Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 10:

More MPI – collective communication

Distributed-memory system architecture

Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Introduction to collectives in MPI

Collectives in MPI

Collectives: operations including all ranks of a communicator

All ranks must call the function!

- Blocking variants: buffer can be reused after return
- Nonblocking variants (since MPI 3.0):
  buffer can be used after completion (**`MPI_Wait*`**/**`MPI_Test*`**)

- May or may not synchronize the processes
- Cannot interfere with point-to-point communication
  - Completely separate modes of operation!

Collectives in MPI

- Rules for all collectives
  - Data type matching
  - No tags
  - Count must be exact, i.e., there is only one message length, buffer must be large enough
- Types:
  - Synchronization (barrier)
  - Data movement (broadcast, scatter, gather, all to all)
  - Collective computation (reduction, scan)
  - Combinations of data movement and computation (reduction + broadcast)
- General assumption: MPI does a better job at collectives than you trying to emulate them with point-to-point calls

Global communication

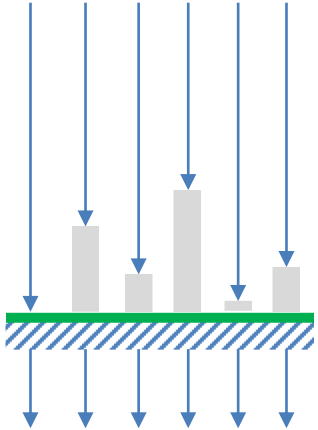
Barrier

- Explicit synchronization of all ranks from specified communicator

  ```
  MPI_Barrier(comm);
  ```

- Ranks only return from call after every rank has called the function

- `MPI_Barrier()` rarely needed
  - Debugging

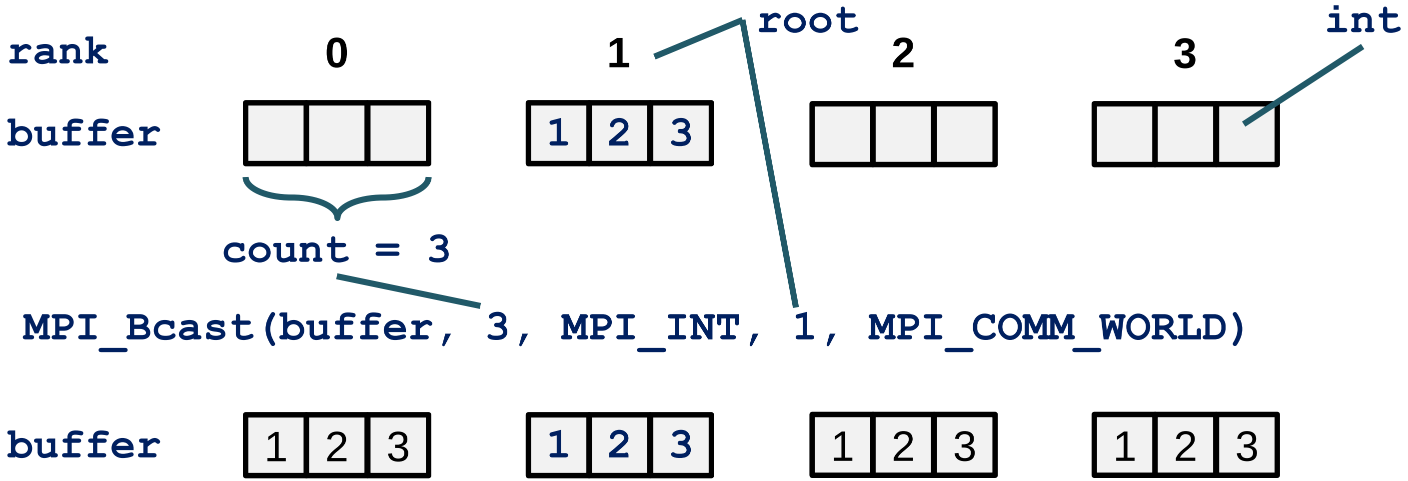


Broadcast

- Send buffer contents from one rank ("root") to all ranks

```
MPI_Bcast(buf, count, datatype, int root, comm);
```

- no restrictions on which rank is root – often rank 0



Scatter

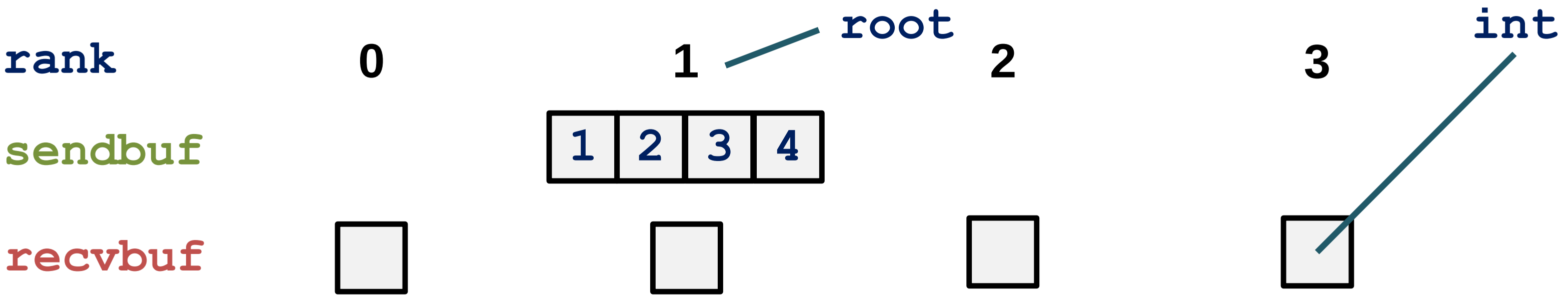
- Send the i-th chunk of an array to the i-th rank

```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm);
```

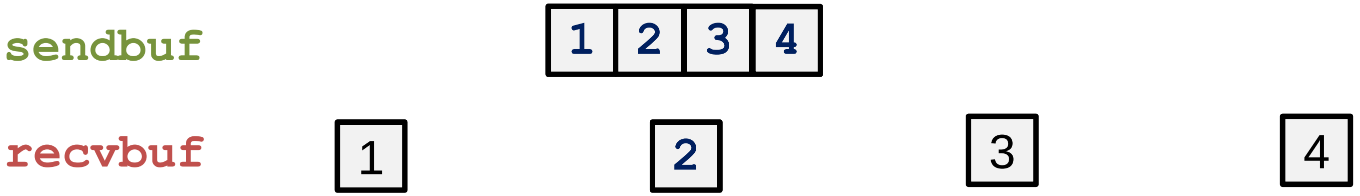
- In general, **`sendcount`** = **`recvcount`**
  - This is the length of the chunk

