Introduction to the Message Passing Interface (MPI)

Jan Fostier

May 3rd 2017

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Moore’s Law

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

16 Core SPARC T3
Six Core Core i7
Six Core Xeon 7400
10 Core Xeon Westmere EX
Dua Core tanium 2
8 core POWER7
Quad core z196
Quad Core tanium Tukwi a
8 Core Xeon Neha em EX
AMD K10
POWER6
tanium 2 with 9MB cache
Six Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Ce
tanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6
AMD K6
Pentium
Pentium
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

curve shows transistor count doub ing every two years

Transistor count: 2,600,000,000; 1,000,000,000; 100,000,000; 10,000,000; 1,000,000; 100,000; 10,000; 2,300

Date of introduction: 1971; 1980; 1990; 2000; 2011

“Transistor count doubles every two years”

Illustration from Wikipedia

Moore's Law

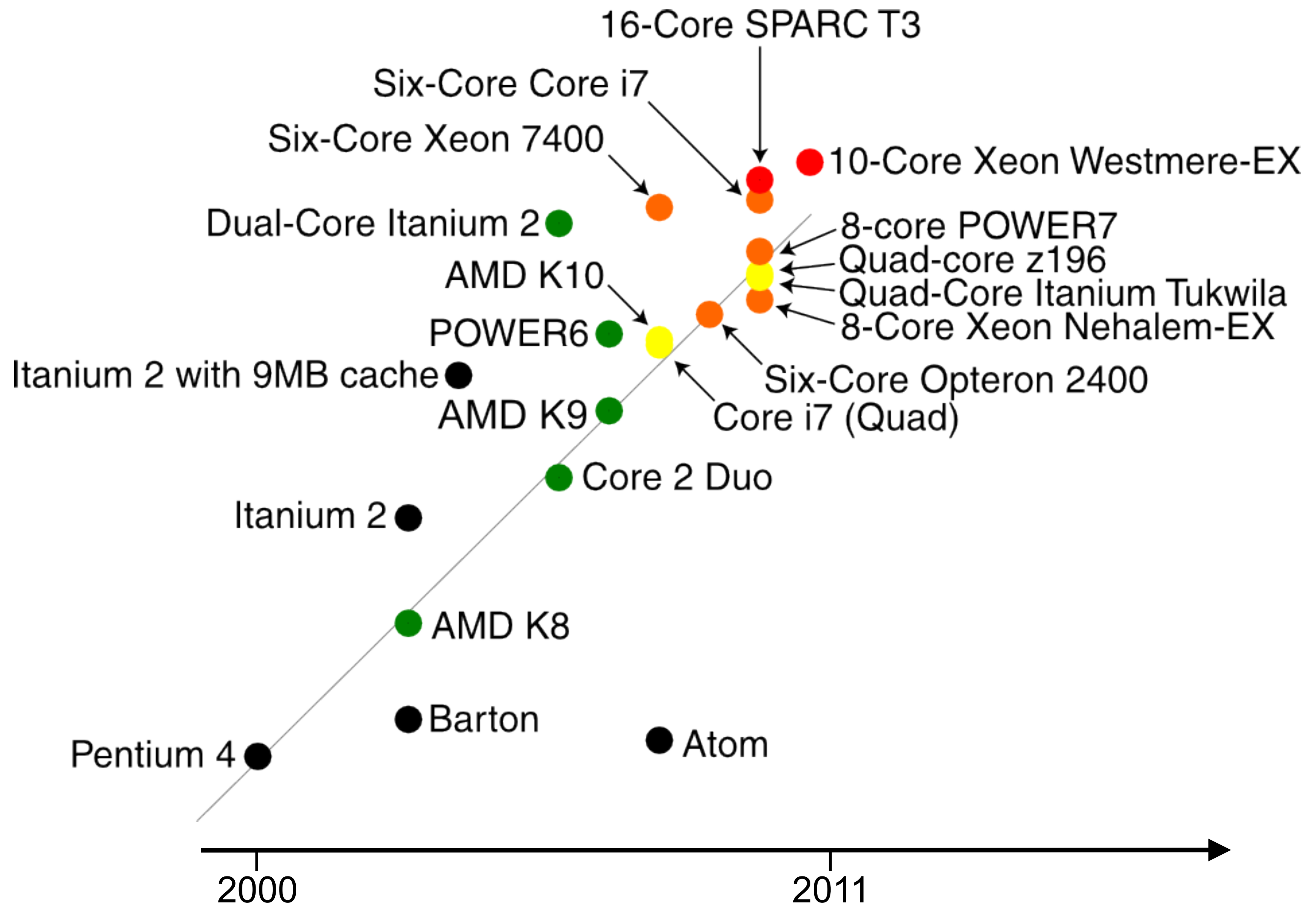
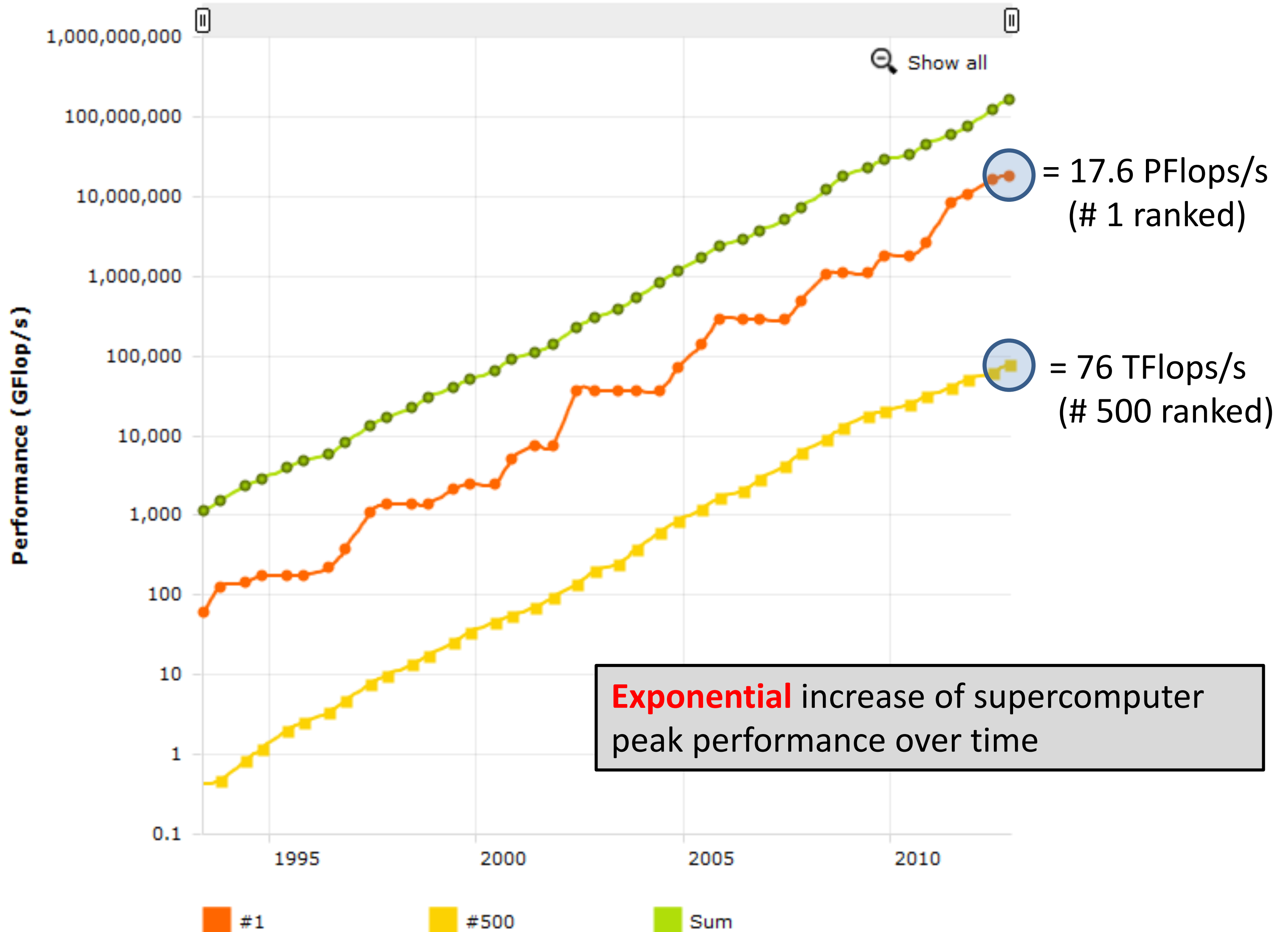


