales Rechenzentrum Erlangen
[b]Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2010

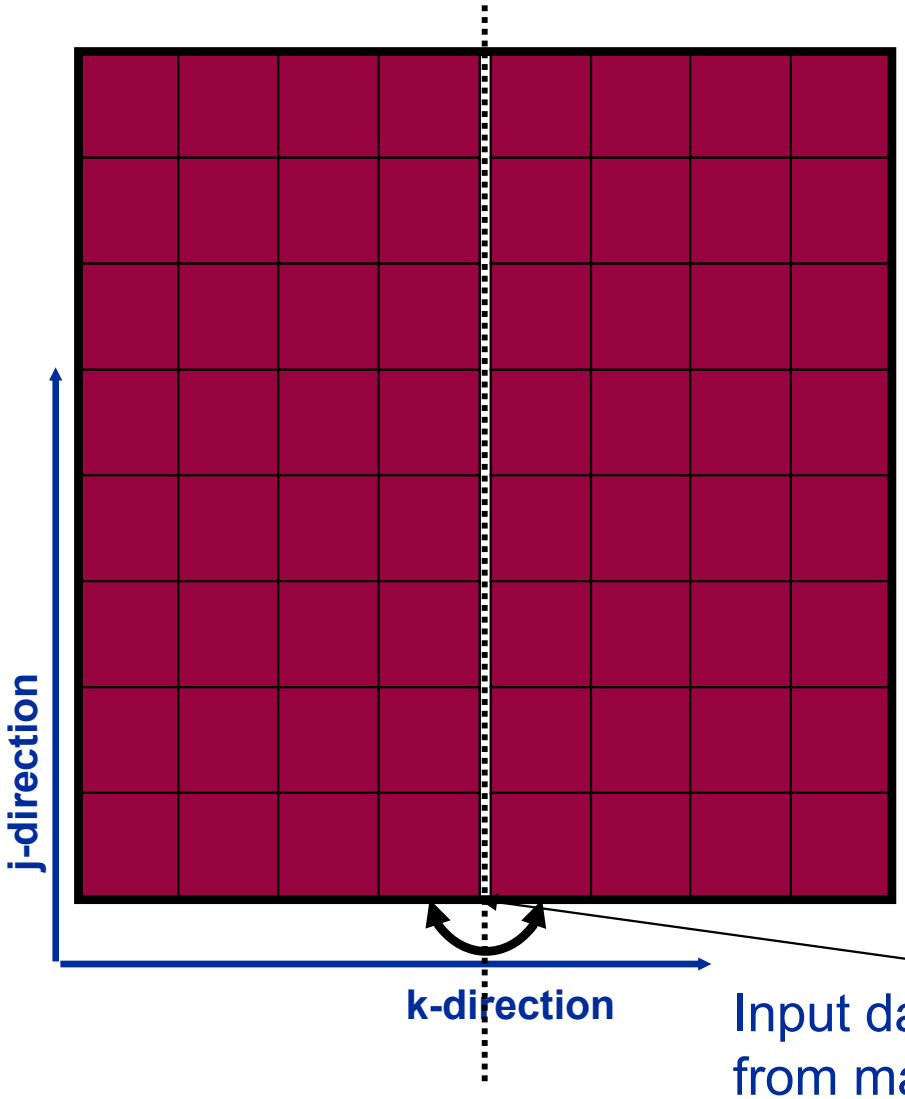
MPI parallelization
Jacobi solver – again a simple prototype

```
do k = 1 , Nk
  do j = 1 , Nj
       do i = 1 , Ni
          y(i,j,k) = b* (x(i-1,j,k)+x(i+1,j,k)+
                        x(i,j-1,k)+x(i,j+1,k)+
                        x(i,j,k-1)+x(i,j,k+1))
       enddo
  enddo
enddo
```



- Using two arrays:
  - Read input data from `x(:,:,:)` ; write new data to `y(:,:,:)`
  - Each entry of `y(:,:,:)` is updated only once in an outer iteration
  - → All updates can be done in parallel – provided that input data from neighboring cells are available
  - OpenMP/shared memory parallelization is straightforward; all threads can access complete input data in `x(:,:,:)`

MPI parallelization
Jacobi solver – OpenMP: implicit domain decomposition



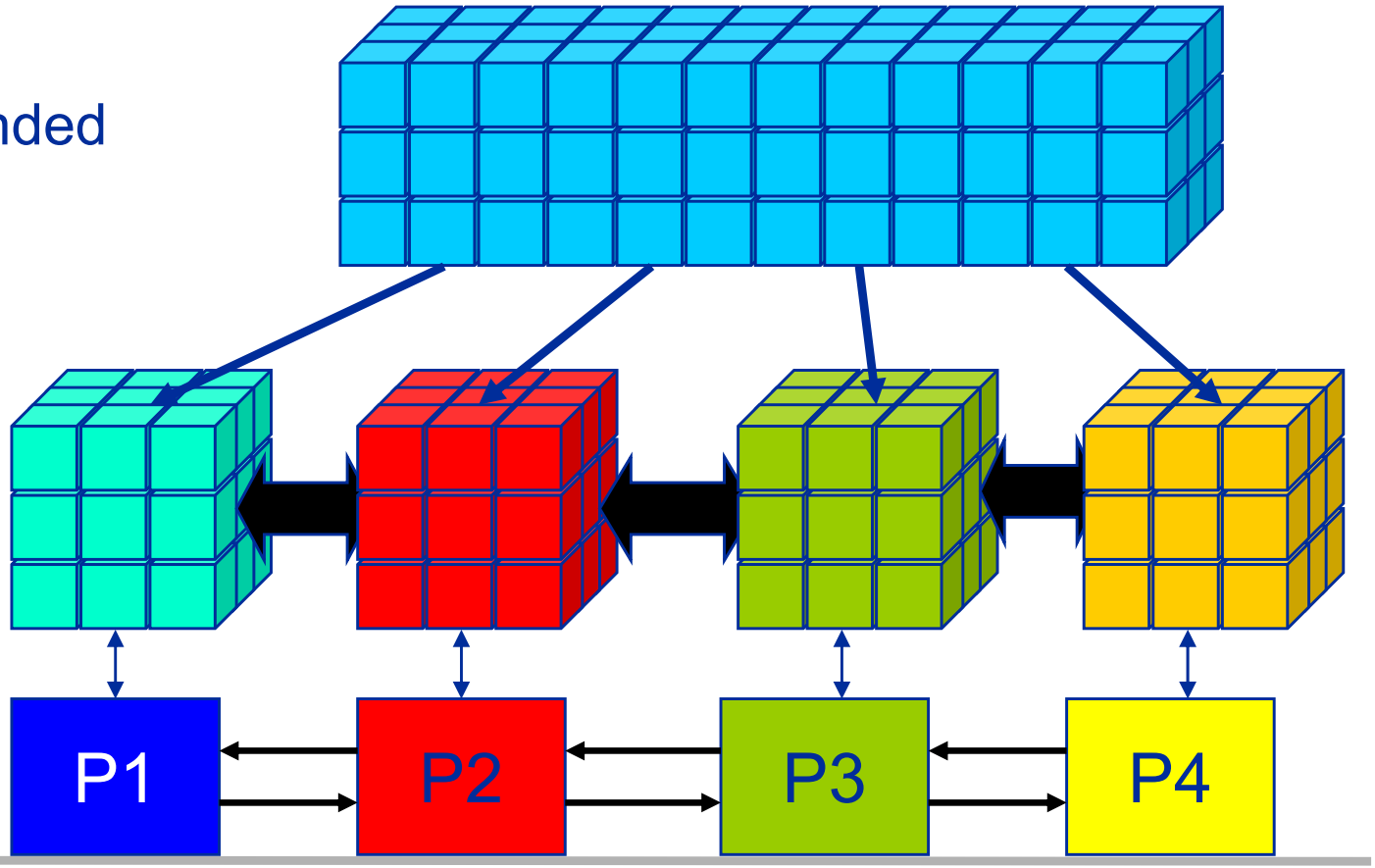

```
do t=1,tMax
!$OMP PARALLEL DO private(…)
   do k=1,Nk
      do j=1,Nj
         do i=1,Ni
            y(i,j,k) = …
         enddo
      enddo
   enddo (IMPLICIT BARRIER)
   y ←→ x
enddo
```

Input data at domain boundaries are read from main memory – no intervention from programmer required

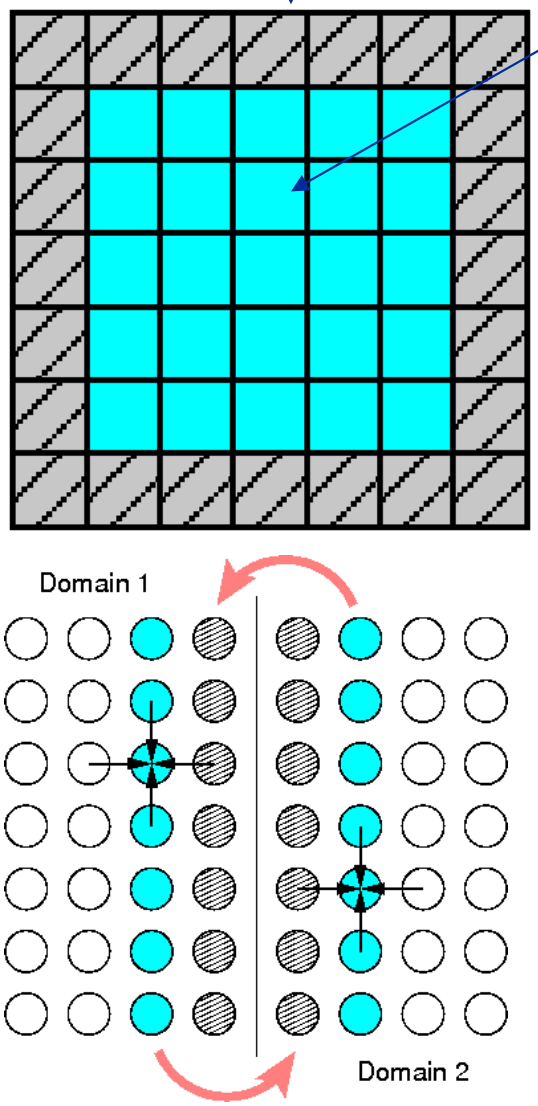
MPI parallelization
Jacobi solver – MPI: explicit domain decomposition