- **`sendbuf`** is ignored on non-root ranks because there is nothing to send

Scatter




```
MPI_Scatter(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT,
            root, MPI_COMM_WORLD)
```



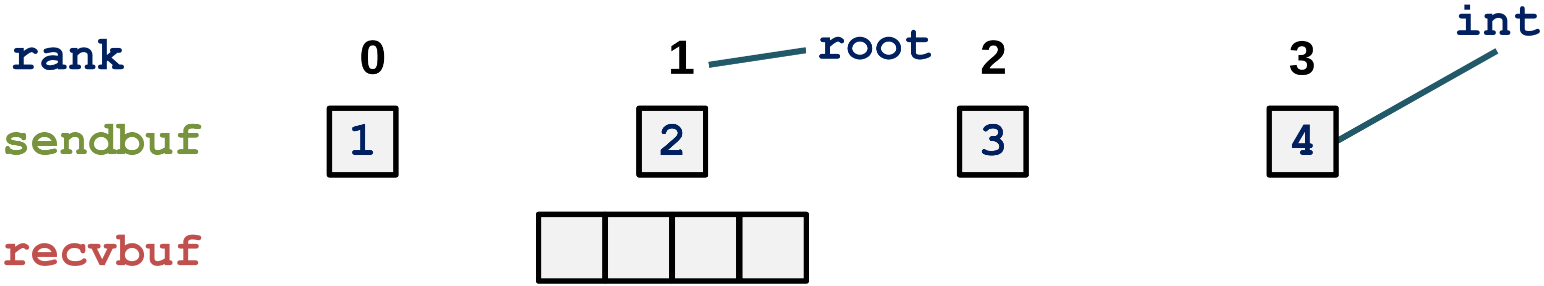
Gather

- Receive a message from each rank and place i-th rank’s message at i-th position in receive buffer

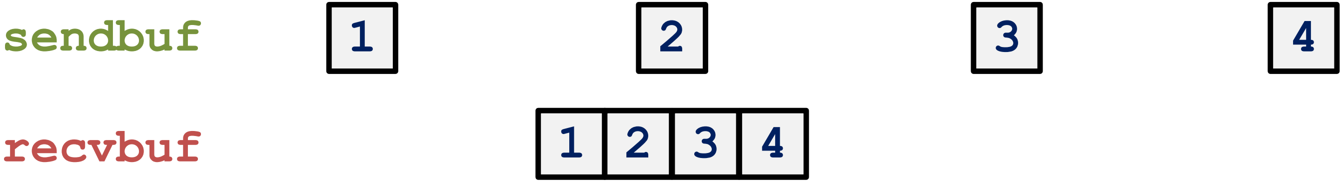
```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm)
```

- In general, **`sendcount`** = **`recvcount`**
- **`recvbuf`** is ignored on non-root ranks because there is nothing to receive

Gather




```
MPI_Gather(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT,
           root, MPI_COMM_WORLD)
```



Scatterv: more flexible scatter

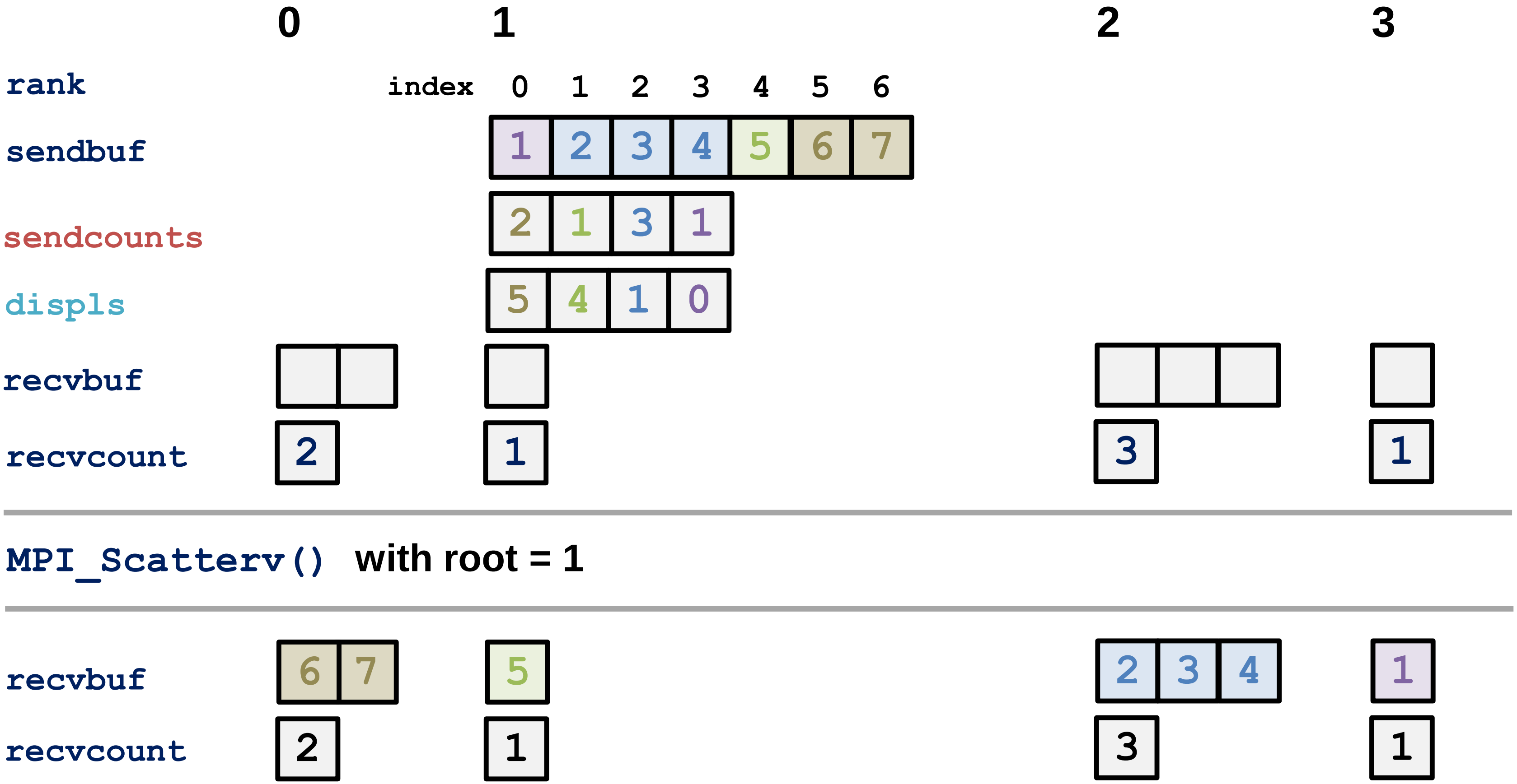
- Send chunks of different sizes to different ranks

```
MPI_Scatterv(
    sendbuf, int sendcounts[], int displs[], sendtype,
    recvbuf, recvcount, recvtype,
    root, comm);
```