Illustration from Wikipedia

Evolution of top 500 supercomputers over time



Application area – performance share

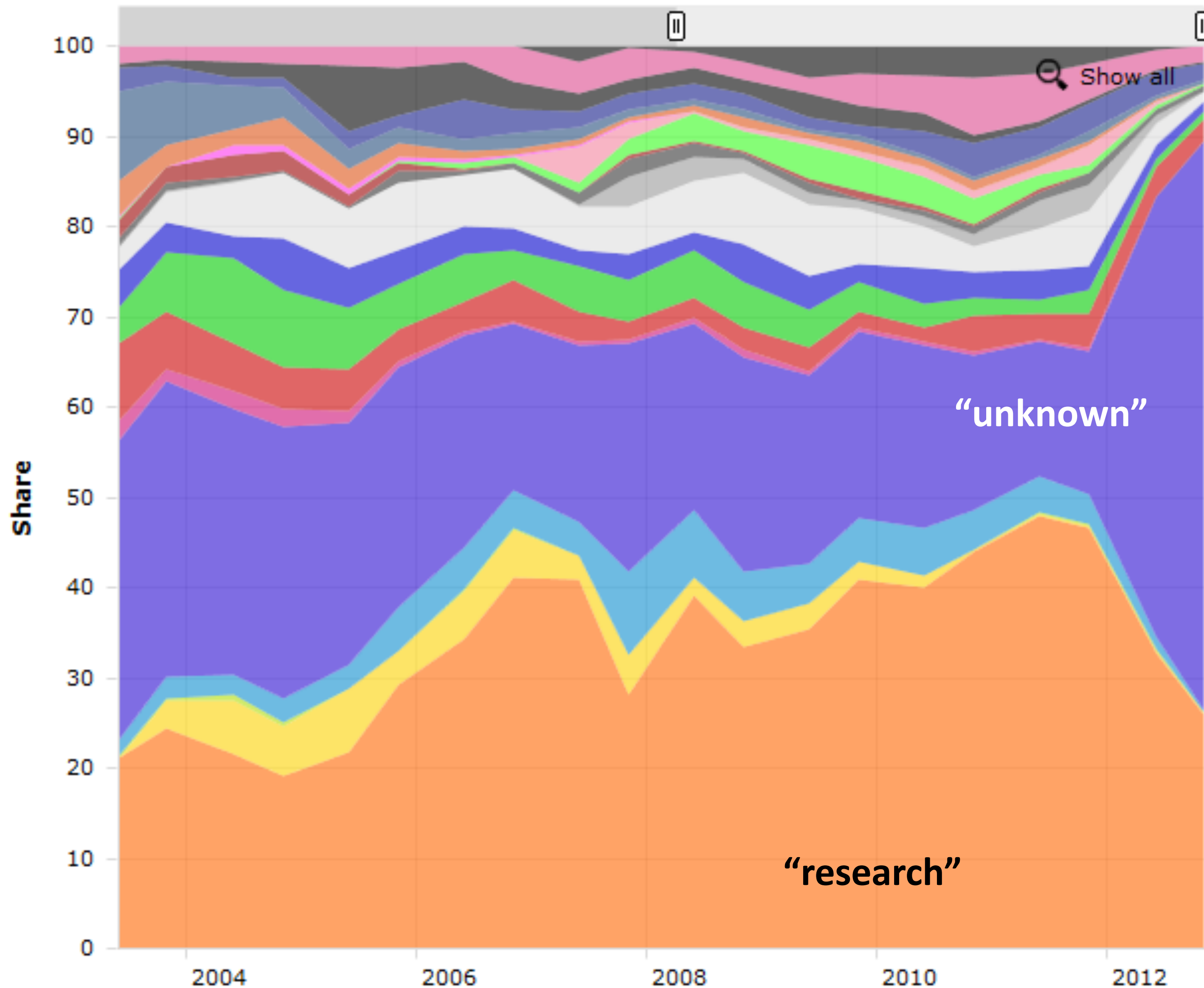
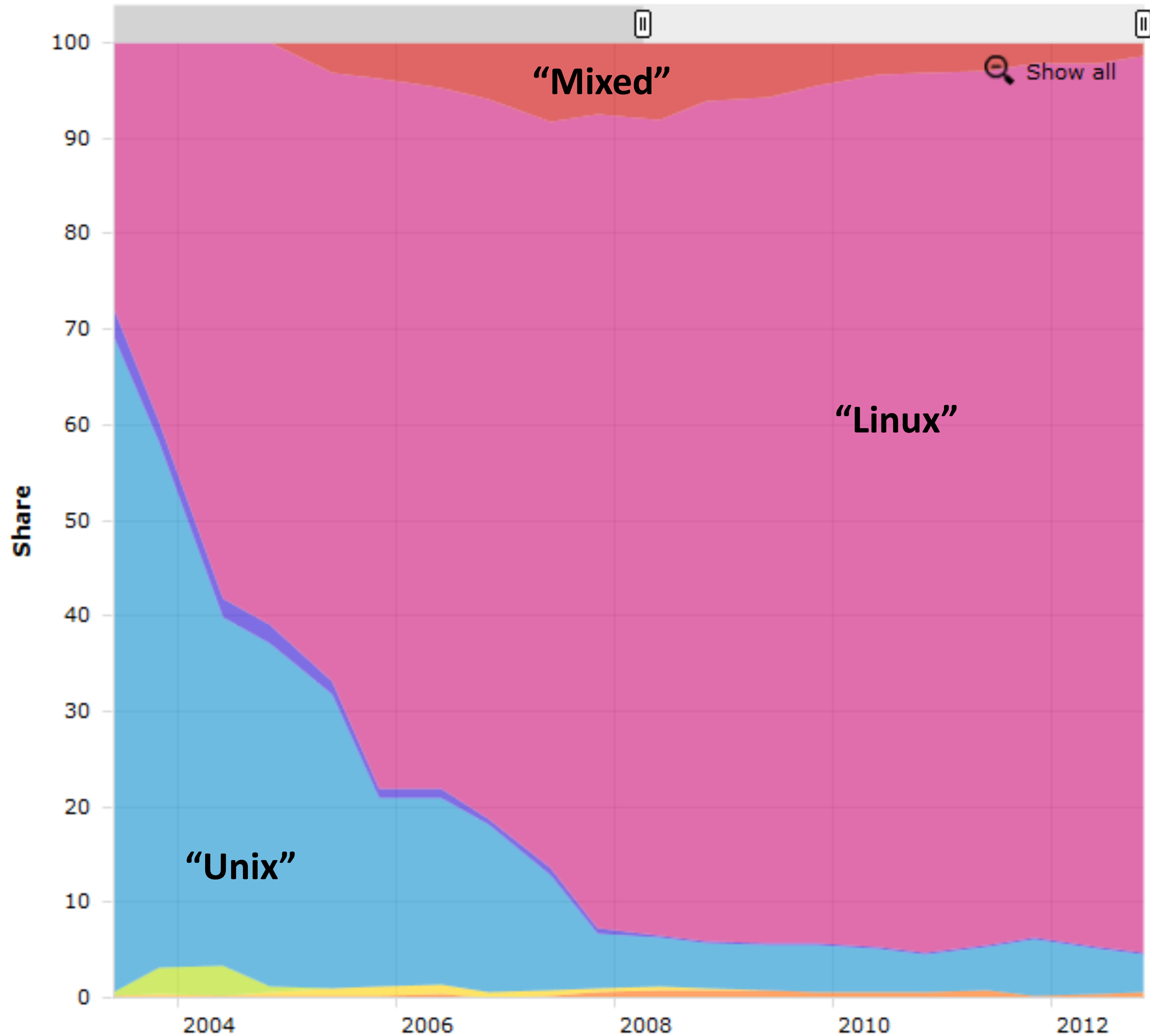


Image taken from www.top500.org

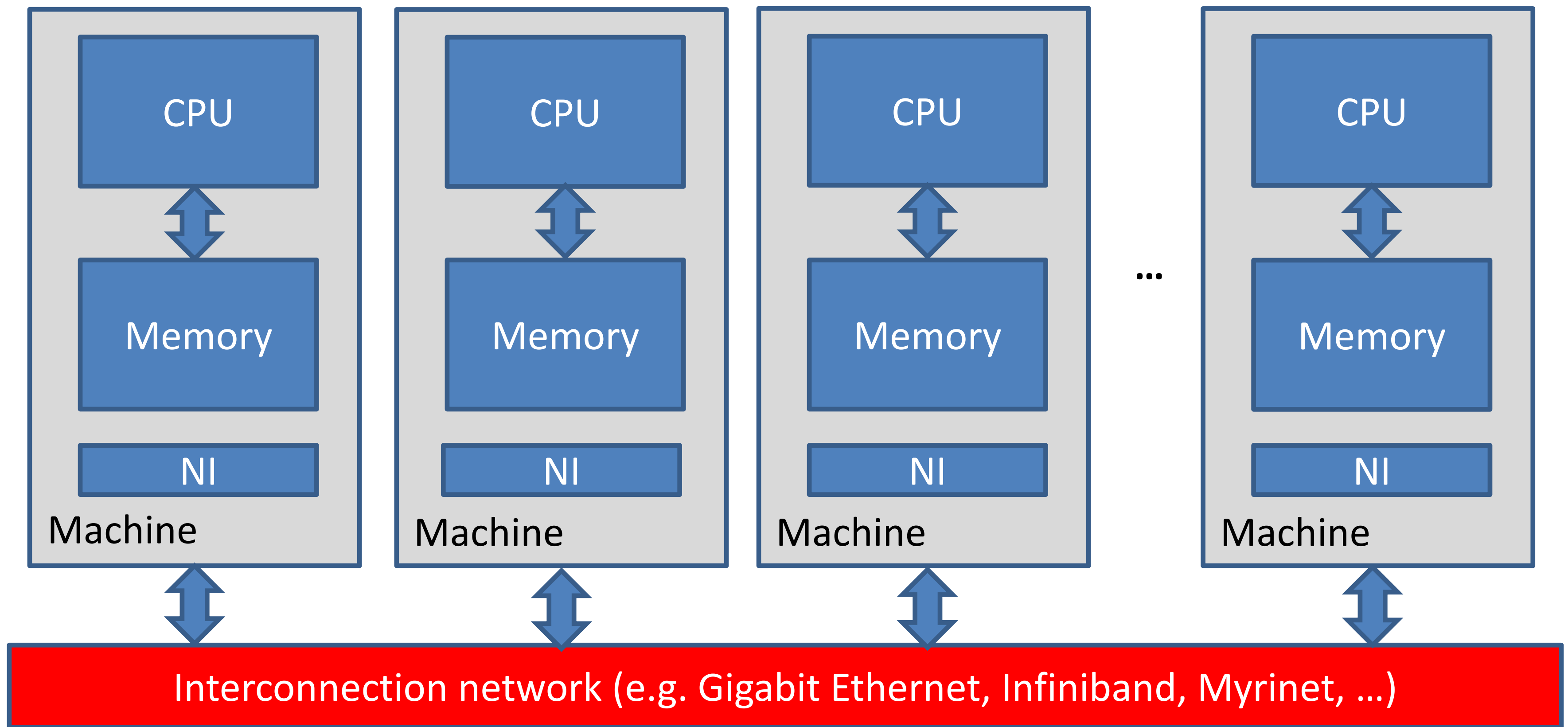
Operating system family



Motivation for parallel computing

- Want to run the **same program faster**
  - Depends on the application what is considered an acceptable runtime
    - **SETI@Home, Folding@Home, GIMPS**: years may be acceptable
    - For **R&D** applications: days or even weeks are acceptable
      - CFD, CEM, Bioinformatics, Cheminformatics, and many more
    - **Prediction of tomorrow's weather** should take less than a day of computation time.
    - Some applications require **real-time behavior**
      - Computer games, algorithmic trading
- Want to run **bigger datasets**
- Want to reduce **financial cost** and/or **power consumption**

Distributed-memory architecture



Outline

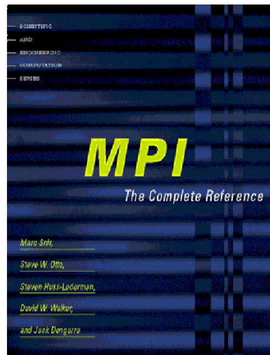
- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
    - Point-to-point communication
        - Blocking communication
        - Point to point network performance
        - Non-blocking communication
    - Collective communication
        - Collective communication algorithms
        - Global network performance
- Parallel program performance evaluation
    - Amdahl’s law
    - Gustafson’s law
- Parallel program development: case studies

Message Passing Interface (MPI)

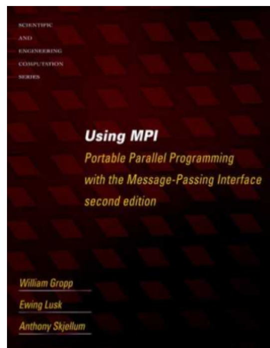
- MPI = **library specification**, not an implementation
- Most important implementations
  - Open MPI (http://www.open-mpi.org, MPI-3 standard)
  - Intel MPI (proprietary, MPI-3 standard)
- Specifies routines for (among others)
  - Point-to-point communication (between 2 processes)
  - Collective communication (> 2 processes)
  - Topology setup
  - Parallel I/O
- Bindings for C/C++ and Fortran

MPI reference works

- **MPI standards:** http://www.mpi-forum.org/docs/



- **MPI: The Complete Reference** (M. Snir, S. Otto, S Huss-Lederman, D. Walker, J. Dongarra)
  Available from http://switzernet.com/people/emin-gabrielyan/060708-thesis-ref/papers/Snir96.pdf



- **Using MPI: Portable Parallel Programming with the Message Passing Interface**, 2nd ed. (W. Gropp, E. Lusk, A. Skjellum).