- Split up 12 x 3 x 3 domain into 4 equally sized disjoint subdomains
- Each subdomain can be updated by a single MPI process, provided that the boundary cells from adjacent subdomains are available
- Boundary cells need to be exchanged between adjacent subdomains (i.e. MPI processes) after each outer iteration

- Subdomains are extended by halo-layers which hold boundary data from adjacent sub-domains



- **Add halo layers to computational domain on each MPI process**

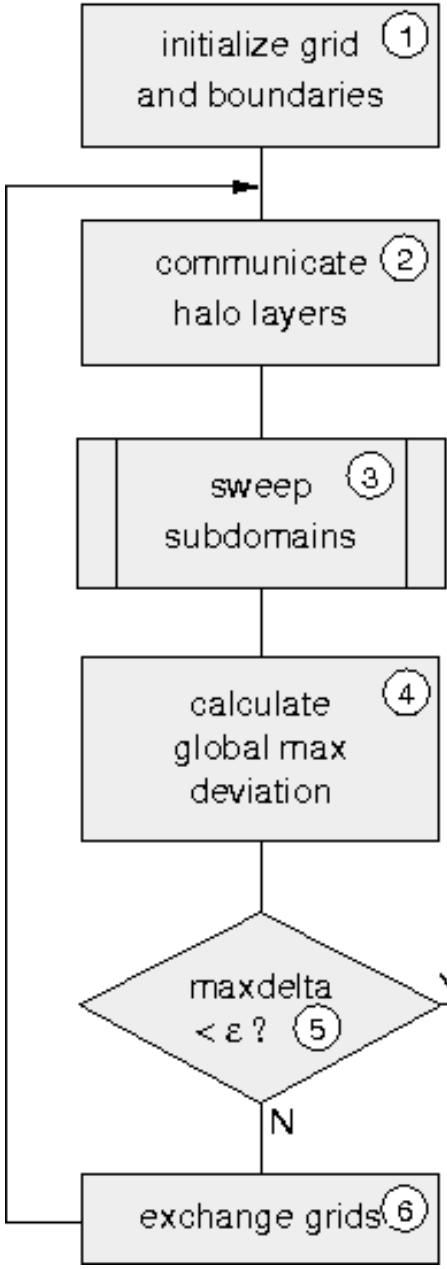


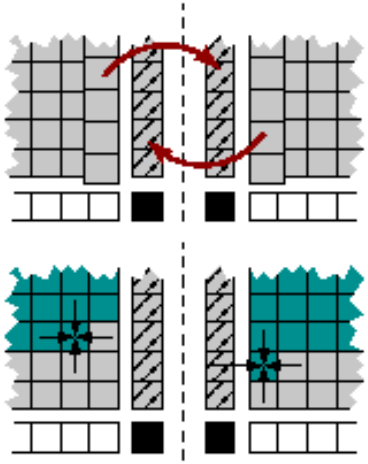
If N_{loc} is extension of cubic subdomain on each process then the overall cells on each process (including halo cells) is $(N_{loc}+2)^3$ for Jacobi3D

→

- local computation: N_{loc}^3
- communication overhead (in 3D): $6*N_{loc}^2$
- Overhead ~ $1 / N_{loc}$ (holds for 2D as well)
- But N_{loc} is limited by available memory!
- N_{loc} is typically smaller for 3D case
- 10^6 cells per process → $N_{loc}=10^2$ (3D) and 10^3 (2D)

MPI parallelization
Jacobi solver – MPI code flowchart





Update halo layers:
MPI point-to-point communication

Use optimized Jacobi3D sweep/update routine on each MPI process

```
MPI_Allreduce(MPI_IN_PLACE,
              maxdelta,...,
              MPI_MAX,...)
```

Determine global residual norm.
Here maximum norm

Check for global convergence

Swap grids: `y` ←→ `x`

MPI parallelization
Jacobi solver – Initialize (1)

- **3D domain with Ni x Nj x Nk cells running on `numprocs` MPI processes**

`pbc_check(d)` :periodicity in d-direction; if .true. than PBC are applied in d-direction

`spat_dim(d)` :#cells in d-direction (spat_dim(1)=Nk, spat_dim(2)=Nj, spat_dim(3)=Ni)

```
logical, dimension(1:3) :: pbc_check
integer, dimension(1:3) :: spat_dim, proc_dim
```

`proc_dim(d)` : User may specify specific domain decomposition or set 0

```
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)

if(myid.eq.0) then
  write(*,*) ' spat_dim , proc_dim, PBC ? '
  do i=1,3
    read(*,*) spat_dim(i), proc_dim(i), pbc_check(i)
  enddo
endif
```

Read input on master process and broadcast input

```
call MPI_Bcast(spat_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
call MPI_Bcast(proc_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
call MPI_Bcast(pbc_check, 3, MPI_LOGICAL, 0, MPI_COMM_WORLD, ierr)
```

- **Set up 3D Cartesian process topology**

Distribute **numprocs** in a 3D Cartesian topology, e.g. numprocs=16 → proc_dim={4,2,2}

```
call MPI_Dims_create(numprocs, 3, proc_dim, ierr)

if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
    (proc_dim(i),i=1,3)

(1) l_reorder = .true.
call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
    l_reorder, GRID_COMM_WORLD, ierr)
                                    (2)
(3) if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999

call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

Create a 3D process topology with appropriate boundary conditions.

Determine ranks & processes in new communicator

(1) Allow MPI reordering of process ids to match communication pattern

(2) Receive a new communicator for process topology → GRID_COMM_WORLD

(3) If total size of Cartesian process topology is smaller than numprocs → mapped out

MPI parallelization
Jacobi solver – Initialize (3)

- **After topology is set up: Set up process local subdomains and allocate memory accordingly**

```
integer, dimension(1:3) :: loca_dim, mycoord

call MPI_Cart_coords(GRID_COMM_WORLD, myid_grid, 3,
    mycoord,ierr)
do i=1,3
  loca_dim(i) = spat_dim(i)/proc_dim(i)
  if(mycoord(i) < mod(spat_dim(i),proc_dim(i))) then
    local_dim(i) = loca_dim(i)+1
  endif
enddo

iStart = 0 ; iEnd = loca_dim(3)+1
jStart = 0 ; jEnd = loca_dim(2)+1
kStart = 0 ; kEnd = loca_dim(1)+1

allocate(phi(iStart:iEnd, jStart:jEnd, kStart:kEnd,0:1))
```

Initialize computational domain and boundary layers(not shown)

Determine local process coordinates

Determine local subdomain extensions

Allocate local computational domain including boundary cells

MPI parallelization
Jacobi solver – Initialize (4)

- **Set up send and receive buffers for halo exchange through MPI**
  (tc: Current time step ε {0,1}; ldi ←→ local_dim(i))

```
integer, dimension(1:3) :: totmsgsize

! j-k plane    Exchange in "3-direction"
totmsgsize(3) = loca_dim(1)*loca_dim(2)   phi( 0    , 1:ld2 , 1:ld1 , tc) &
MaxBufLen=max(MaxBufLen,totmsgsize(3))    phi(iEnd, 1:ld2 , 1:ld1 , tc)
! i-k plane
totmsgsize(2) = loca_dim(1)*loca_dim(3)   phi(1:ld3    , 0    , 1:ld1 , tc) &
MaxBufLen=max(MaxBufLen,totmsgsize(2))    phi(1:ld3,  jEnd , 1:ld1 , tc)
! i-j plane
totmsgsize(1) = loca_dim(2)*loca_dim(3)   phi(1:ld3, 1:ld2 , 0      , tc) &
MaxBufLen=max(MaxBufLen,totmsgsize(1))    phi(1:ld3, 1:ld2 , kEnd , tc)