`sendcounts[]`: array specifying the number of elements to send to each rank: send **`sendcounts[i]`** elements to rank **`i`**

`displs[]`: integer array specifying the displacements in **`sendbuf`** from which to take the outgoing data to each rank, specified in number of elements

Scatterv


0
1
2
3
rank
index 0 1 2 3 4 5 6
sendbuf
1 2 3 4 5 6 7
sendcounts
2 1 3 1
displs
5 4 1 0
recvbuf
recvcount
2
1
3
1
MPI_Scatterv() with root = 1
recvbuf
6 7
5
2 3 4
1
recvcount
2
1
3
1

Gatherv: more flexible gather

- Receive segments of different sizes from different ranks

```
MPI_Gatherv(
    sendbuf, sendcount, sendtype,
    recvbuf, int recvcounts[], int displs[], recvtype,
    root, comm)
```

`recvcounts[]`: array specifying the number of elements to receive from each rank: receive **`recvcounts[i]`** elements from rank **`i`**

`displs[]`: integer array specifying the displacements where received data from specific rank is put in **`recvbuf`**, in units of elements:

Allgather

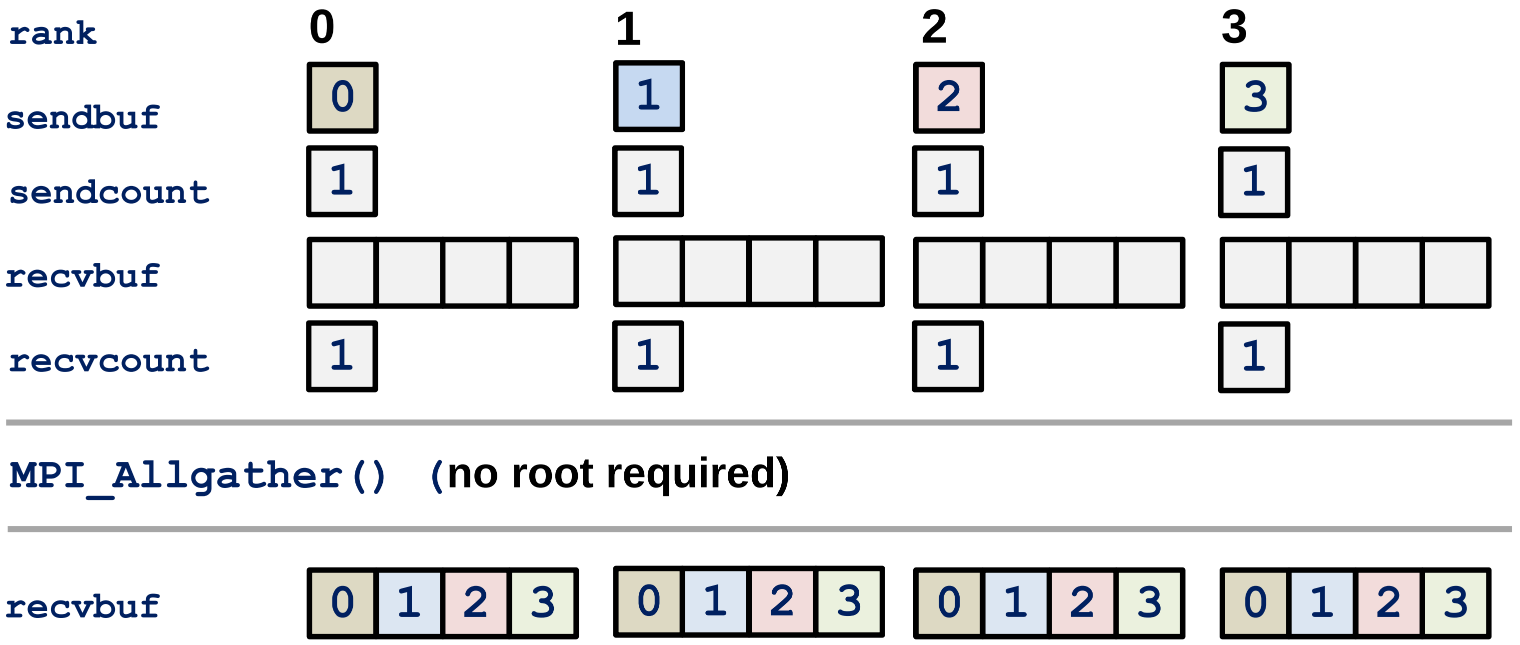
- Combination of gather and broadcast

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype,
              comm);
```

- Also available: `MPI_Allgatherv()` (cf. `MPI_Gatherv()`)

- Why not just use gather followed by a broadcast instead?
  - MPI library has more options for optimization
  - General assumption: Combined collectives are faster than using separate ones

Allgather


rank
0
1
2
3
sendbuf
0
1
2
3
sendcount
1
1
1
1
recvbuf
recvcount
1
1
1
1
MPI_Allgather() (no root required)
recvbuf
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Alltoall

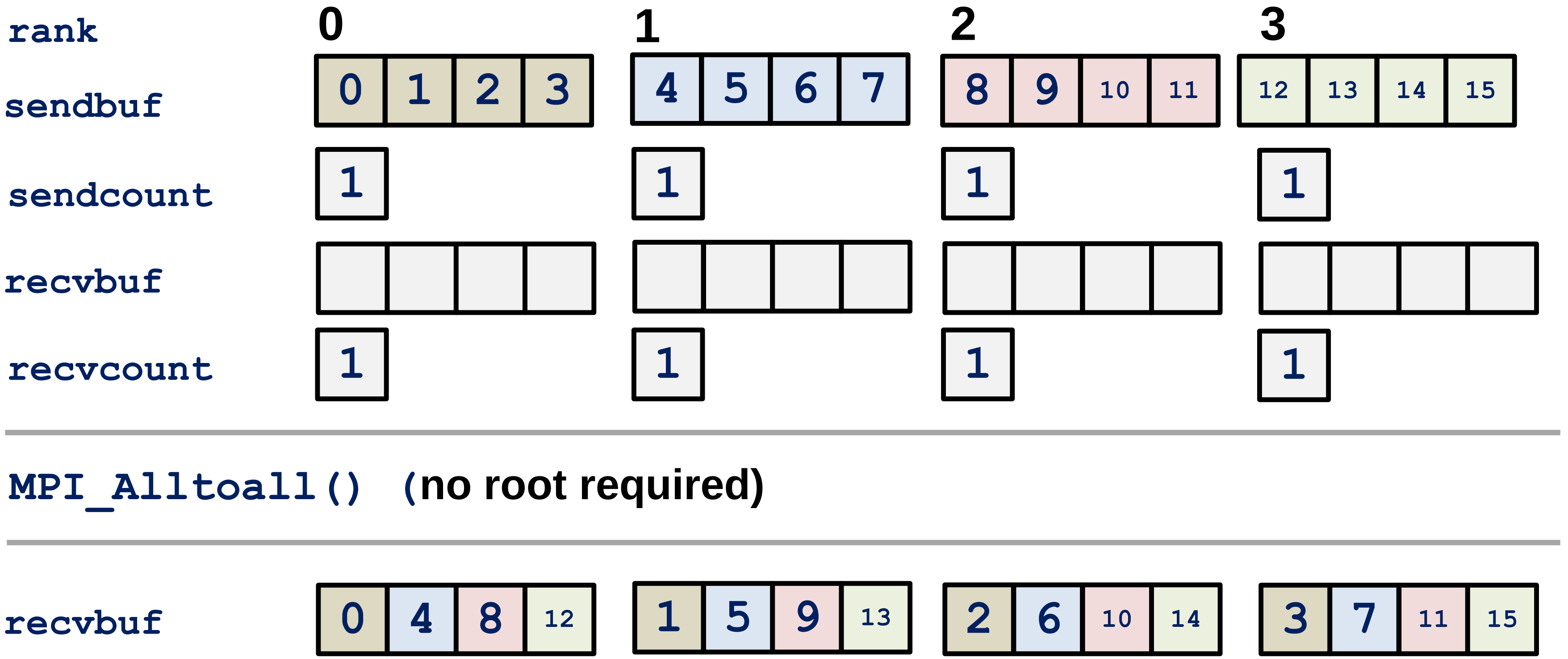
- `MPI_Alltoall`: For all ranks, send i-th chunk to i-th rank

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype,
             comm);
```