MPI standard

- Started in 1992 (Workshop on Standards for Message-Passing in a Distributed Memory Environment) with support from vendors, library writers and academia.
- **MPI version 1.0** (May 1994)
  - Final pre-draft in 1993 (Supercomputing ‘93 conference)
  - Final version June 1994
- **MPI version 2.0** (July 1997)
  - Support for one-sided communication
  - Support for process management
  - Support for parallel I/O
- **MPI version 3.0** (September 2012)
  - Support for non-blocking collective communication
  - Fortran 2008 bindings
  - New one-sided communication routines

Hello world example in MPI

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    cout << "Hello World from process" << rank << "/" << size << endl;

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

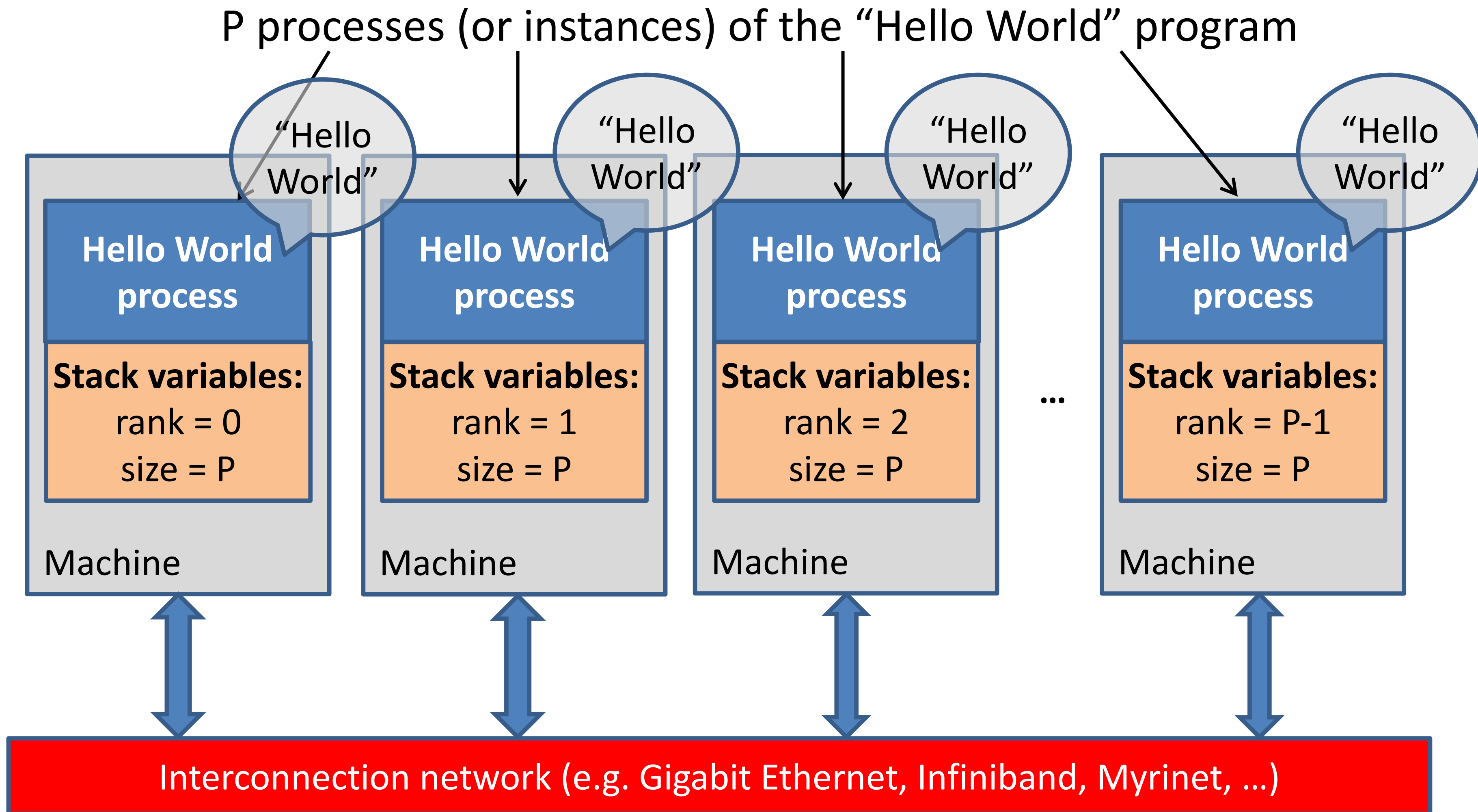
Output **order** is random

```
john@doe ~]$ mpirun -np 4 ./helloWorld
Hello World from process 2/4
Hello World from process 3/4
Hello World from process 0/4
Hello World from process 1/4
```

Basic MPI routines

- `int MPI_Init(int *argc, char ***argv)`
  - Initialization: all processes must call this prior to any other MPI routine.
  - Strips of (possible) arguments provided by “mpirun”.
- `int MPI_Finalize(void)`
  - Cleanup: all processes must call this routine at the end of the program.
  - All pending communication should have finished before calling this.
- `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - Returns the size of the “Communicator” associated with “comm”
  - Communicator = user defined subset of processes
  - MPI_COMM_WORLD = communicator that involves all processes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - Return the rank of the process in the Communicator
  - Range: [0 ... size – 1]

Message Passing Interface Mechanisms



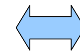
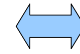
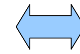
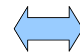
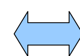
`mpirun` launches P **independent processes** across the different machines

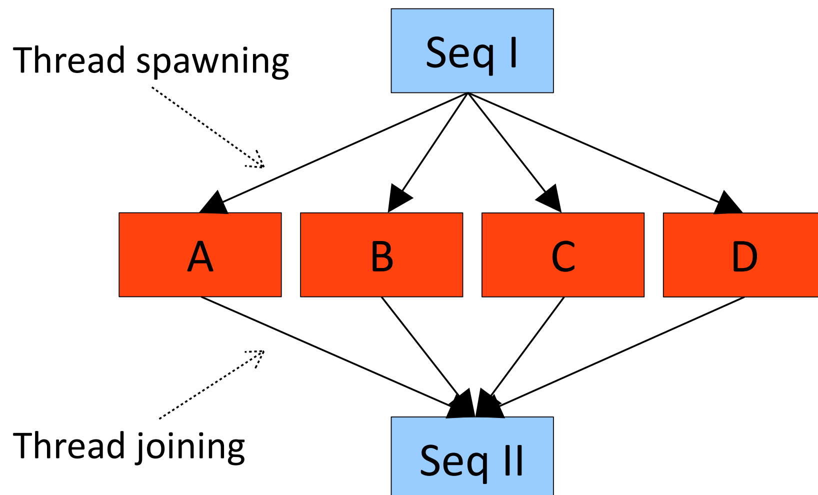
- Each process is a instance of the **same program**

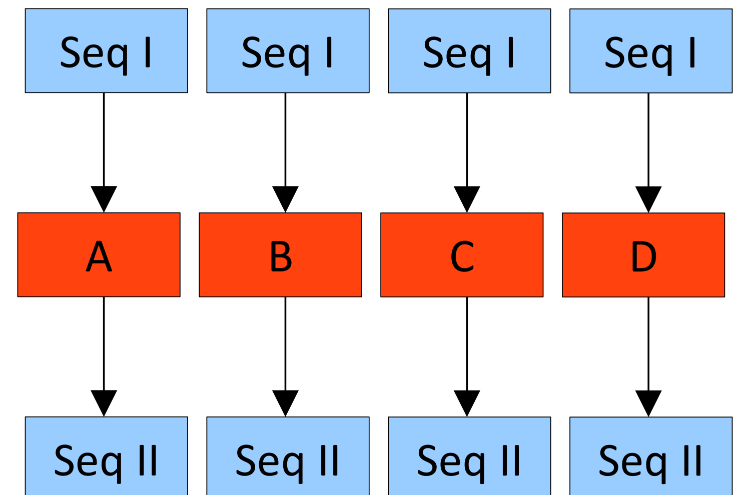
Terminology

- **Computer program** = passive collection of instructions.
- **Process** = instance of a computer program that is being executed.
- **Multitasking** = running multiple processes on a CPU.
- **Thread** = smallest stream of instructions that can be managed by an OS scheduler (= light-weight process).
- **Distributed-memory** system = multi-processor systems where each processor has direct access (fast) to its own private memory and relies on inter-processor communication to access another processor’s memory (typically slower).