allocate(fieldSend(1:MaxBufLen))
allocate(fieldRecv(1:MaxBufLen))
```

Frequent drawback of combining halo cells and computational domain: Additional copy operation between halo cells and separate message buffer

MPI parallelization
Jacobi solver – parallel code

Determine neighbors in **dir** – direction

Receive from **source** and send to **dest**

Copy local boundary cells to send buffer

Exchange boundaries for a maximum of 6 surfaces → 6 Irecv/Send pairs; pair wise synchronization in each step

Copy receive buffer to halo cells

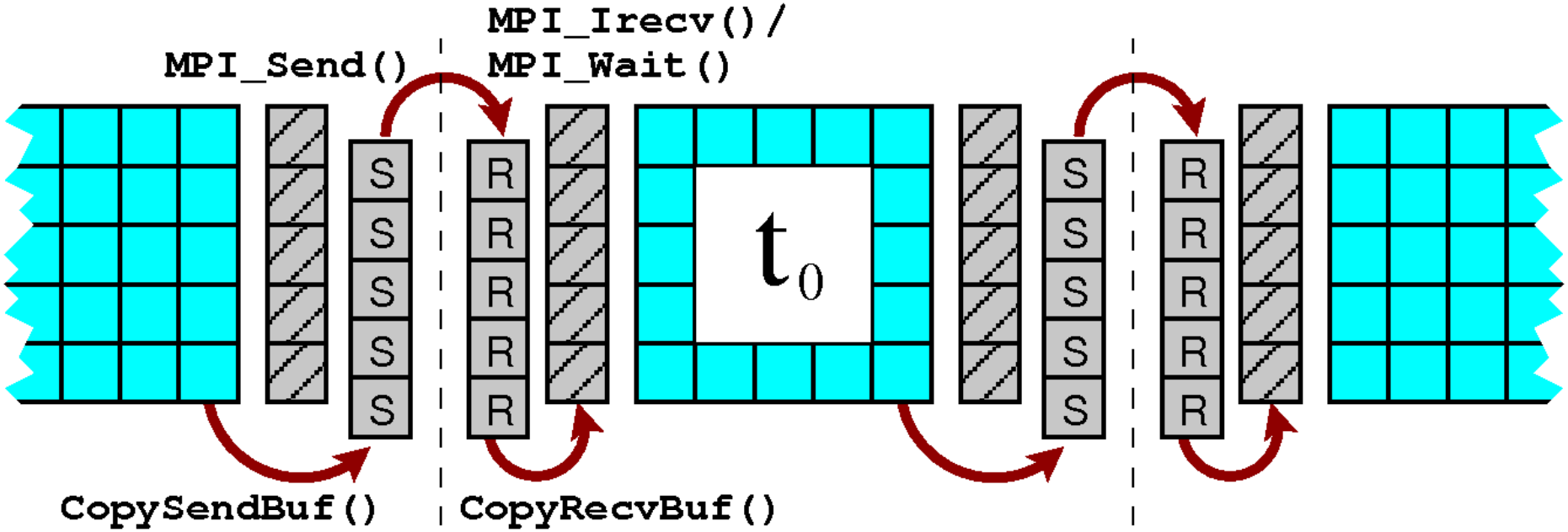
Do Jacobi iteration on each subgrid independently & determine process local residual (**maxdelta**)

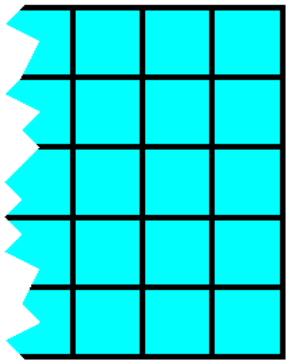
Determine global residual (**maxdelta**)

```
t0=0 ; t1=1
tag = 0
do iter = 1, ITERMAX                    Begin: outer iteration
  do disp = -1, 1, 2
    do dir = 1, 3

      call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
                          disp, source, dest, ierr)

      if(source /= MPI_PROC_NULL) then
        call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
             MPI_DOUBLE_PRECISION, source, &
             tag, GRID_COMM_WORLD, req(1), ierr)
      endif    ! source exists

      if(dest /= MPI_PROC_NULL) then
        call CopySendBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldSend, MaxBufLen)

        call MPI_Send(fieldSend(1), totmsgsize(dir), &
                      MPI_DOUBLE_PRECISION, dest, tag, &
                      GRID_COMM_WORLD, ierr)
      endif    ! destination exists

      if(source /= MPI_PROC_NULL) then
        call MPI_Wait(req, status, ierr)

        call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
                         iStart, iEnd, jStart, jEnd, kStart, kEnd, &
                         disp, dir, fieldRecv, MaxBufLen)
      endif    ! source exists

    enddo    ! dir
  enddo    ! disp

  call Jacobi_sweep(loca_dim(1), loca_dim(2), loca_dim(3), &
                    phi(iStart, jStart, kStart, 0), t0, t1, &
                    maxdelta)

  call MPI_Allreduce(MPI_IN_PLACE, maxdelta, 1, &
                     MPI_DOUBLE_PRECISION, &
                     MPI_MAX, 0, GRID_COMM_WORLD, ierr)
  if(maxdelta<eps) exit                 Check for convergence
  tmp=t0; t0=t1; t1=tmp
enddo    ! iter                         End: outer iteration