- `MPI_Alltoallv`: Allows different number of elements to be send/received by each rank

- `MPI_Alltoallw`: Allows also different data types and displacements in bytes

Alltoall


rank
0
1
2
3
sendbuf
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
sendcount
1 1 1 1
recvbuf
recvcount
1 1 1 1
MPI_Alltoall() (no root required)
recvbuf
0 4 8 12
1 5 9 13
2 6 10 14
3 7 11 15

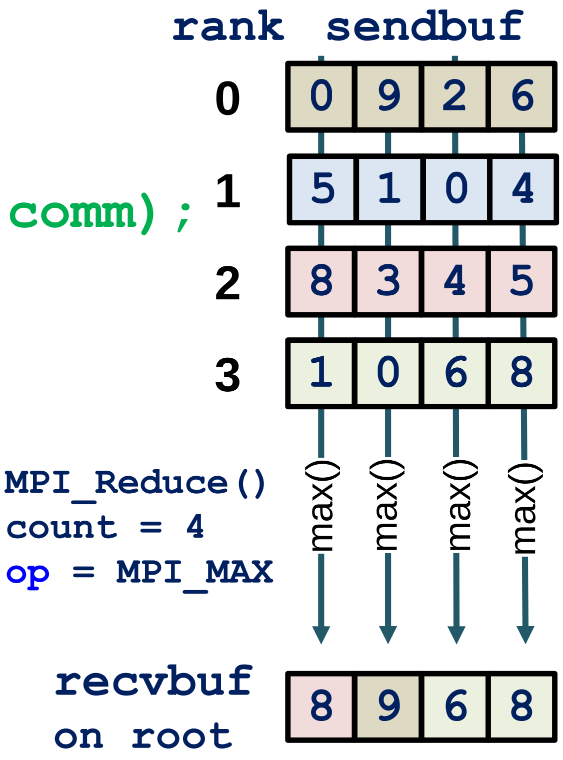
Global operations

Global operations: reduction

- Compute results over distributed data

```
MPI_Reduce(sendbuf, recvbuf, count,
           datatype, MPI_Op op, root, comm);
```

- Result in `recvbuf` only available on root process
- Perform operation on all `count` elements of an array
- If all ranks require result, use `MPI_Allreduce()`
- If the 12 predefined ops are not enough use `MPI_Op_create`/`MPI_Op_free` to create own ones



Global operations – predefined operators

Name	Operation	Name	Operation
MPI_SUM	Sum	MPI_PROD	Product
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_LAND	Logical AND	MPI_BAND	Bit-AND
MPI_LOR	Logical OR	MPI_BOR	Bit-OR
MPI_LXOR	Logical XOR	MPI_BXOR	Bit-XOR
MPI_MAXLOC	Maximum+Position	MPI_MINLOC	Minimum+Position

- Define own operations with `MPI_Op_create/MPI_Op_free`
- MPI assumes that the operations are associative
  → be careful with floating-point operations

“In-place” buffer specification

Override local input buffer with a result

MPI_Reduce

```
int partial_sum = …, total_sum;
MPI_Reduce(&partial_sum, &total_sum,
           1, MPI_INT,
           MPI_SUM, root, comm);
```

```
int partial_sum = …, total_sum;
if (rank == root) {
  total_sum = partial_sum;
  MPI_Reduce(MPI_IN_PLACE, &total_sum,
             1, MPI_INT,
             MPI_SUM, root, comm);
}
else {
  MPI_Reduce(&partial_sum, &total_sum,
             1, MPI_INT,
             MPI_SUM, root, comm);
}
```

MPI_Allreduce

```
int partial_sum = …, total_sum;
MPI_AllReduce(&partial_sum, &total_sum,
              1, MPI_INT,
              MPI_SUM, comm);
```

```
int partial_sum = …, total_sum;

total_sum = partial_sum;
MPI_AllReduce(MPI_IN_PLACE, &total_sum,
              1, MPI_INT,
              MPI_SUM, comm);
```