Multithreading versus multiprocessing

Multi-threading		**Message passing**
Single process		Multiple processes
Shared memory address space		Separate memory address spaces
Protect data against simultaneous writing		Explicitly communicate everything
Limited to a single machine		Multiple machines possible
E.g. Pthreads, CILK, OpenMP, etc.		E.g. MPI, Unified Parallel C, PVM





Message Passing Interface (MPI)

- **MPI mechanisms (depends on implementation)**
  - Compiling an MPI program from source
    - mpicc -O3 main.cpp -o main

      gcc -O3 main.cpp -o main -L<IncludeDir> -l<mpiLibs>
    - Also mpic++ (or mpicxx), mpif77, mpif90, etc.
  - Running MPI applications (manually)
    - mpirun -np <number of program instances> <your program>
    - List of worker nodes specified in some config file.
- **Using MPI on the Ugent HPC cluster**
  - Load appropriate module first, e.g.
    - module load intel/2017a
  - Compiling an MPI program from source

    C
    - mpigcc (uses the GNU "gcc" C compiler)
    - mpiicc (uses the Intel "icc" C compiler)

    C++
    - mpigxx (uses the GNU "g++" C++ compiler)
    - mpiicpc (uses the Intel "icpc" C++ compiler)
  - Submit job using a jobscript (see further)

mpicc / mpic++ defaults to gcc

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

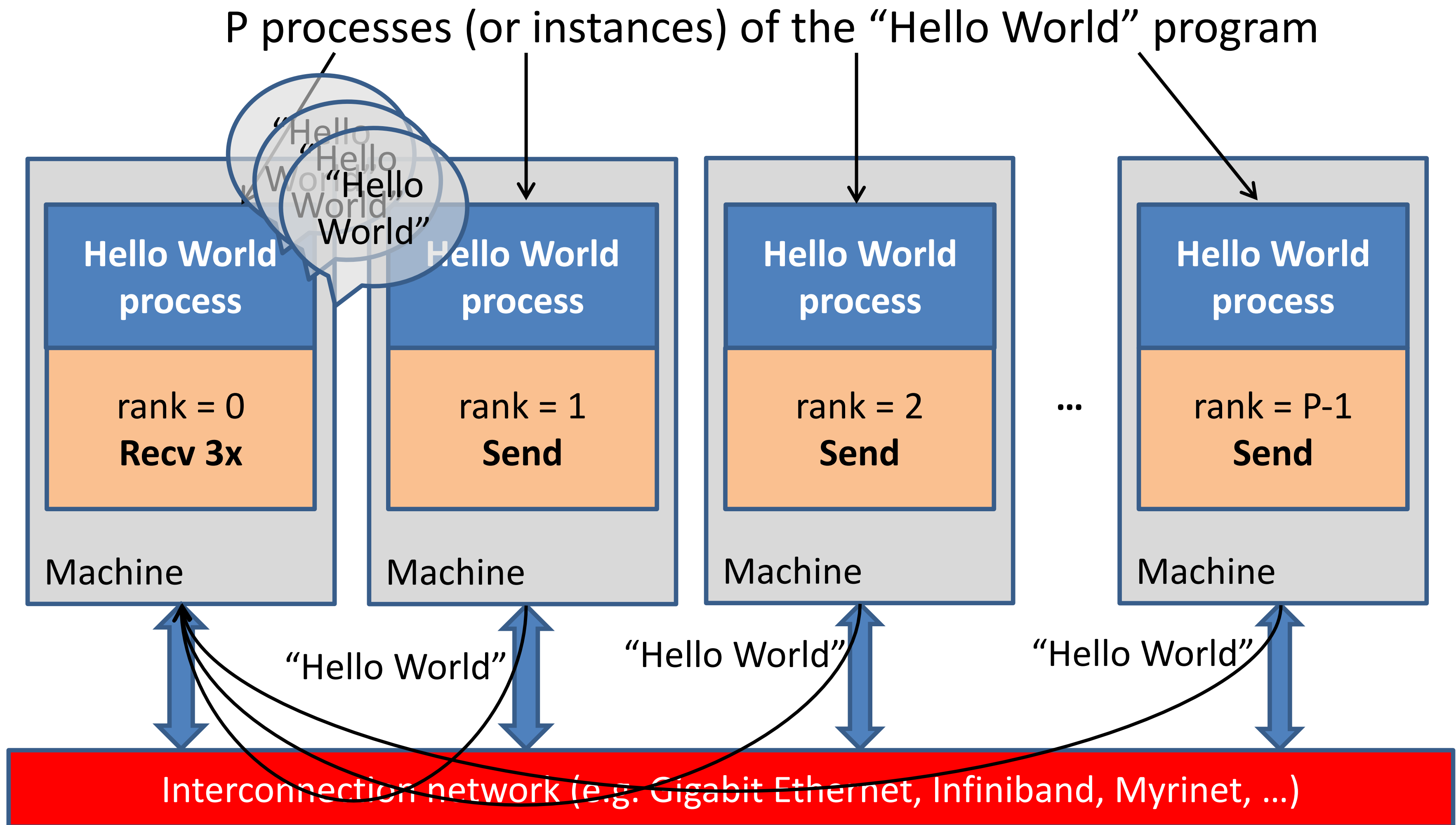
Basic MPI point-to-point communication

```
...
int rank, size, count;
char b[40];
MPI_Status status;

...   // init MPI and rank and size variables

if (rank != 0) {
  char * str = "Hello World";
  MPI_Send(str, 12, MPI_CHAR, 0, 123, MPI_COMM_WORLD);
} else {
  for (int i = 1; i < size; i++) {
    MPI_Recv(b, 40, MPI_CHAR, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &count);
    printf("I received %s from process %d with size %d and tag %d\n",
           b, status.MPI_SOURCE, count, status.MPI_TAG);
  }
}
...
```

branching on rank

```
john@doe ~]$ mpirun -np 4 ./ptpcomm
I received Hello World from process 1 with size 12 and tag 123
I received Hello World from process 2 with size 12 and tag 123
I received Hello World from process 3 with size 12 and tag 123
```

Message Passing Interface Mechanisms



`mpirun` launches P **independent processes** across the different machines

- Each process is a instance of the **same program**

Blocking send and receive

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

- **buf**: pointer to the message to send
- **count**: number of items to send
- **datatype**: datatype of each item
  - *number of bytes sent: count * sizeof(datatype)*
- **dest**: rank of destination process
- **tag**: value to identify the message [0 ... at least (32 767)]
- **comm**: communicator specification (e.g. MPI_COMM_WORLD)

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

- **buf:** pointer to the buffer to store received data
- **count**: upper bound (!) of the number of items to receive
- **datatype**: datatype of each item
- **source**: rank of source process (or MPI_ANY_SOURCE)
- **tag**: value to identify the message (or MPI_ANY_TAG)
- **comm**: communicator specification (e.g. MPI_COMM_WORLD)
- **status**: structure that contains { MPI_SOURCE, MPI_TAG, MPI_ERROR }

Sending and receiving

Two-sided communication:

- Both the sender and receiver are involved in data transfer

  As opposed to one-sided communication

- Posted send must match receive

When do MPI_Send and MPI_recv match ?

- 1. Rank of *receiver* process
- 2. Rank of *sending* process
- 3. *Tag*

  - custom value to distinguish messages from same sender

- 4. *Communicator*

Rationale for Communicators

- Used to create subsets of processes
- Transparent use of tags
  - modules can be written in isolation
  - communication within module through own Communicator
  - communication between modules through shared Communicator

MPI Datatypes

MPI_Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long in
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	no conversion, bitpattern transferred as is
MPI_PACKED	grouped messages

Querying for information

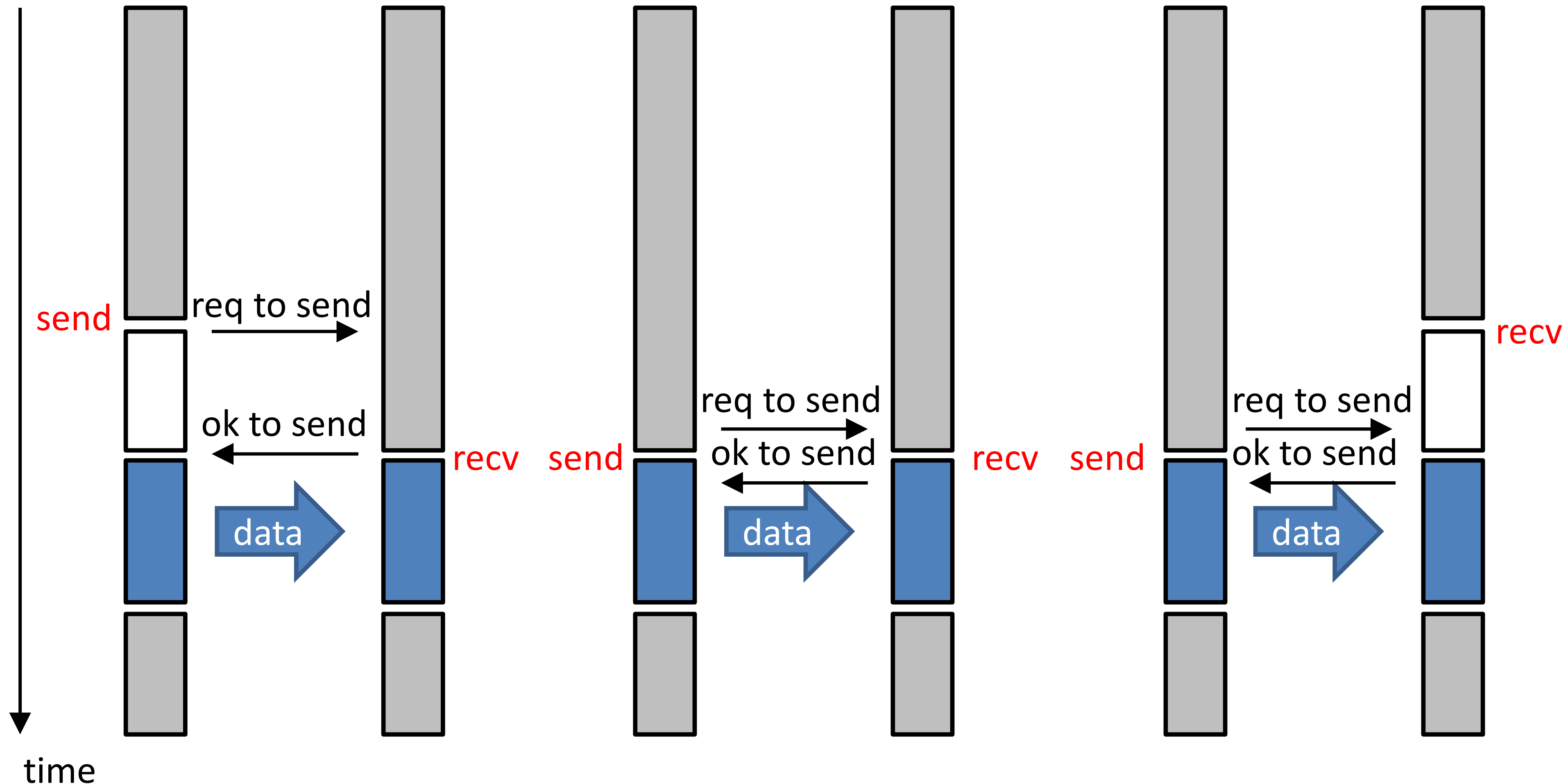
- **MPI_Status**
  - Stores information about the MPI_Recv operation

```
typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
}
```

  - Does not contain the size of the received message

- `int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)`
  - returns the number of data items received in the count variable
  - not directly accessible from status variable

Blocking send and receive



a) sender comes first, **idling at sender** (no buffering of message)

b) sending/receiving at about the same time, **idling minimized**

c) receiver comes first, **idling at receiver**

Slide reproduced from slides by John Mellor - Crummey

Deadlocks

```
int a[10], b[10], myRank;
MPI_Status s1, s2;
MPI_Comm_rank( MPI_COMM_WORLD, &myRank);

if ( myRank == 0 ) {
    MPI_Send( a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD );
    MPI_Send( b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD );
}
else if ( myRank == 1 ) {
    MPI_Recv( b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &s1 );
    MPI_Recv( a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &s2 );
}
```