999 continue
```

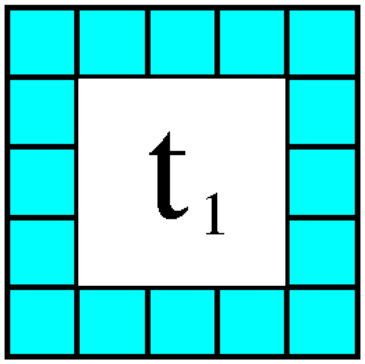
MPI parallelization
Jacobi solver – data exchange

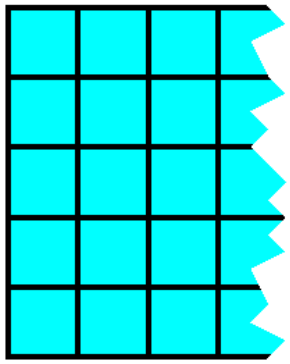
One boundary exchange step, e.g. `disp=+1, dir =3`





After complete boundary exchange, local subdomains can be update

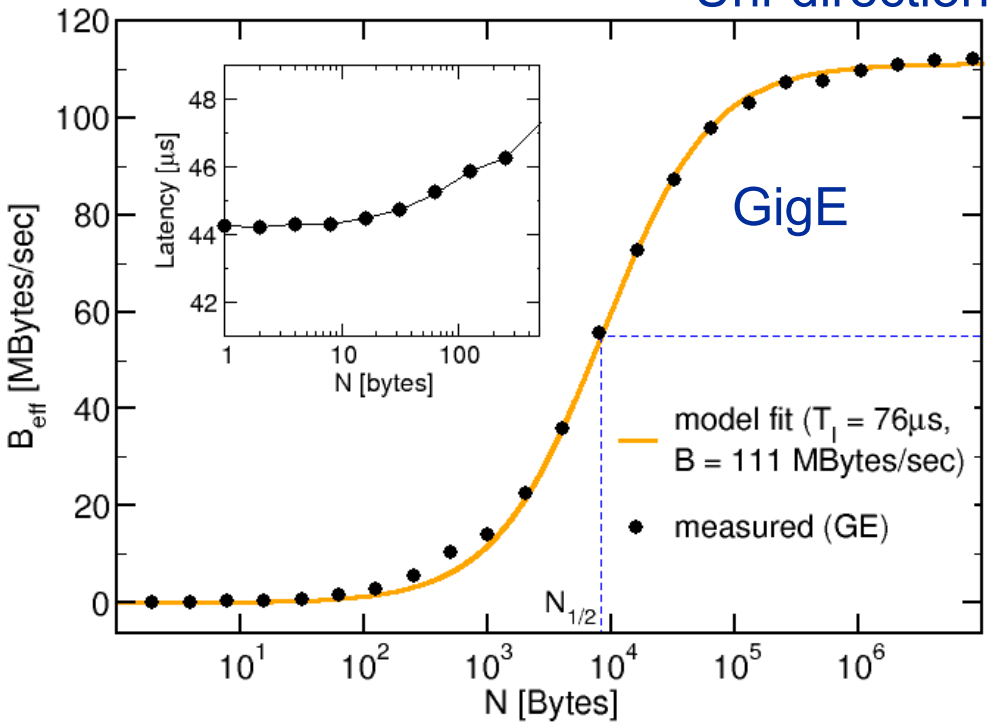


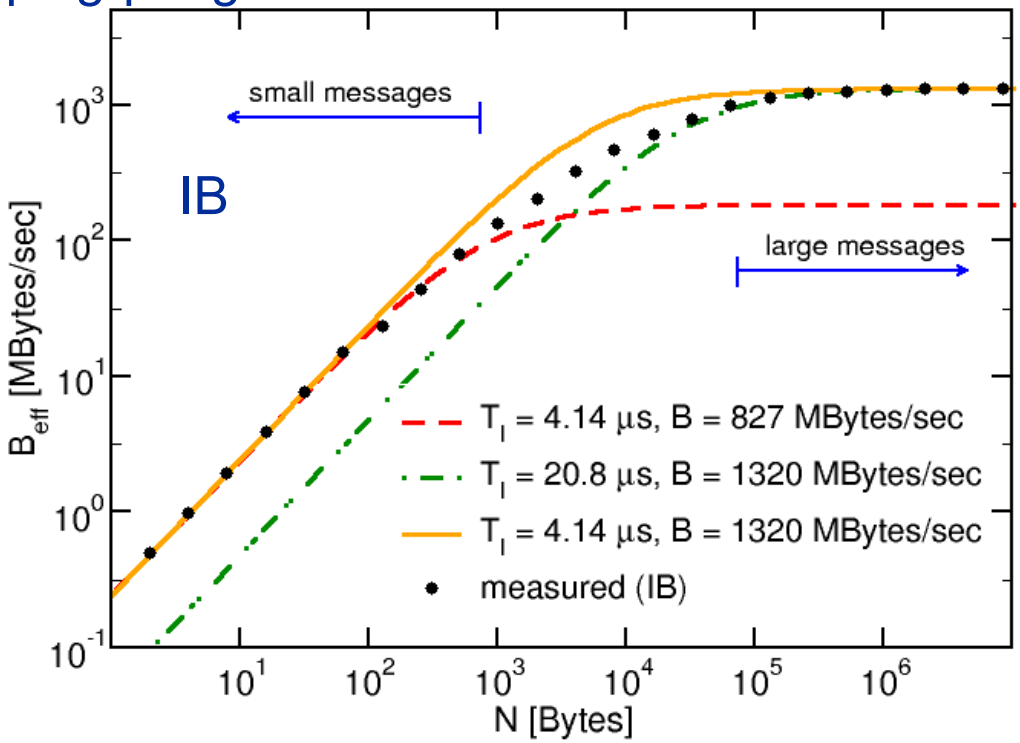


MPI parallelization
Jacobi solver – GBit/IB cluster

- **Run parallel code on single node cluster with**
  - GBit Ethernet (GigE) and DDR Infiniband (IB) interconnect
  - Each node is a dual-core Intel Xeon 3070 processor (Core2)
  - Run a single MPI process per node to test for network limitations
  - Single core – serial performance: 130 MLUPs/sec

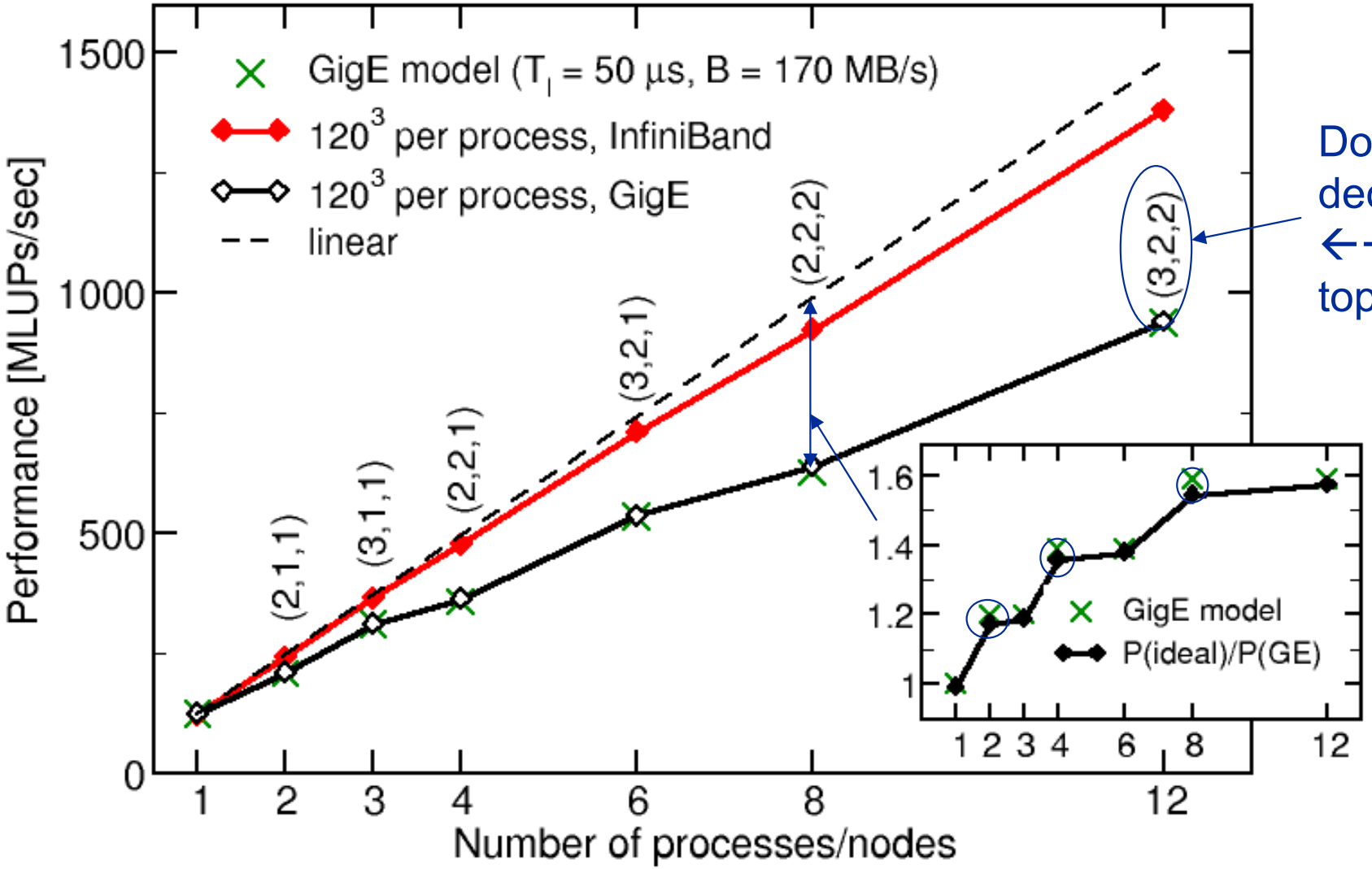
Uni-directional ping-pong bandwidth





MPI parallelization
Jacobi solver – weak scaling (measurement)

- **Domain per MPI process: 120^3**



Domain decomposition ←→ process topology

numprocs=2,4,8 → scalability gets worse?

- **Maximum data volume to be communicated (L=120; cubic subdomains!) per process**
  - L^2 cells * 8 Byte / cell * 2 * k =: c(L,**N**)

Send/receive in each direction — Max. number of neighbors for communication

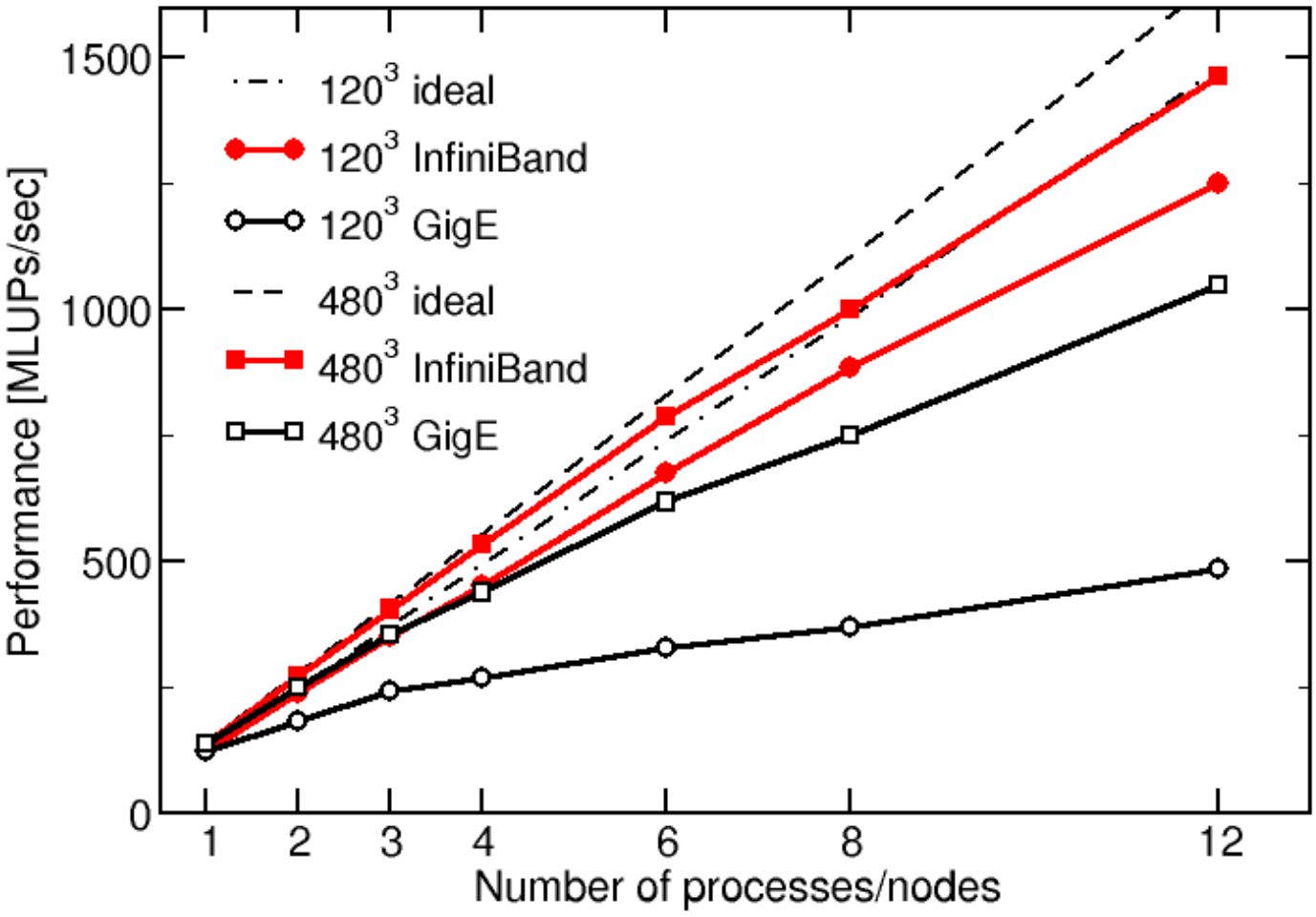
- **Max. time for communication:** $T_c(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_\ell$
  (B: Bi-directional bandwidth; T_l: latency)