MPI_IN_PLACE cheat sheet

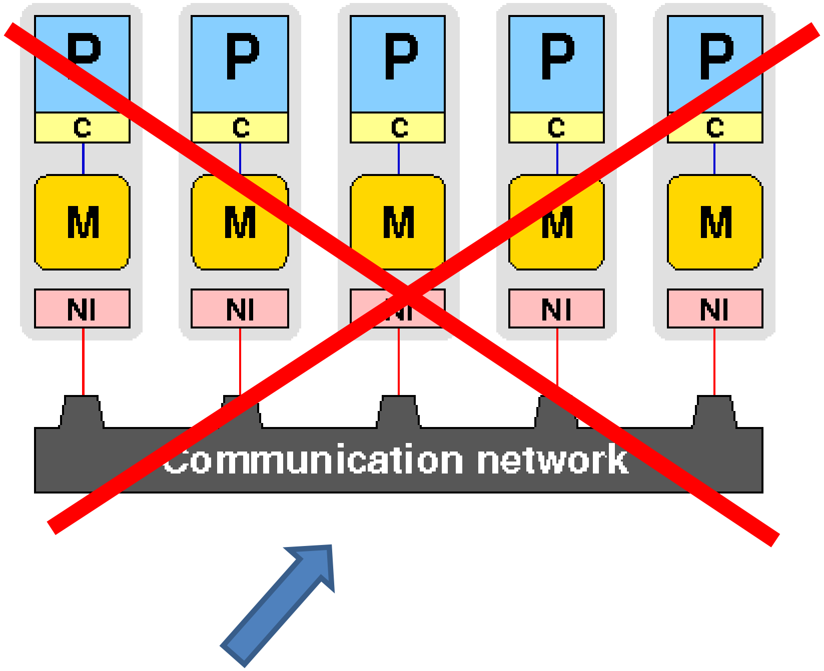
Function	MPI_IN_PLACE argument	@ rank(s)	Comment [MPI 3.0]
MPI_GATHER	send buffer	root	Recv value at root already in the correct place in receive buffer.
MPI_GATHERV	send buffer	root	Recv value at root already in the correct place in receive buffer.
MPI_SCATTER	receive buffer	root	Root-th segment of send buffer is not moved.
MPI_SCATTERV	receive buffer	root	Root-th segment of send buffer is not moved.
MPI_ALLGATHER	send buffer	all	Input data at the correct place were process would receive its own contribution.
MPI_ALLGATHERV	send buffer	all	Input data at the correct place were process would receive its own contribution.
MPI_ALLTOALL	send buffer	all	Data to be sent is taken from receive buffer and replaced by received data, data sent/received must be of the same type map specified in receive count/receive type.
MPI_ALLTOALLV	send buffer	all	Data to be sent is taken from receive buffer and replaced by received data. Data sent/received must be of the same type map specified in receive count/receive type. The same amount of data and data type is exchanged between two processes.
MPI_REDUCE	send buffer	root	Data taken from receive buffer, replaced with output data.
MPI_ALLREDUCE	send buffer	all	Data taken from receive buffer, replaced with output data.

Summary of MPI collective communication

- MPI (blocking) collectives
  - All ranks in communicator must call the function
- Communication and synchronization
  - Barrier, broadcast, scatter, gather, and combinations thereof
- Global operations
  - Reduce, allreduce, some more…
- In-place buffer specification **`MPI_IN_PLACE`**
  - Save some space if need be

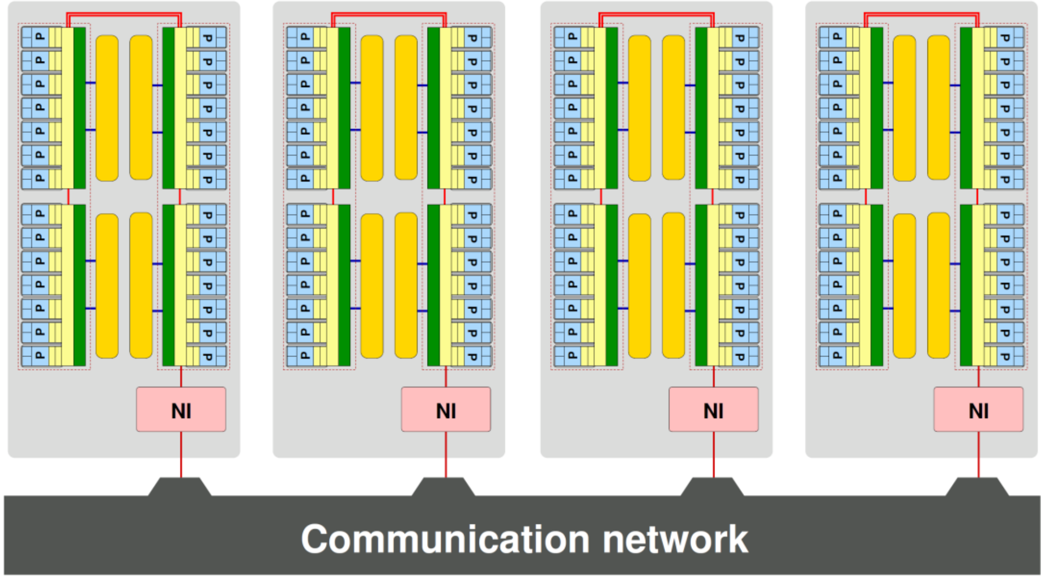
Distributed-memory system architecture

Distributed-memory parallel computers today



Long gone!

- Clusters of shared-memory nodes
- ccNUMA per node
- Multiple cores per ccNUMA domain



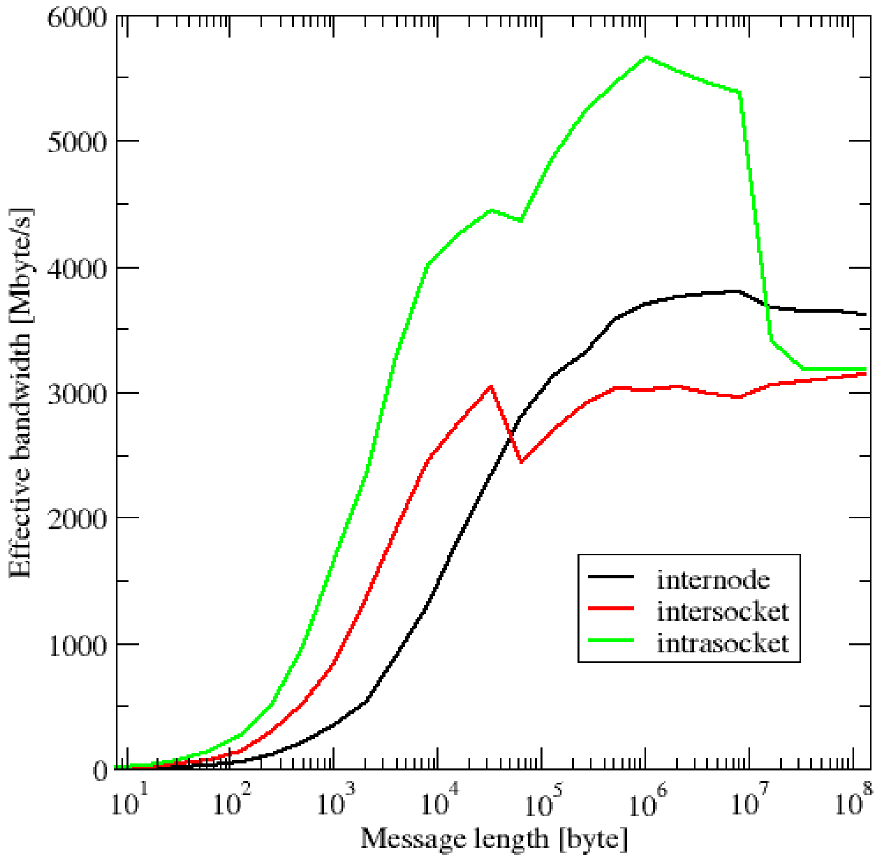
Point-to-point data transmission performance