If MPI_Send is blocking (handshake protocol), this program will **deadlock**

- If 'eager' protocol is used, it may run
- Depends on message size

Slide credit: John Mellor - Crummey

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Network cost modeling

- Various choices of **interconnection network**
  - **Gigabit Ethernet**: cheap, but far too slow for HPC applications
  - **Infiniband /Myrinet**: high speed interconnect
- **Simple performance model** for point-to-point communication

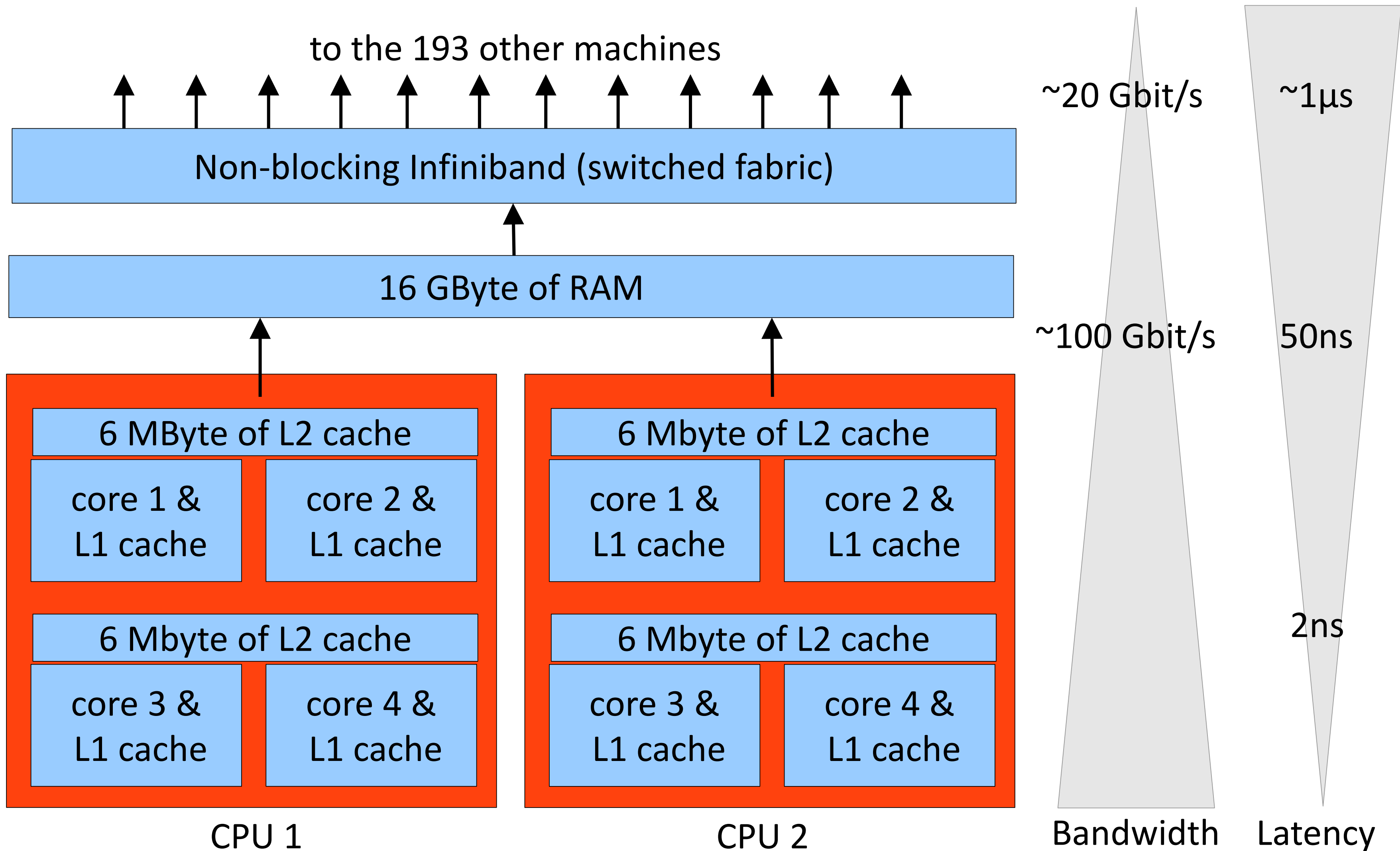
$$T_{comm} = \alpha + \beta * n$$

  - α = latency
  - $B = 1/\beta$ = saturation (asymptotic) bandwidth (bytes/s)
  - n = number of bytes to transmit
  - Effective bandwidth B_{eff}:

$$B_{eff} = \frac{n}{\alpha + \beta * n} = \frac{n}{\alpha + \frac{n}{B}}$$

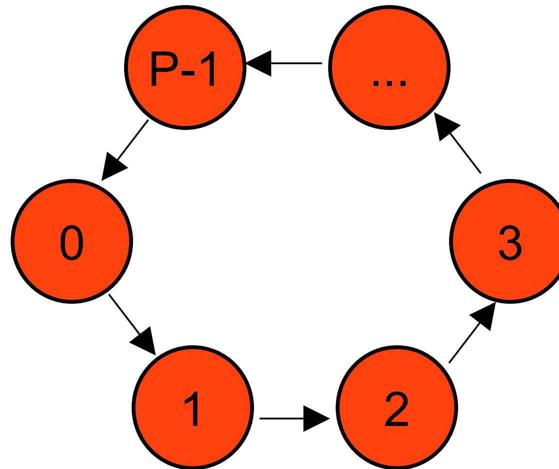
The case of Gengar (UGent cluster)

Bandwidth and latency



Measure effective bandwidth: ringtest

Idea: send a single message of size N in a circle



- Increase the message N size exponentially
  - 1 byte, 2 bytes, 4 bytes, ... 1024 bytes, 2048 bytes, 4096 bytes
- Benchmark the results (measure wall clock time T), ...
  - Bandwidth = N * P / T

Hands-on: ringtest in MPI

```
void sendRing( char *buffer, int length ) {
    /* send message in a ring here */
}

int main( int argc, char * argv[] )
{
    ...
    char *buffer = (char*) calloc ( 1048576, sizeof(char) );
    int msgLen = 8;
    for (int i = 0; i < 18; i++, msgLen *= 2) {
        double startTime = MPI_Wtime();
        sendRing( buffer, msgLen );
        double stopTime = MPI_Wtime();
        double elapsedSec = stopTime - startTime;
        if (rank == 0)
            printf( "Bandwidth for size %d is : %f\", ... );
    }
    ...
}
```

Jobscript example

mpijob.sh job script example:

```
#!/bin/sh
#
#PBS -o output.file
#PBS -e error.file
#PBS -l nodes=2:ppn=all
#PBS -l walltime=00:02:00
#PBS -m n

cd $VSC_SCRATCH/<yourdirectory>
module load intel/2017a
module load scripts
mympirun ./<program name> <program arguments>
```

qsub mpijob.sh

qstat / qdel / etc remains the same

Hands-on: ringtest in MPI (solution)

```
void sendRing( char *buffer, int msgLen )
{
    int myRank, numProc;
    MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
    MPI_Comm_size( MPI_COMM_WORLD, &numProc );
    MPI_Status status;

    int prevR = (myRank - 1 + numProc) % numProc;
    int nextR = (myRank + 1 ) % numProc;

    if (myRank == 0) {        // send first, then receive
        MPI_Send( buffer, msgLen, MPI_CHAR, nextR, 0, MPI_COMM_WORLD);
        MPI_Recv( buffer, msgLen, MPI_CHAR, prevR, 0, MPI_COMM_WORLD,
                  &status );
    } else {                  // receive first, then send
        MPI_Recv( buffer, msgLen, MPI_CHAR, prevR, 0, MPI_COMM_WORLD,
                  &status );
        MPI_Send( buffer, msgLen, MPI_CHAR, nextR, 0, MPI_COMM_WORLD);
    }
}
```

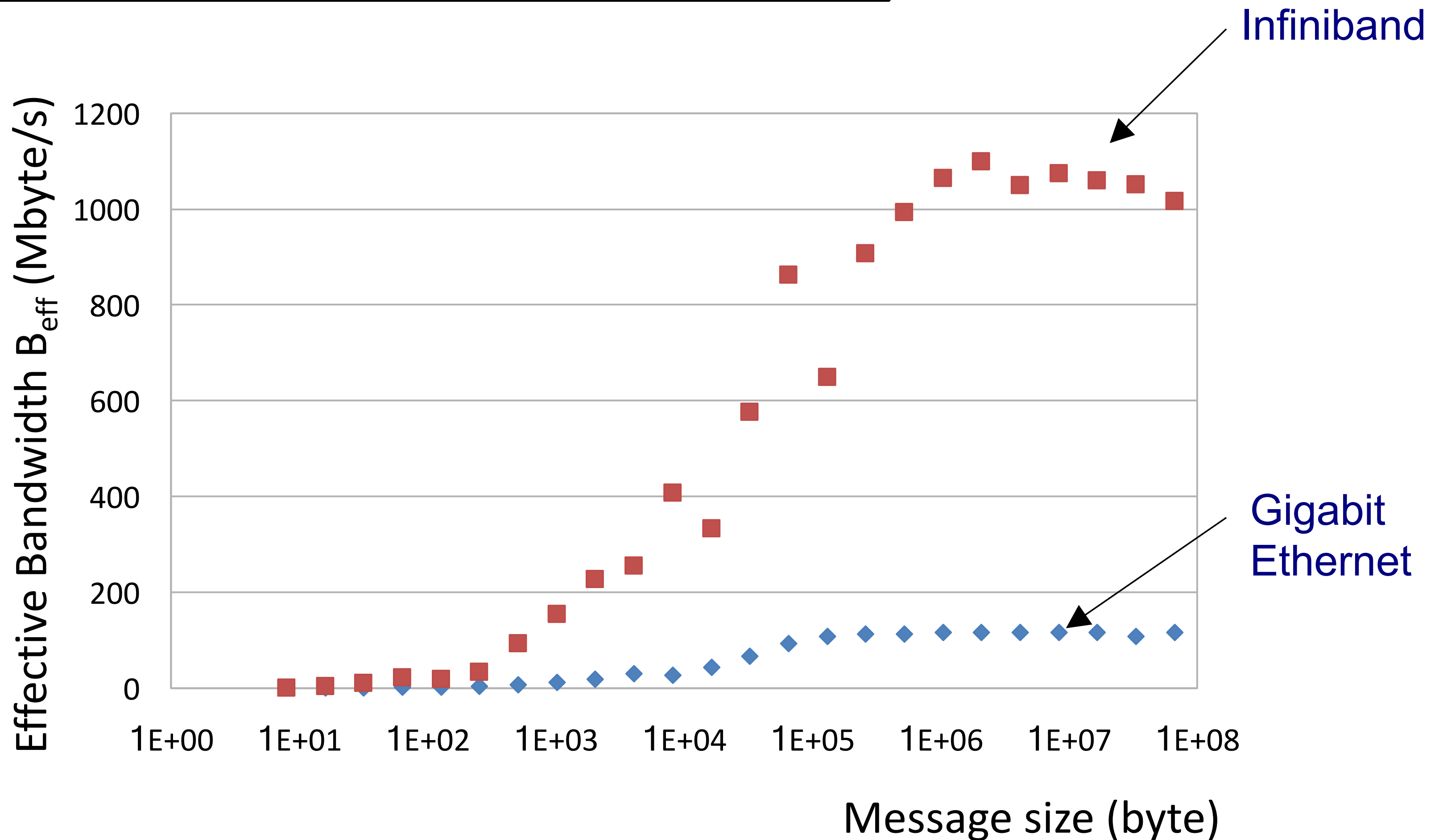
Basic MPI routines

Timing routines in MPI

- `double MPI_Wtime( void )`
  - returns the time in seconds relative to “some time” in the past
  - “some time” in the past is fixed during process
- `double MPI_Wtick( void )`
  - Returns the resolution of MPI_Wtime() in seconds
  - e.g. 10^{-3} = millisecond resolution