- **$T_s(L)$ is the time required to update a subdomain of L^3 cells**
- **Performance model:**

$$P(L,\vec{N}) = \frac{L^3 N}{T_s(L) + T_c(L,\vec{N})}$$

B=170 MByte/s; T_l=50µs

N	(N_z,N_y,N_x)	k	$c(L,\vec{N})$ [MB]	$P(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1(L)}{P(L,\vec{N})}$
1	(1,1,1)	0	0.000	124	1.00
2	(2,1,1)	2	0.461	207	1.20
3	(3,1,1)	2	0.461	310	1.20
4	(2,2,1)	4	0.922	356	1.39
6	(3,2,1)	4	0.922	534	1.39
8	(2,2,2)	6	1.382	625	1.59
12	(3,2,2)	6	1.382	938	1.59

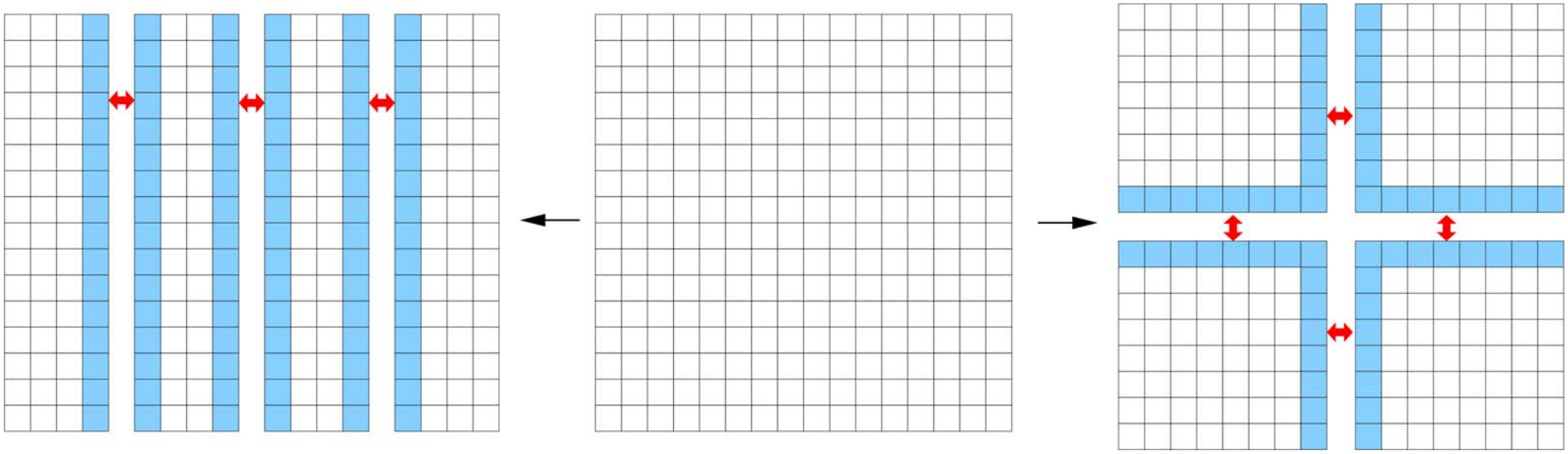
MPI parallelization
Jacobi solver – strong scaling



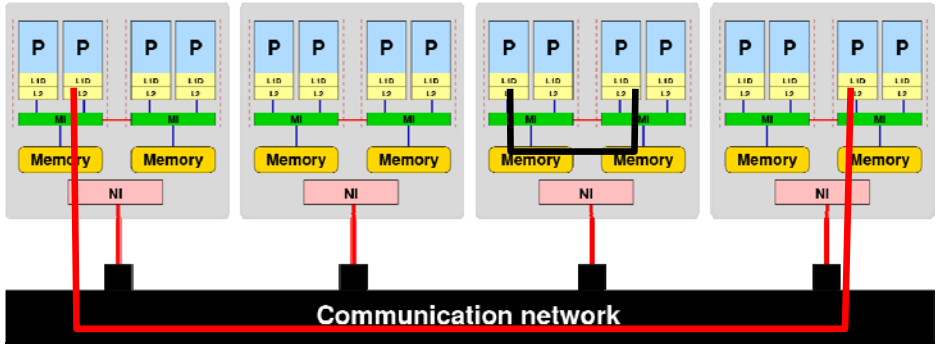
- Scalability is rather limited on GigE
- B → B(L(N)); cf. ping-pong measurements
- Single core performance depends on L
- Refined performance model required.
- Recover magic procesor numbers: 2,4,8

MPI parallelization
Choose appropriate domain decomposition

- **Choose a domain decomposition which ensures**
  1. Perfect load balancing ($O(L^3)$ effect)
  2. **Then** try to minimize communication overhead ($O(L^2)$ effect)

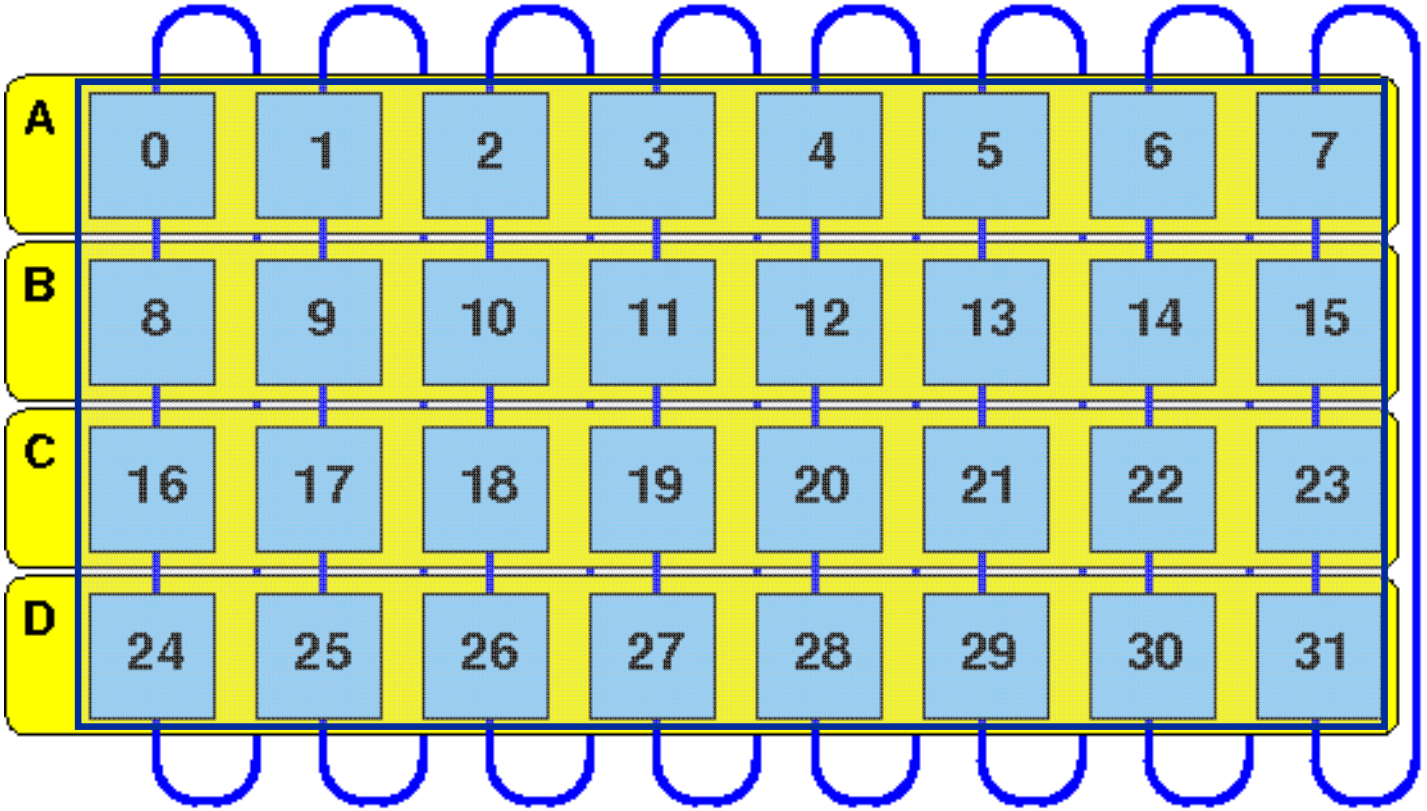


- **All modern systems implement a hierarchical architecture with at least two different interprocess data paths**
  - Communicating within a node is assumed to be (infinitely) **fast**
  - Communication between nodes through the network (**slow path**)



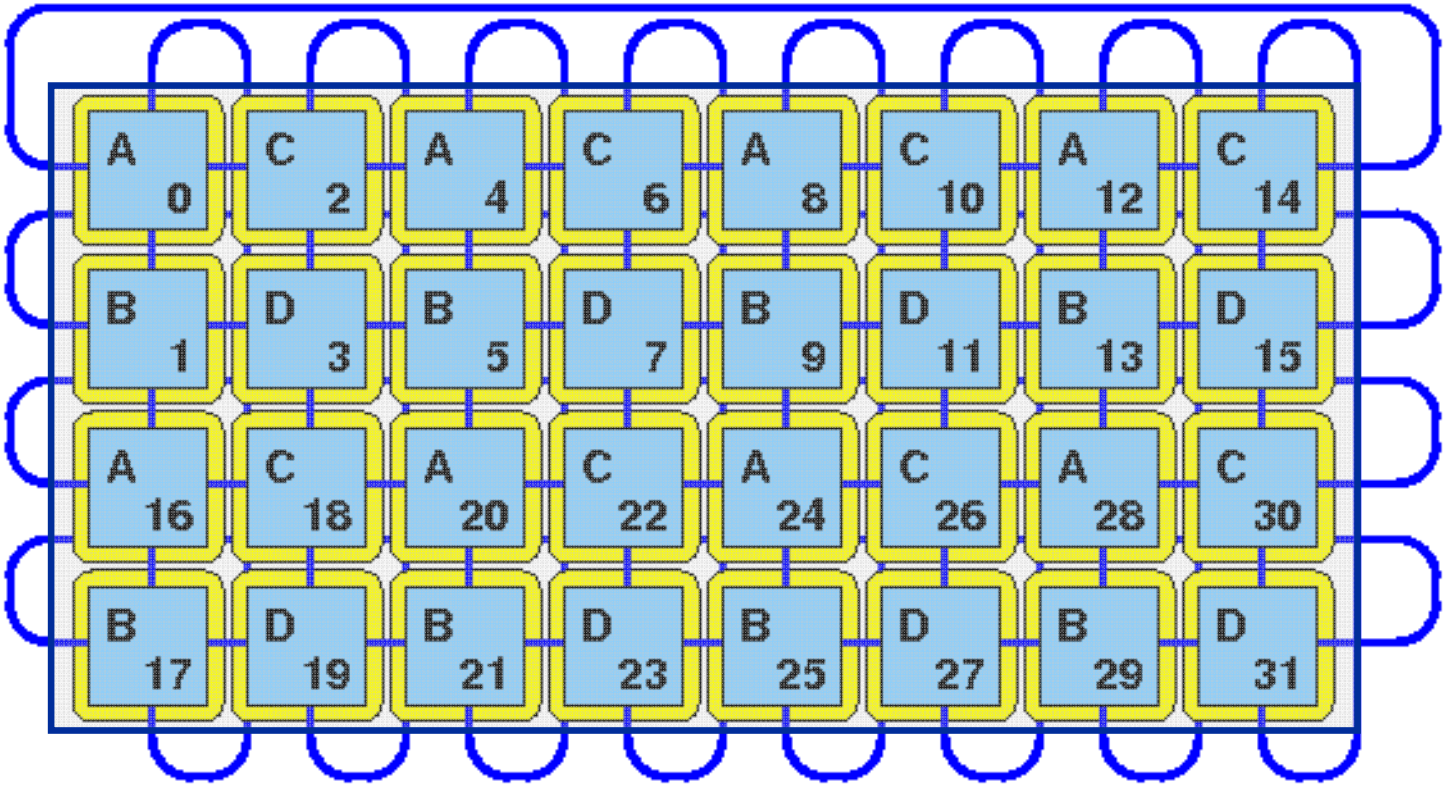