- Simple “Hockney model” for data transfer time

  $$T_{comm} = \lambda + \frac{V}{b},\ \ B_{\mathrm{eff}} = \frac{V}{T_{comm}}$$

  λ: latency, b: asymptotic BW

- Reality is more complicated
  - System topology
  - Protocol switches
  - Contention effects



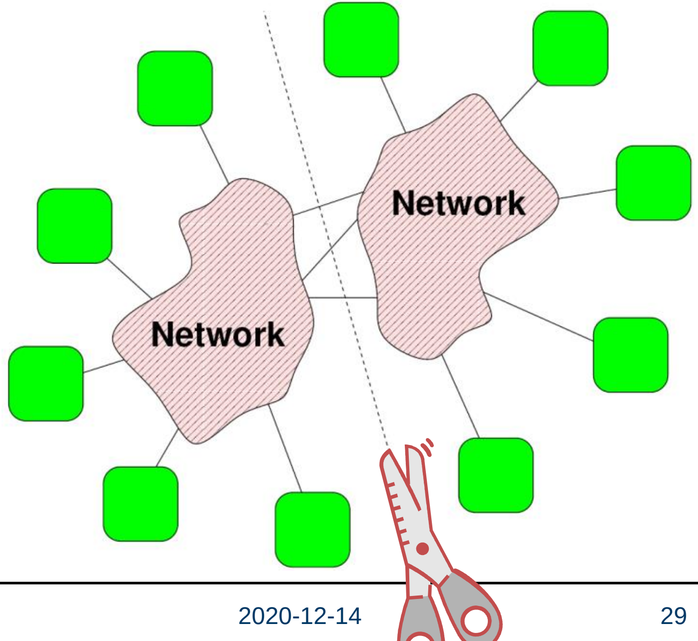
Characterizing communication networks

- Network bisection bandwidth B_b is a general metric for the data transfer "capability" of a system:

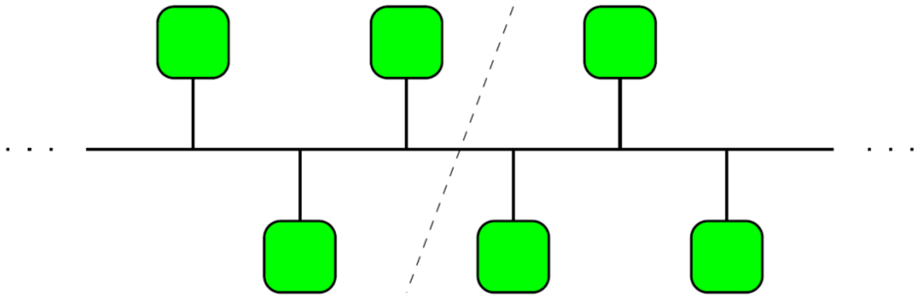
  Minimum sum of the bandwidths of all connections cut when splitting the system into two equal parts

- More meaningful metric for system scalability: bisection BW per node: B_b/N_{nodes}

- Bisection BW depends on
  - Bandwidth per link
  - Network topology



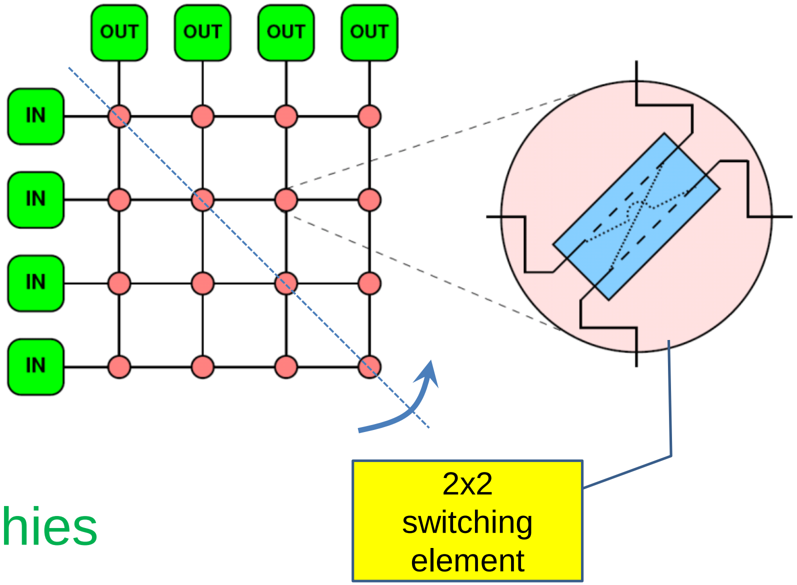
Network topologies: bus



- Bus can be used by one connection at a time
- Bandwidth is shared among all devices
- Bisection BW is constant → $B_b/N_{nodes} \sim 1/N_{nodes}$
- Examples: diagnostic buses, old Ethernet network with hubs, Wi-Fi channel
- Advantages
  - Low latency
  - Easy to implement
- Disadvantages
  - Shared bandwidth, not scalable
  - Problems with failure resiliency (one defective agent may block bus)
  - Large signal power per agent

Network topologies: non-blocking crossbar

- Non-blocking crossbar can mediate a number of connections among groups of input and output elements
- This can be used as a n-port non-blocking switch (fold at the secondary diagonal)



- Switches can be cascaded to form hierarchies (common case)
  - Allows scalable communication at high hardware/energy costs
  - Crossbars are rarely used as interconnects for large scale computers
    - NEC SX9 vector system (“IXS”)

Network topologies: switches and fat trees

- Standard clusters are built with switched networks
- Compute nodes ("devices") are split up in groups – each group is connected to single (non-blocking crossbar-)switch ("leaf switches")
- Leaf switches are connected with each other using an additional switch hierarchy ("spine switches") or directly (for small configurations)
- Switched networks: "Distance" between any two devices is heterogeneous (number of "hops" in switch hierarchy)
- Diameter of network: The maximum number of hops required to connect two arbitrary devices (e.g., diameter of bus=1)