Bandwidth on Gengar (Ugent cluster)

Effective BW increases for larger messages



Benchmark results

Comparison of **CPU load**

Infiniband

```
top - 11:39:38 up 10 days, 18:25,  1 user,  load average: 0.02, 0.35, 0.32
Tasks: 187 total,   9 running, 177 sleeping,   0 stopped,   1 zombie
Cpu(s): 99.0%us,  1.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si  0.0%st
Mem:  16440052k total,  1684008k used, 14756044k free,   227224k buffers
Swap: 31357404k total,   563072k used, 30794332k free,   172864k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
15879 vsc400    20   0 65668  20m 3460 R 100.0  0.1   0:03.23 ringtest
15882 vsc400    20   0  128m  52m 3460 R 100.0  0.3   0:03.29 ringtest
15883 vsc400    18   0  128m  20m 3456 R 100.0  0.1   0:03.22 ringtest
15884 vsc400    18   0 65672  20m 3464 R 100.0  0.1   0:03.22 ringtest
15885 vsc400    19   0  128m  52m 3460 R 100.0  0.3   0:03.28 ringtest
15886 vsc400    20   0  192m  84m 3488 R 100.0  0.5   0:03.23 ringtest
15880 vsc400    20   0  191m  52m 3460 R 99.7  0.3   0:03.21 ringtest
15881 vsc400    20   0  128m  52m 3460 R 99.3  0.3   0:03.22 ringtest
```

Gigabit Ethernet

```
top - 11:33:13 up 10 days, 18:19,  1 user,  load average: 1.24, 0.30, 0.22
Tasks: 188 total,  10 running, 177 sleeping,   0 stopped,   1 zombie
Cpu(s): 19.8%us, 78.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  1.8%si,  0.0%st
Mem:  16440052k total,  1470984k used, 14969068k free,   227092k buffers
Swap: 31357404k total,   563096k used, 30794308k free,   168428k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14866 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.02 ringtest
14867 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.05 ringtest
14868 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.05 ringtest
14869 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.05 ringtest
14870 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.01 ringtest
14871 vsc400    25   0 89028  18m 1892 R 100.0  0.1   0:10.05 ringtest
14872 vsc400    25   0 89028  34m  17m R 99.7  0.2   0:10.00 ringtest
14865 vsc400    25   0 89028  18m 1892 S 99.3  0.1   0:09.89 ringtest
```

Exchanging messages in MPI

```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype
    sendtype, int dest, int sendtag, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int source,
    int recvtag, MPI_Comm comm, MPI_Status *status )
```

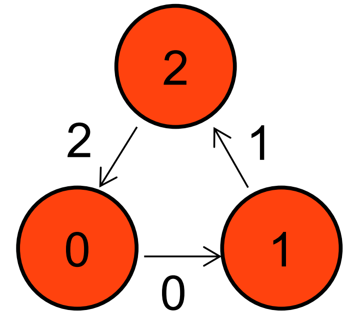
- **sendbuf:** pointer to the message to send
- **sendcount**: number of elements to transmit
- **sendtype:** datatype of the items to send
- **dest:** rank of destination process
- **sendtag**: identifier for the message
- **recvbuf:** pointer to the buffer to store the message (**disjoint** with sendbuf)
- **recvcount: upper bound (!)** to the number of elements to receive
- **recvtype:** datatype of the items to receive
- **source:** rank of the source process (or MPI_ANY_SOURCE)
- **recvtag**: value to identify the message (or MPI_ANY_TAG)
- **comm**: communicator specification (e.g. MPI_COMM_WORLD)
- **status**: structure that contains { MPI_SOURCE, MPI_TAG, MPI_ERROR }
- **sendbuf:** pointer to the buffer to send

```
int MPI_Sendrecv_replace( ... )
```

- Buffer is replace by received data

Basic MPI routines

Sendrecv example




```
const int len = 10000;
int a[len], b[len];
if ( myRank == 0 ) {
    MPI_Send( a, len, MPI_INT, 1, 0, MPI_COMM_WORLD );
    MPI_Recv( b, len, MPI_INT, 2, 2, MPI_COMM_WORLD, &status
    );
} else if ( myRank == 1 ) {
    MPI_Sendrecv( a, len, MPI_INT, 2, 1, b, len, MPI_INT, 0,
                  0, MPI_COMM_WORLD, &status );
} else if ( myRank == 2) {
    MPI_Sendrecv( a, len, MPI_INT, 0, 2, b, len, MPI_INT, 1,
                  1, MPI_COMM_WORLD, &status );
}
```

safe to exchange !

- Compatibility between Sendrecv and 'normal' send and recv
- Sendrecv can help to prevent deadlocks

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Non-blocking communication

Idea:

- Do something useful while waiting for communications to finish
- Try to overlap communications and computations

How?

- Replace blocking communication by non-blocking variants

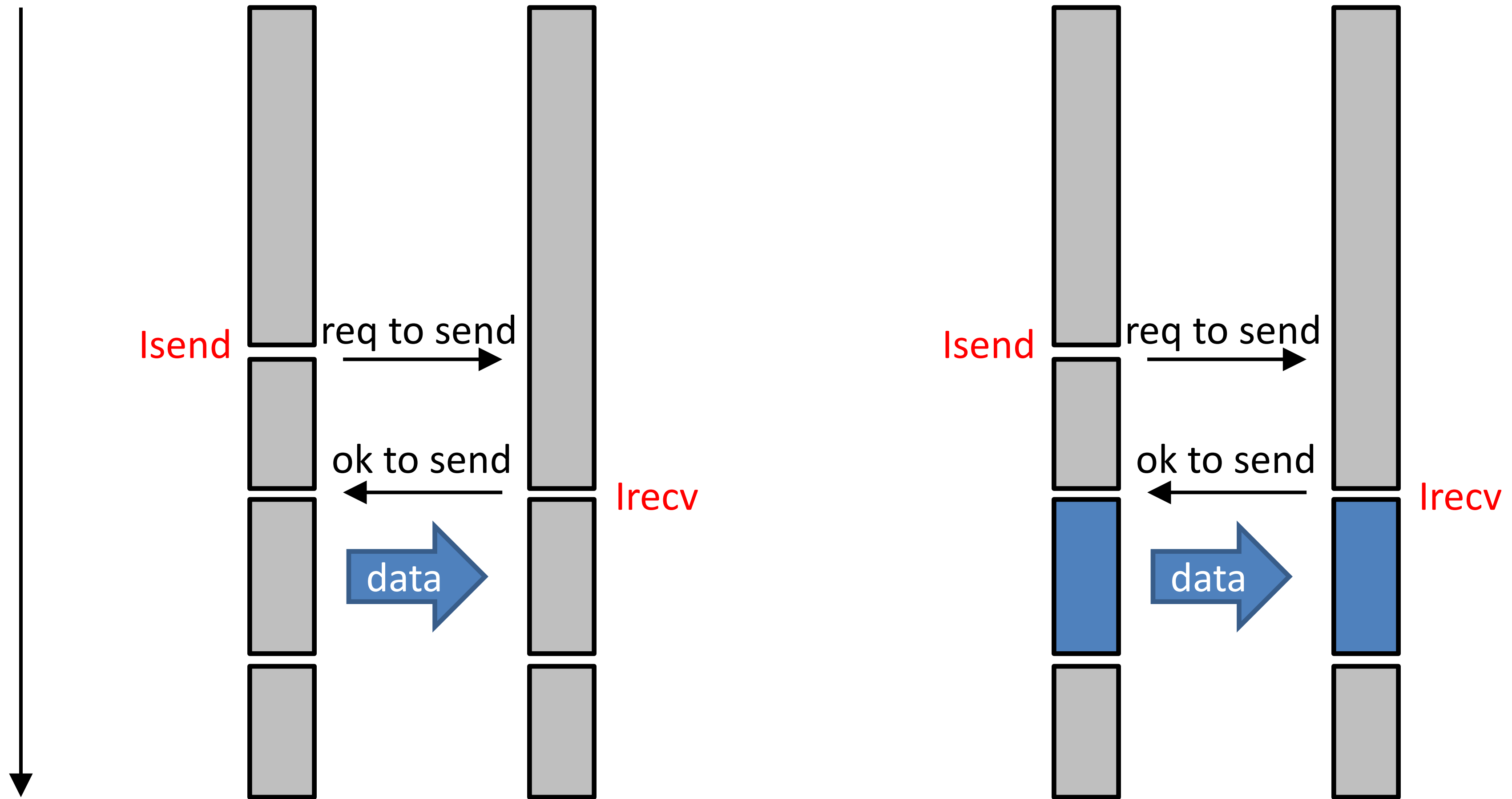
- I = intermediate functions
- MPI_Isend and MPI_Irecv routines return immediately
- Need polling routines to verify progress
  - **request** handle is used to identify communications
  - **status** field moved to polling routines (see further)

Non-blocking communications

Asynchronous progress

- = ability to progress communications while performing calculations
- Depends on hardware
  - Gigabit Ethernet = very limited
  - Infiniband = much more possibilities
- Depends on MPI implementation
  - Multithreaded implementations of MPI (e.g. Open MPI)
  - Daemon for asynchronous progress (e.g. LAM MPI)
- Depends on protocol
  - Eager protocol
  - Handshake protocol
- Still the subject of ongoing research