- **`MPI_Cart_create(…, reorder=.true.,…)` should know about topology and optimally reorder process topology but it does not!**
- **Consider a 4 x 8 Cartesian process grip with periodic boundary conditions (PBC)**
- **Nearest neighbor data exchange**
- **How to best distribute to 4 nodes (A,B,C,D) with 8 cores each?**
- **Consider internode communication only**

- Blockwise mapping of 32 MPI processes to the 4 nodes (A,B,C,D) (default for most `mpirun` implementations)



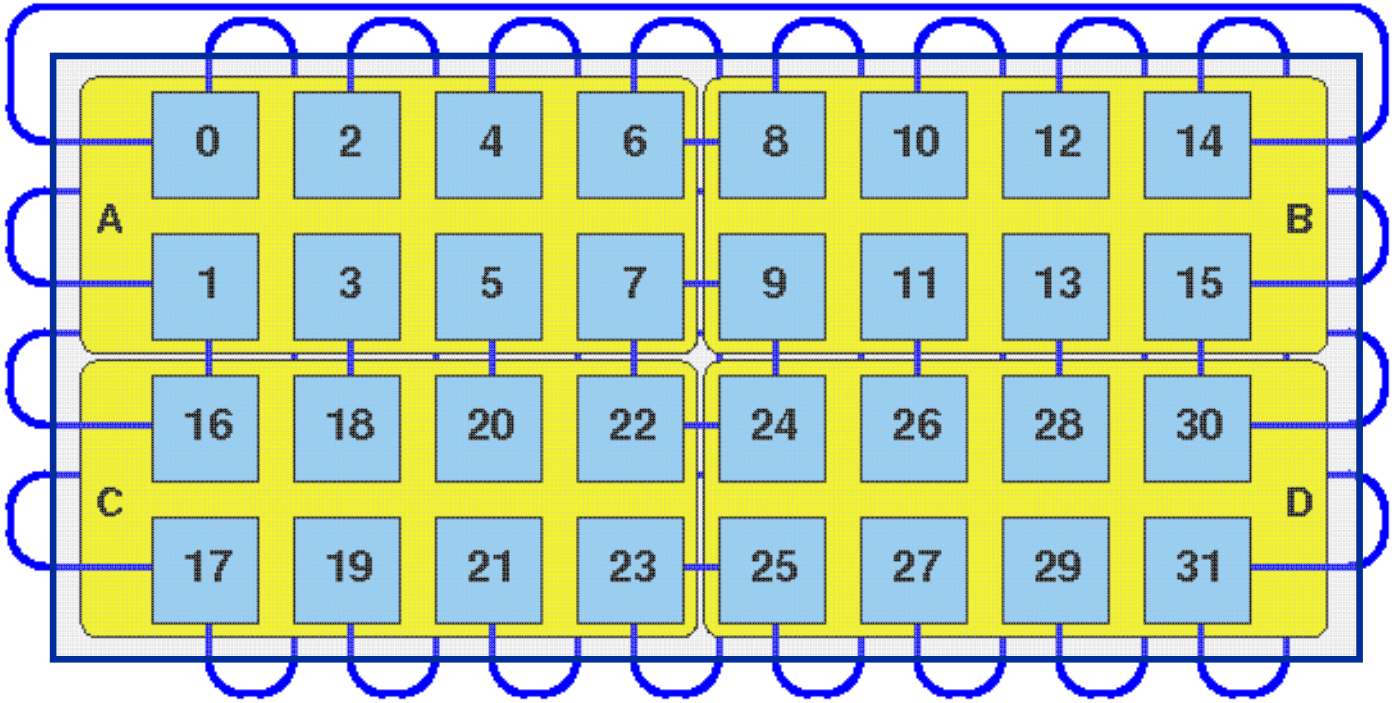
- 16 internode connections when using this process topology and mapping for domain decomposition

- Round-robin (cyclic) mapping of 32 MPI processes to 4 nodes (A,B,C,D) (`mpirun -rr` for most implementations)



- 32 internode connections when using this process topology and mapping for domain decomposition

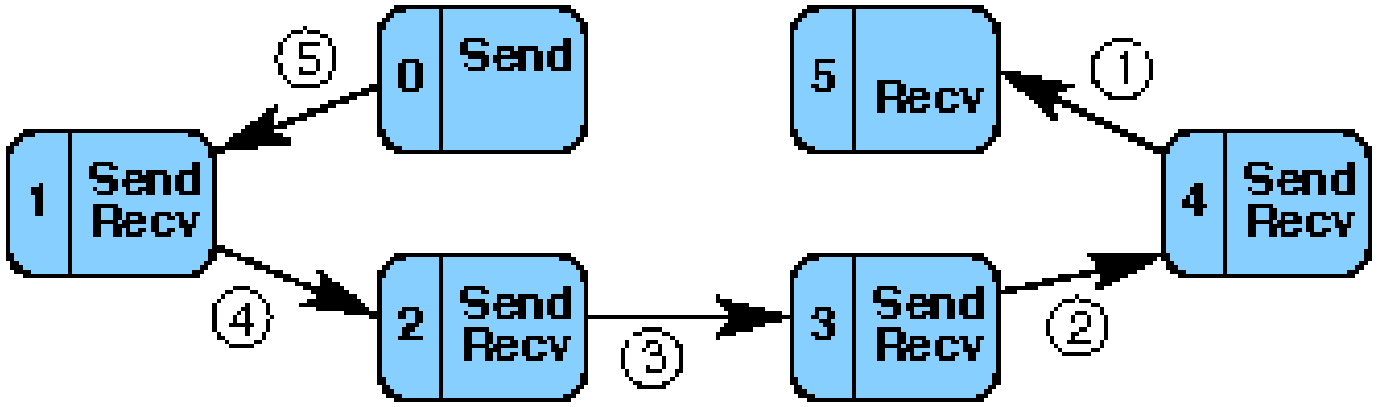
- Optimal mapping of 32 MPI processes to 4 nodes (A,B,C,D)



- 12 internode connections when using this process topology and mapping for domain decomposition
- `MPI_Cart_create(…, reorder=.true.,…)` should do it - does not!
- No standard way → Manual remapping of ranks: `rank` ←→ `newrank` (e.g. blockwise `{4,5,6,7}` ←→ `{8,9,10,11}`)

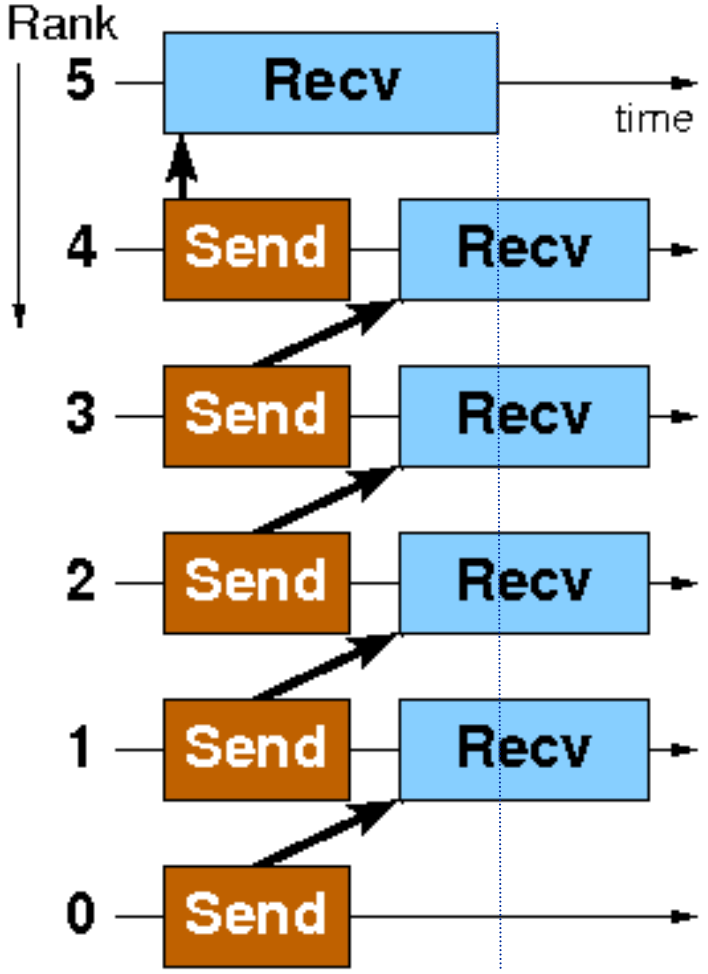
- Be aware of implicit serialization and synchronization
- Consider linear shift in an open chain, e.g. a process in the chain issues