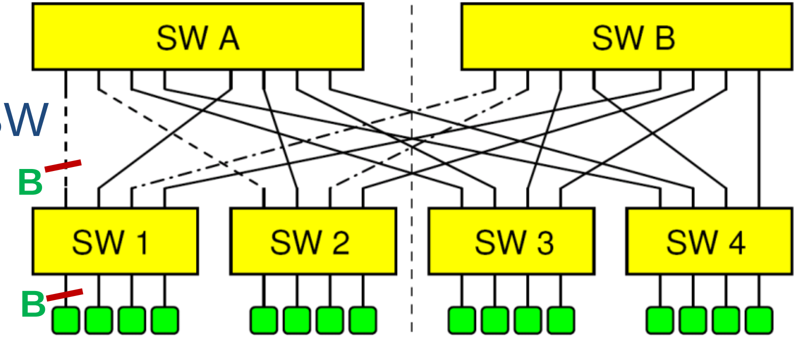
- "Perfect" world: "Fully non-blocking", i.e. any choice of $N_{nodes}/2$ disjoint node (device) pairs can communicate at full speed

Fat tree switch hierarchies

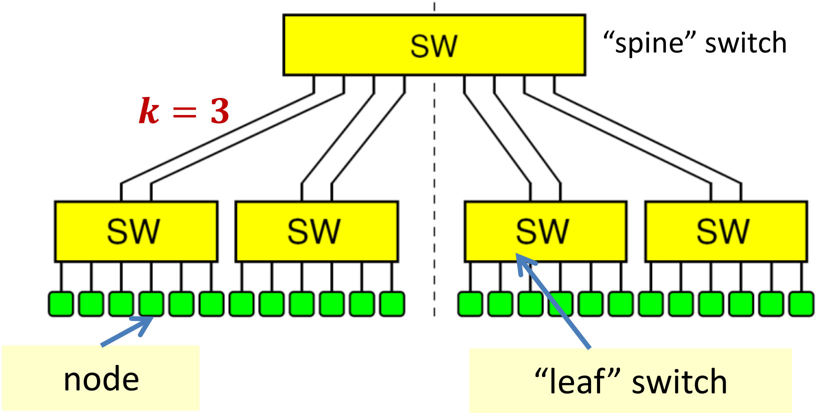
- "Fully non-blocking"
  - N_{nodes}/2 end-to-end con-nections with full BW

    $\rightarrow B_b = B \times N_{nodes}/2, \; B_b/N_{nodes} = B/2$

  - Sounds good, but see next slide

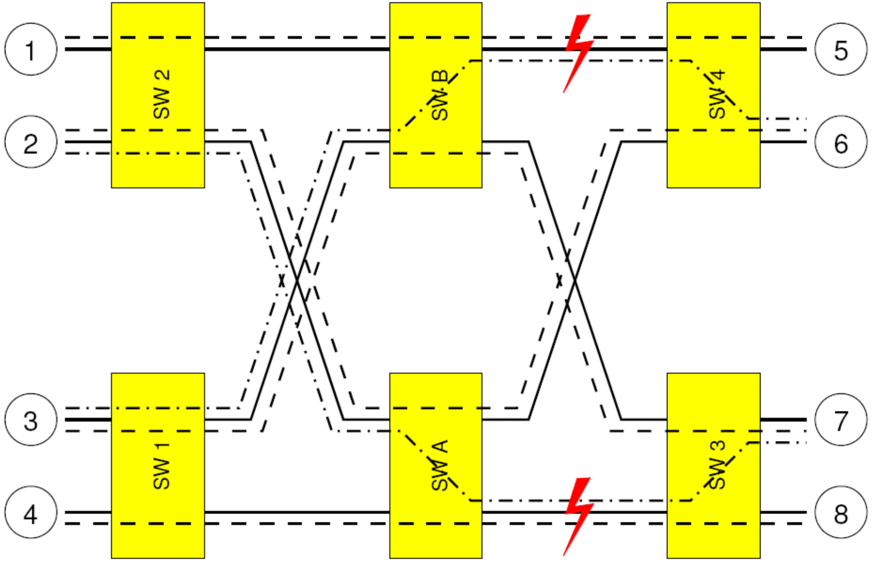


- "Oversubscribed"
  - Spine does not support $N_{nodes}/2$ full BW end-to-end connections
  - $B_b/N_{nodes} = const. = \; B/(2k)$, with k = oversubscription factor
  - Resource management (job placement) is crucial