Non-blocking sending and receiving



time

a) network interface supports overlapping computations and communications

b) network interface has no such support

Slide reproduced from slides by John Mellor - Crummey

Non-blocking communications

Polling / waiting routines

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

`request:` handle to identify communication

`status:` status information (cfr. '*normal*' MPI_Recv)

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )
```

Returns immediately. Sets `flag = true` if communication has completed

```
int MPI_Waitany( int count, MPI_Request *array_of_requests,
                 int *index, MPI_Status *status )
```

Waits for exactly one communication to complete

If more than one communication has completed, it picks a random one

`index` returns the index of completed communication

```
int MPI_Testany( int count, MPI_Request *array_of_requests,
                 int *index, int *flag, MPI_Status *status )
```

Returns immediately. Sets `flag = true` if at least one communication completed

If more than one communication has completed, it picks a random one

`index` returns the index of completed communication

If `flag = false`, `index` returns MPI_UNDEFINED

Example: client-server code

```
if ( rank != 0 ) {       // client code
    while ( true ) {   // generate requests and send to the server
        generate_request( data, &size );
        MPI_Send( data, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD );
    }
} else {                 // server code (rank == 0)
    MPI_Request *reqList = new MPI_Request[nProc];
    for ( int i = 0; i < nProc - 1; i++ )
        MPI_Irecv( buffer[i].data, MAX_LEN, MPI_CHAR, i+1, tag,
                   MPI_COMM_WORLD, &reqList[i] );
    while ( true ) {             // main consumer loop
        MPI_Status status;
        int reqIndex, recvSize;

        MPI_Waitany( nProc-1, reqList, &reqIndex, &status );
        MPI_Get_count ( &status, MPI_CHAR, &recvSize );
        do_service( buffer[reqIndex].data, recvSize );
        MPI_Irecv( buffer[reqIndex].data, MAX_LEN, MPI_CHAR,
                   status.MPI_SOURCE, tag, MPI_COMM_WORLD,
                   &reqList[reqIndex] );
    }
}
```

Non-blocking communications

Polling / waiting routines (cont'd)

```
int MPI_Waitall( int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses )
```

Waits for **all** communications to complete

```
int MPI_Testall ( int count, MPI_Request *array_of_requests,
                  int *flag, MPI_Status *array_of_statuses )
```

Returns immediately. Sets `flag = true` if all communications have completed

```
int MPI_Waitsome ( int incount, MPI_Request * array_of_requests,
                   int *outcount, int *array_of_indices,
                   MPI_Status *array_of_statuses )
```

Waits for **at least one** communications to complete

`outcount` contains the number of communications that have completed

Completed requests are set to `MPI_REQUEST_NULL`

```
int MPI_Testsome ( int incount, MPI_Request * array_of_requests,
                   int *outcount, int *array_of_indices,
                   MPI_Status *array_of_statuses )
```

Same as Waitsome, but returns immediately.

flag field no longer needed, returns **outcount = 0** if no completed communications

Example: improved client-server code

```
if ( rank != 0 ) {       // same client code
    ...
} else {                 // server code (rank == 0)
    MPI_Request *reqList = new MPI_Request[nProc-1];
    MPI_Status *status = new MPI_Status[nProc-1];
    int *reqIndex = new MPI_Request[nProc];

    for ( int i = 0; i < nProc - 1; i++ )
        MPI_Irecv( buffer[i].data, MAX_LEN, MPI_CHAR, i+1, tag,
                   MPI_COMM_WORLD, &reqList[i] );
    while ( true ) {                // main consumer loop
        int numMsg;
        MPI_Waitsome( nProc-1, reqList, &numMsg, reqIndex, status );
        for ( int i = 0; i < numMsg; i++ ) {
            MPI_Get_count ( &status[i], MPI_CHAR, &recvSize );
            do_service( buffer[reqIndex[i]].data, recvSize);
            MPI_Irecv( buffer[reqIndex[i]].data, MAX_SIZE, MPI_CHAR,
                       status[i].MPI_SOURCE, tag, MPI_COMM_WORLD,
                       &reqList[reqIndex[i]] );
        }
    }
}
```

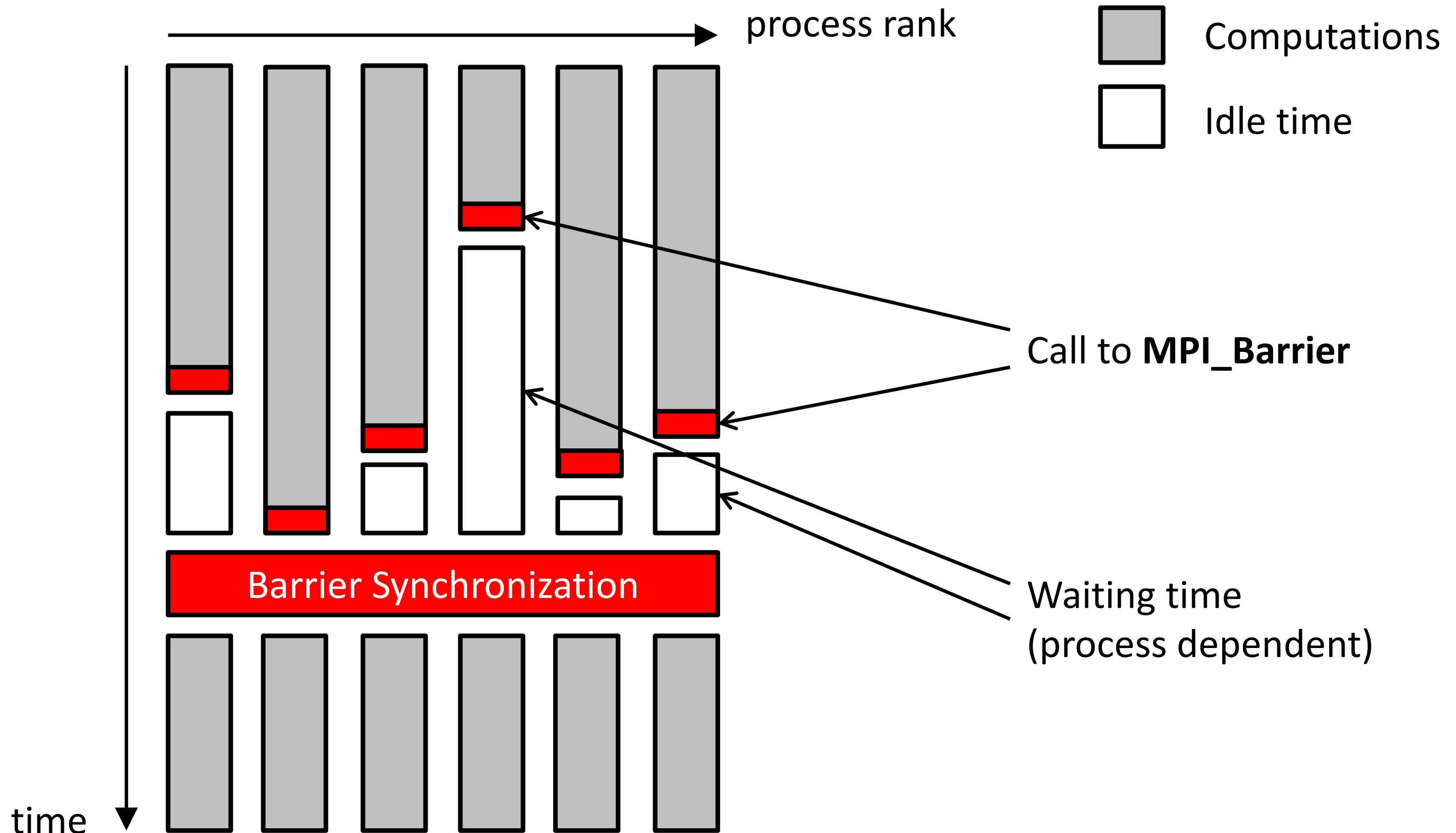
Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Barrier synchronization

`MPI_Barrier(MPI_Comm comm)`

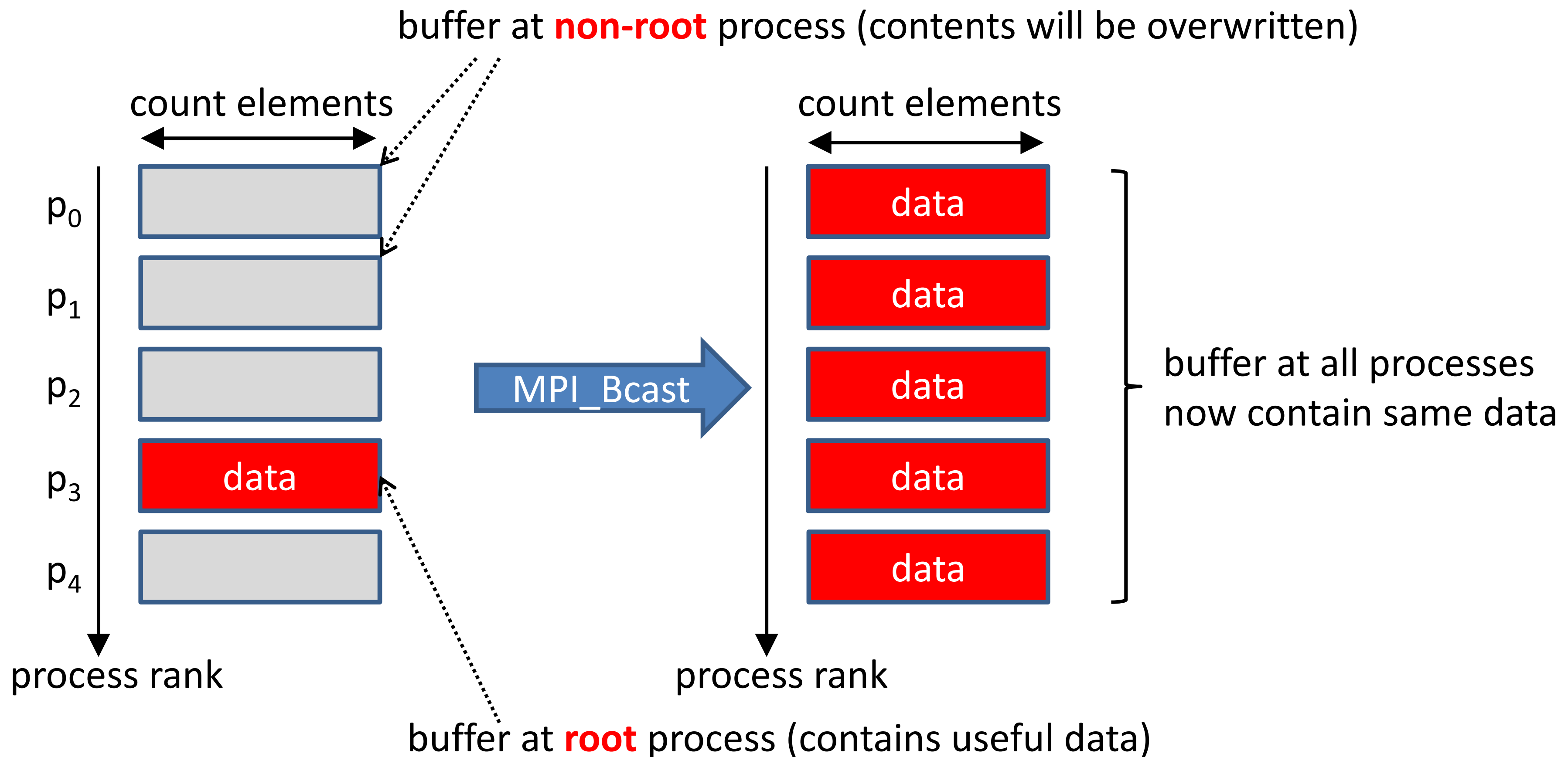
This function does not return until all processes in comm have called it.



Broadcast

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
          int root, MPI_Comm comm)
```

MPI_Bcast broadcasts `count` elements of type `datatype` stored in `buffer` at the `root` process to all other processes in `comm` where this data is stored in `buffer`.