```
call MPI Send(…,rank+1,…)
call MPI_Recv(…,rank-1,…)
```



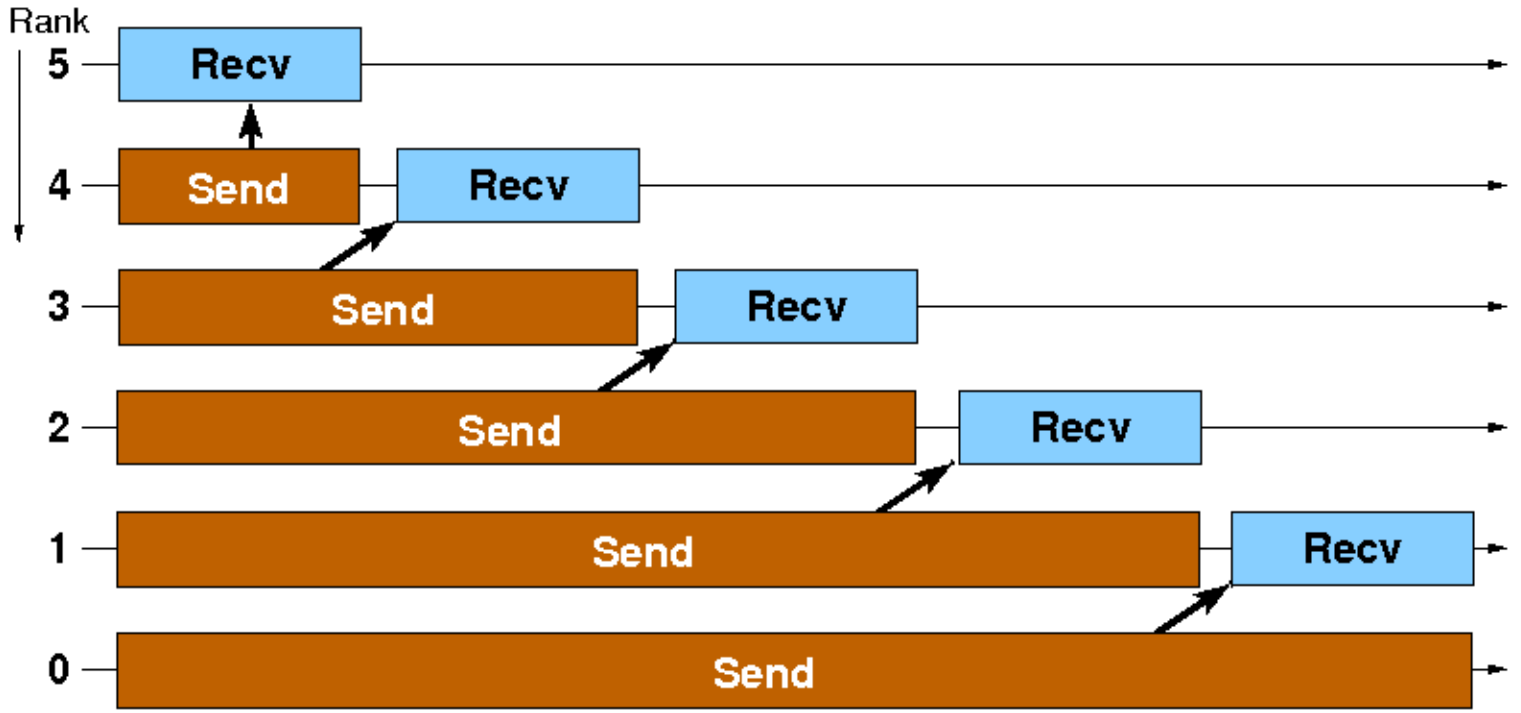
- There is no danger of deadlock but performance depend on many MPI-library specific parameters
  - Buffered or Synchronous MPI_Send
  - If synchronous MPI_Send: Eager or Rendezvous protocol?

- Best case scenario: MPI_Send is operating in a buffered send mode
- MPI_Send returns after message is copied to a system buffer
- Send/Receive operations can be overlapped on nonblocking, bidirectional networks

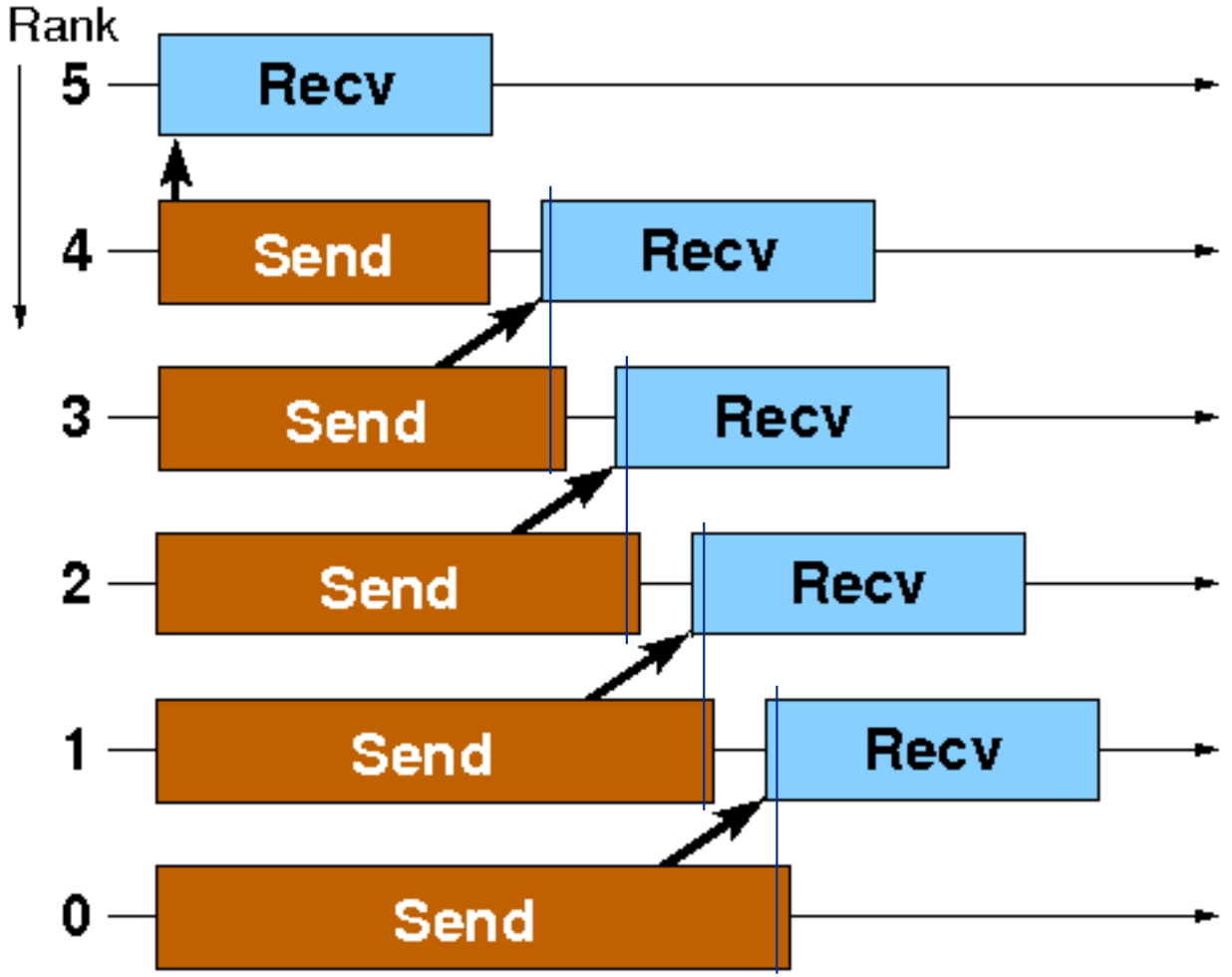


MPI parallelization
Pitfalls and best practices: Implicit serialization & sync (3)

- Worst case scenario: Synchronous send using the *rendezvous* protocol
- *Rendezvous:* Send operation blocks until complete message has been transferred!
- Serialization of all data transfers!

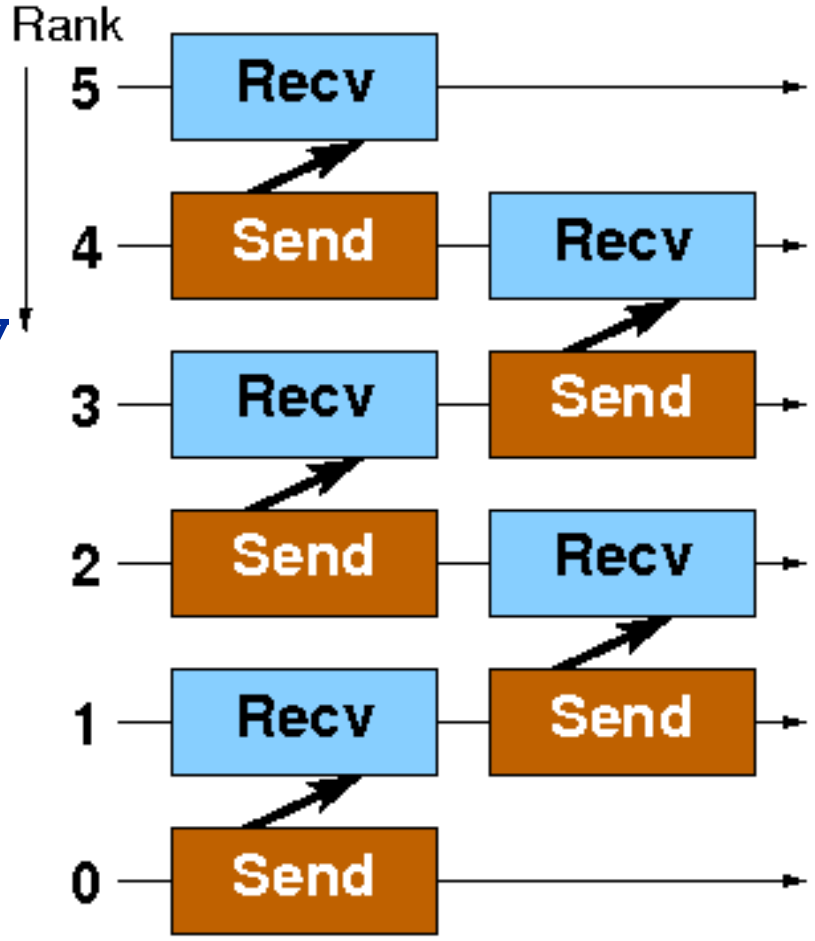


- Worst case scenario: Synchronous send using the *eager* protocol
- *Eager*: Message may be transmitted to receiver without a matching receive operation issued. Data is put in a local system buffer at receiver side. Depends on message length and availability of system buffer space.
- Handshake needs to be performed!



Better implementation alternatives

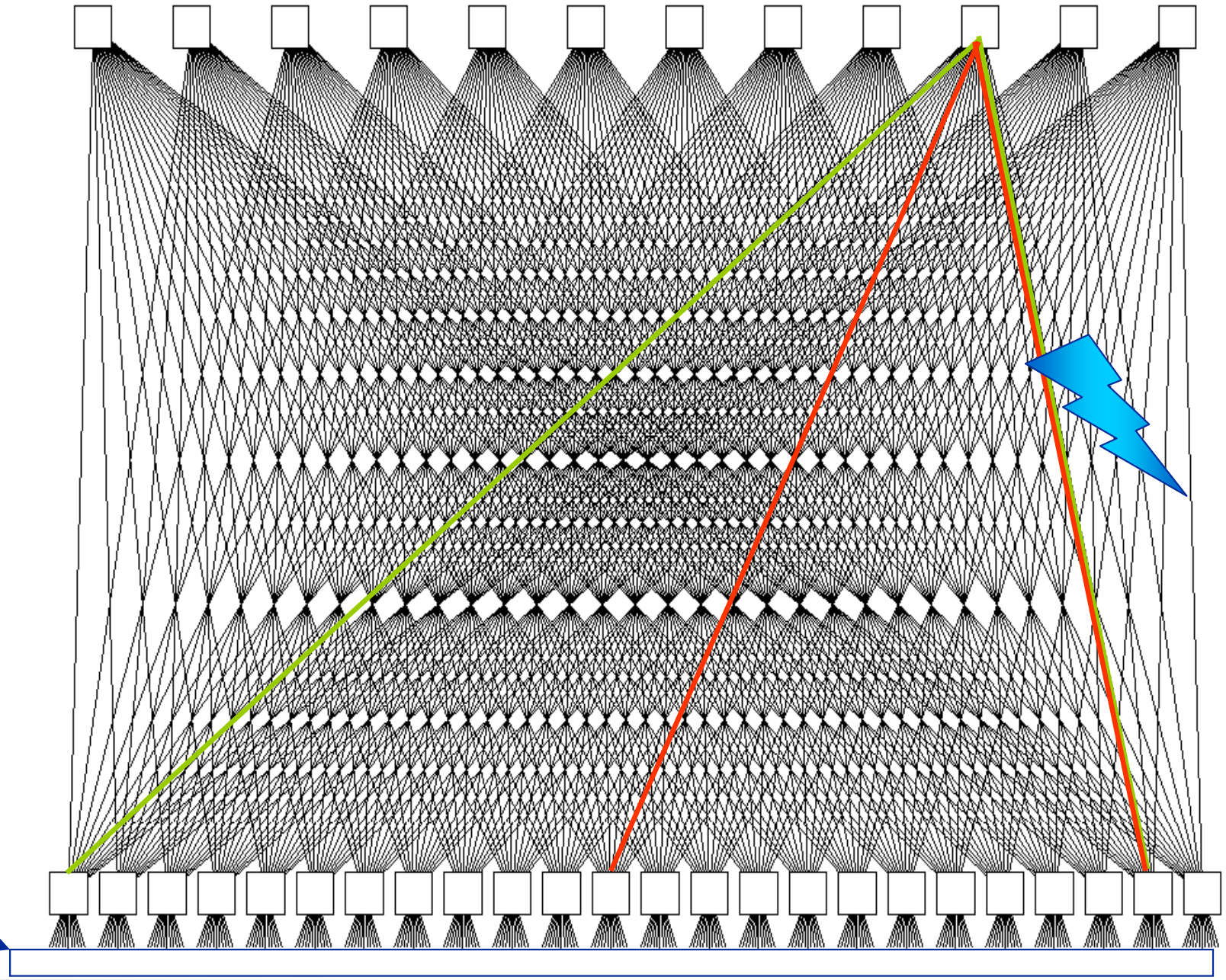
- Different order of send/receives calls on even and odd numbered processors.
- Use non-blocking `MPI_Isend/MPI_Irecv` pairs. Multiple outstanding/open communication requests allow for flexible optimal and scheduling. Provide also potential of asynchronous data transfer.
- Use `MPI_Sendrecv` or `MPI_Sendrecv_replace`: Simple coding with flexible message scheduling; but no asynchronous data transfer



- **Contention on network level may occur:**
  - Multiple processes on a node try to use the network interface (NI) at the same time. Network bandwidth per process decreases linearly if a single process can already achieve full network bandwidth
  - Network topology is not fully blocking, i.e. bisection bandwidth/compute node decreases with increasing compute nodes. Examples: Torus networks, (non-fat) tree networks
  - Even for full bisectional networks contention may occur on internal network links, due to non-optimal routing (cf. next slide).
- **Communication pattern most vulnerable for contention: `MPI_alltoall` → Every process wants to talk to every one else at the same time (imagine 300.000 processes do that)**

MPI parallelization
Pitfalls and best practices: Contention

Full fat tree built up from (24+12) switches with 24-ports each:



288 (=24*12) ports to compute nodes

MPI parallelization
Pitfalls and best practices: Asynchronous communication

- Overlap communication and computation: Effective communication costs can be reduced, if communication is done while the process is performing useful work ("asynchronous communication")
- Non-blocking MPI point-to-point communication provides a potential framework for implementing asynchronous communication:

```
MPI_Isend(a,…)
…do useful work and do not modify a… ! Data transfer !
MPI_WAIT()
```

- Of course one needs to identify "useful" work which can be done in between… → Jacobi solver?
- Perfect world: "useful" work takes more time than data transfer → communication is for free!
- However, MPI implementation gets more complex (e.g. by providing separate threads);
- MPI standard does not force the implementation of asynchronous data transfer in a fully compliant implementation ("MPI progress may also happen within MPI calls only")

- **Simple test if asynchronous data transfer is supported:**

Rank=0:

Receive non-blocking message of 80 MB

Do some work (“delay”)

WAIT for completion

Rank=1

Send 80 MB

```
double precision :: delay
integer :: count, req
count = 80000000
delay = 0.d0

do
  call MPI_Barrier(MPI_COMM_WORLD, ierr)
  if(rank.eq.0) then
    t = MPI_Wtime()
    call MPI_Irecv(buf, count, MPI_BYTE, 1, 0, &
                   MPI_COMM_WORLD, req, ierr)
    call do_work(delay)
    call MPI_Wait(req, status, ierr)
    t = MPI_Wtime() - t
  else
    call MPI_Send(buf, count, MPI_BYTE, 0, 0, &
                  MPI_COMM_WORLD, ierr)
  endif
  write(*,*) 'Overall: ',t,' Delay: ',delay
  delay = delay + 1.d-2
  if(delay.ge.2.d0) exit
enddo
```

MPI parallelization
Pitfalls and best practices: Asynchronous communication