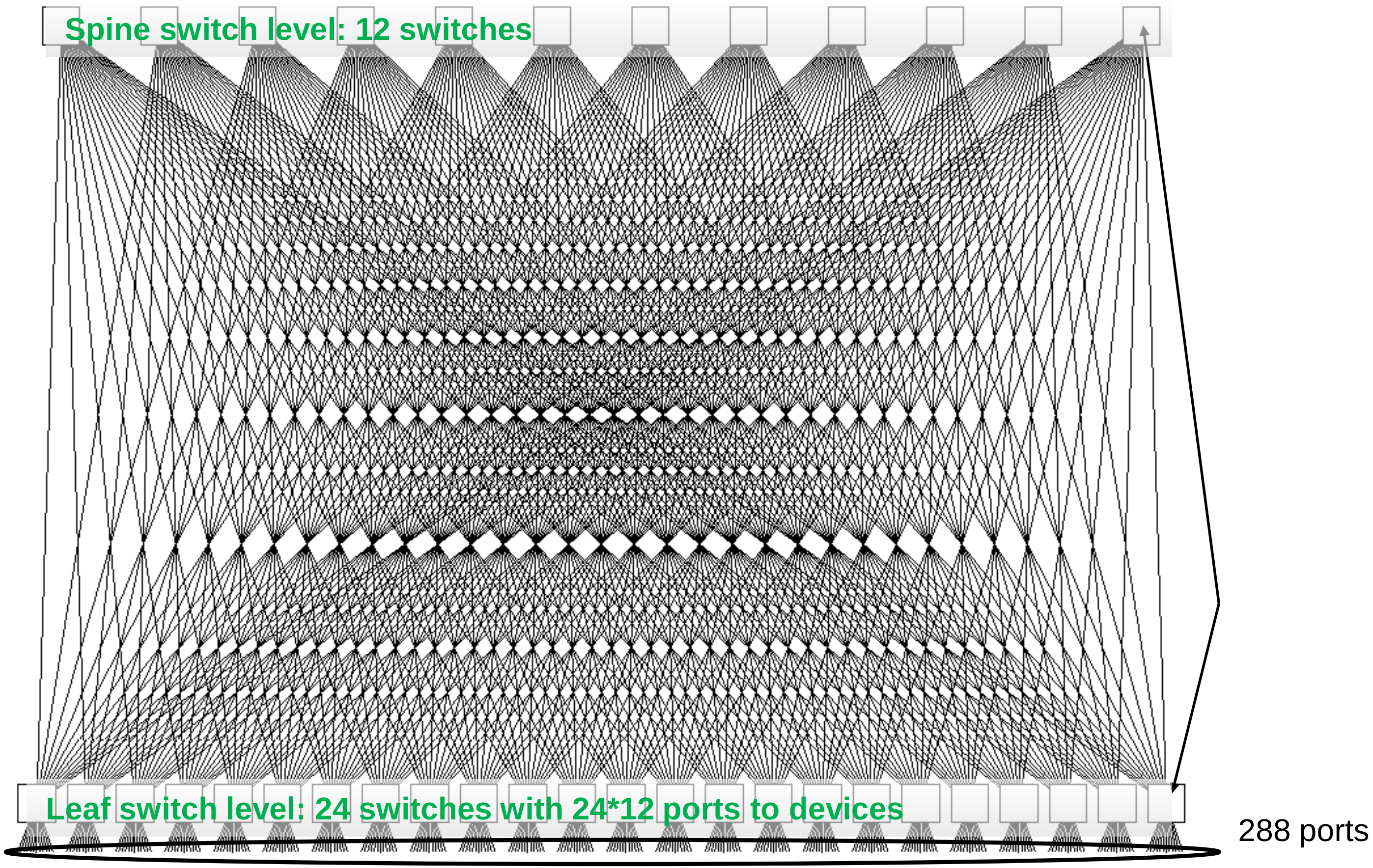
Fat trees and static routing

- If all end-to-end data paths are preconfigured ("static routing"), not all possible combinations of N agents will get full bandwidth
- Example: – – – – is a collision-free pattern here (1→5, 2→6,3→7, 4→8)
- Change (2→6,3→7) to (2→7,3→6): – · – · – has collisions if no other connections are re-routed at the same time



- Static routing: potential collisions even for full fat tree
- Dynamic/adaptive routing: collision mitigation

A “single” 288-port InfiniBand DDR switch



Examples for fat tree networks in HPC

- Ethernet
  - 1 Gbit/s &10 & 100 Gbit/s variants

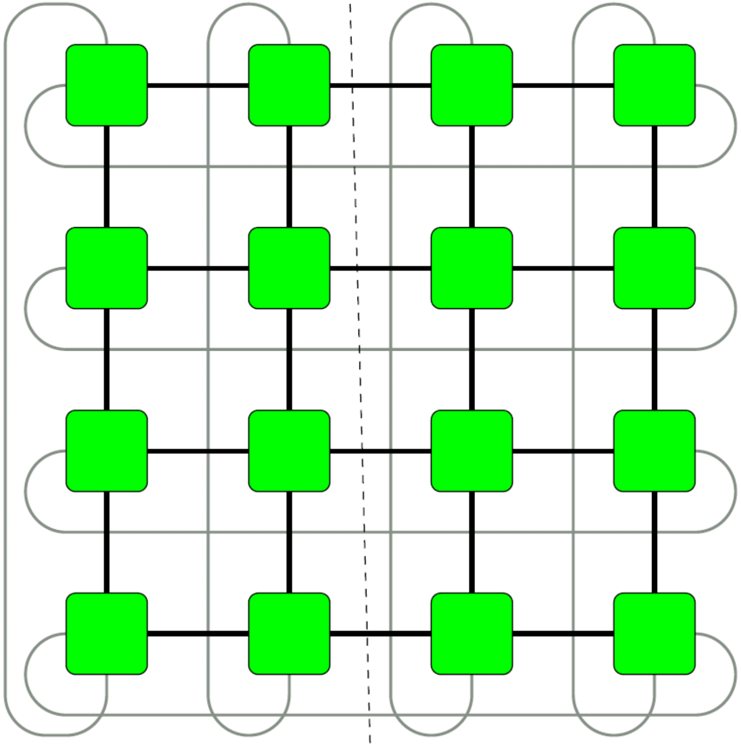
- InfiniBand: Dominant high-performance “commodity” interconnect
  - DDR: 20 Gbit/s per link and direction (Building blocks: 24-port switches)
  - QDR: 40 Gbit/s per link and direction, building blocks: 36-port switches
    → “Large” 36x18=648-port switches
  - FDR-10 / FDR: 40/56 Gbit/s per link and direction
  - EDR: 100 Gbit/s per link and direction, HDR: 200 Gbit/s

- Expensive & complex to scale to very high node counts

Mesh networks

- Fat trees can become prohibitively expensive in large systems
- Compromise: Meshes
  - n-dimensional Hypercubes
  - Toruses (2D / 3D)
  - Many others (including hybrids)
- Each node is a "router"
- Direct connections only between direct neighbors

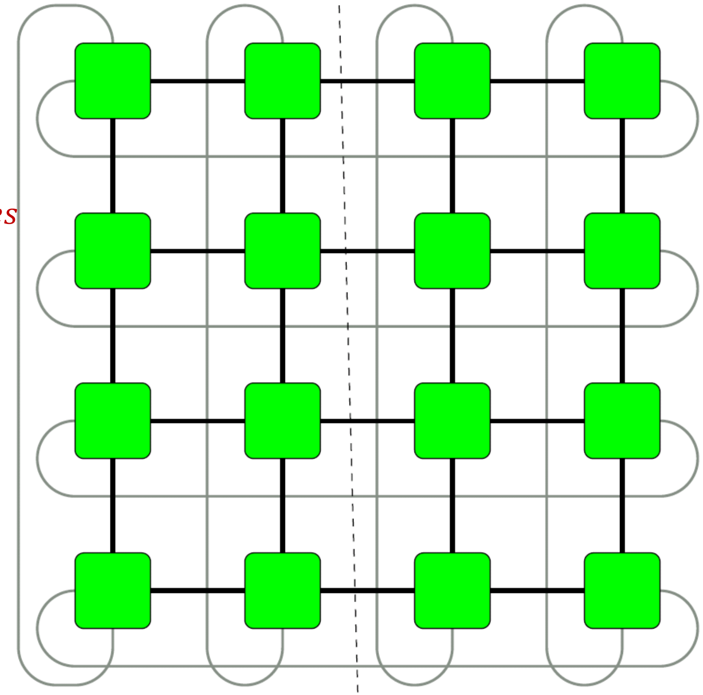
Example: 2D torus mesh



Mesh networks

- This is not a non-blocking corossbar!
  - Intelligent resource management and routing algorithms are essential

- Toruses at very large systems:
  Cray XE/XK series, IBM Blue Gene
  - $B_b \sim N_{nodes}^{(d-1)/d}$ → $B_b / N_{nodes} \rightarrow 0$ for large N_{nodes}
  - Sounds bad, but those machines show good scaling for many codes
  - Well-defined and predictable bandwidth behavior!



Summary of distributed-memory architecture

- “Pure” distributed-memory parallel systems are rare
  - Hierarchical parallelism rules
- Simple latency/bandwidth model good for insights, but unrealistic
  - Protocol switches, contention
- Wide variety of network topologies available
  - Nonblocking crossbar
  - Fat tree
  - Meshes (torus, hypercube, hybrids)