Broadcast example

```
...
int rank, size;

...   // init MPI and rank and size variables

int root = 0;
char buffer[12];

if (rank == root)
        sprintf(buffer, "Hello world");

MPI_Bcast(buffer, 12, MPI_CHAR, root, MPI_COMM_WORLD);

printf("Process %d has %s stored in the buffer.\n", buffer, rank);

...
```

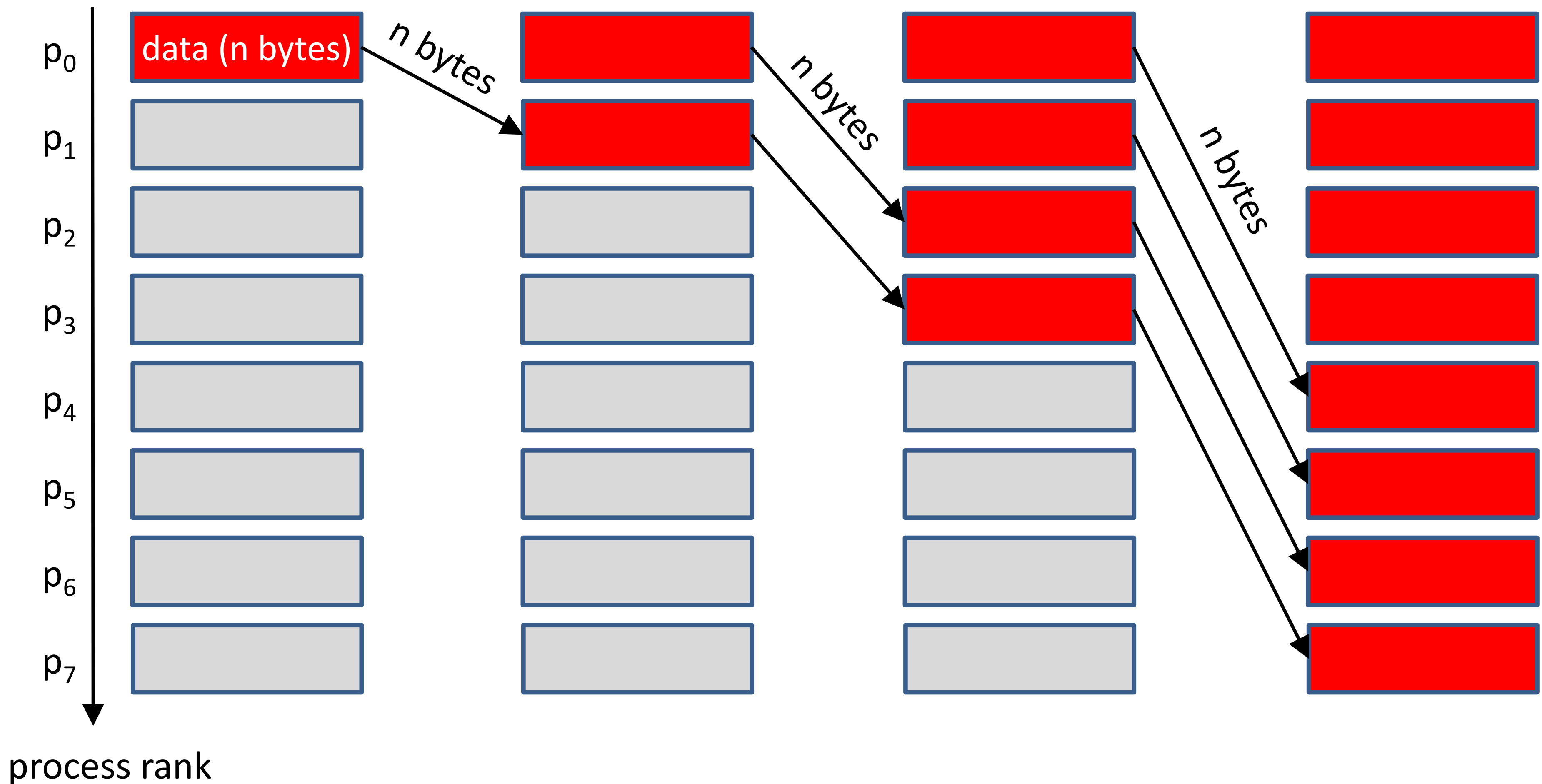
fill the buffer at the root process only

all processes must call MPI_Bcast

```
john@doe ~]$ mpirun -np 4 ./broadcast
Process 1 has Hello World stored in the buffer.
Process 0 has Hello World stored in the buffer.
Process 3 has Hello World stored in the buffer.
Process 2 has Hello World stored in the buffer.
```

Broadcast algorithm

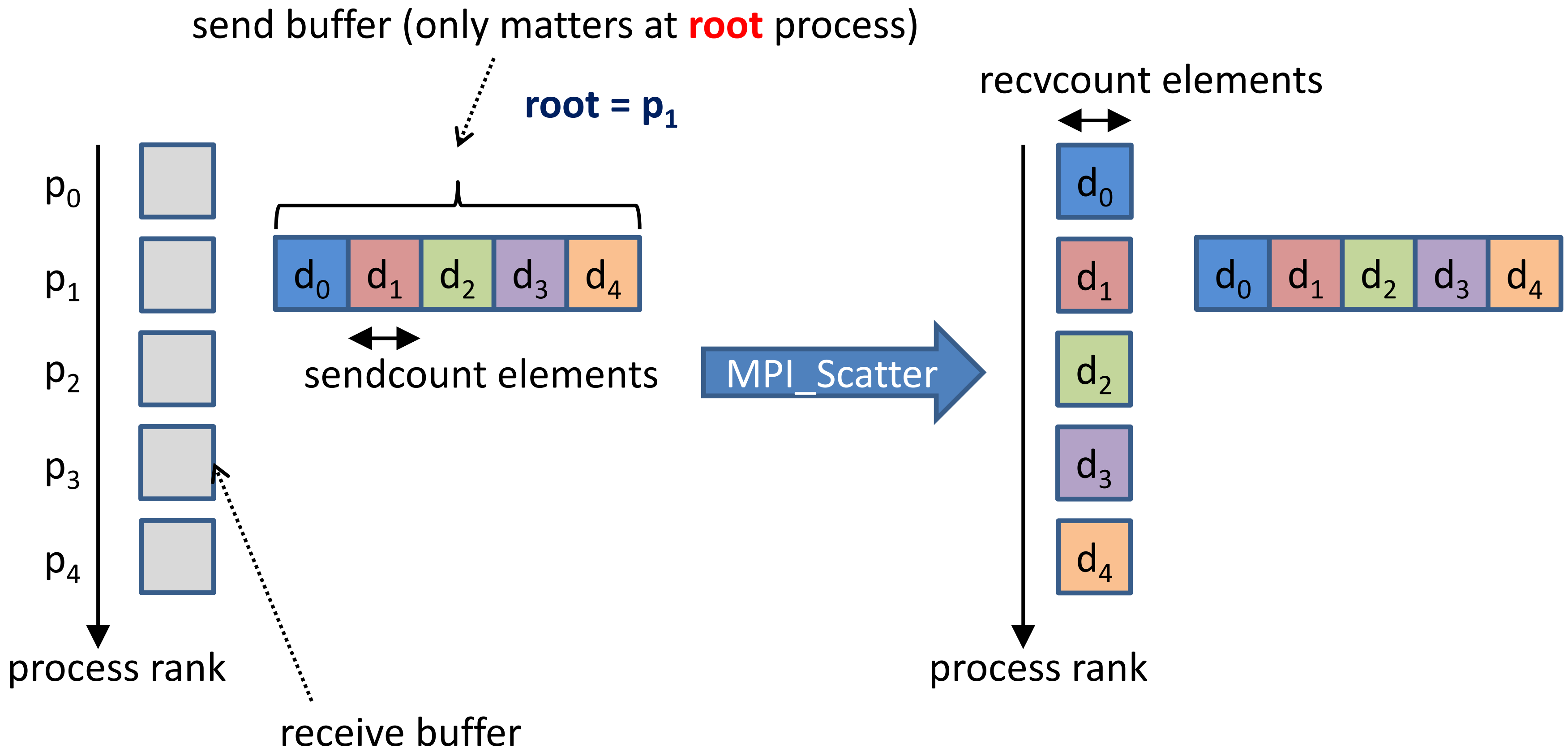
- **Linear algorithm**, subsequent sending of n bytes from root process to P-1 other processes takes $(\alpha + \beta n)(P-1)$ time.
- **Binary tree algorithm** takes only $(\alpha + \beta n)\lceil \log_2 P \rceil$ time.



Scatter

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendType,
            void *recvbuf, int recvcount, MPI_Datatype recvType,
            int root, MPI_Comm comm)
```

MPI_Scatter partitions a `sendbuf` at the `root` process into P equal parts of size `sendcount` and sends each process in `comm` (including root) a portion in rank order.



Scatter example

```
int root = 0;
char recvBuf[7];

if (rank == root) {
    char sendBuf[25];
    sprintf(sendBuf, "This is the source data.");

    MPI_Scatter(sendBuf, 6, MPI_CHAR, recvBuf, 6, MPI_CHAR,
                root, MPI_COMM_WORLD);
} else {
    MPI_Scatter(NULL, 0, MPI_CHAR, recvBuf, 6, MPI_CHAR,
                root, MPI_COMM_WORLD);
}

recvBuf[6] = '\0';
printf("Process %d has %s in receive buffer\n", rank, recvBuf);

...
```

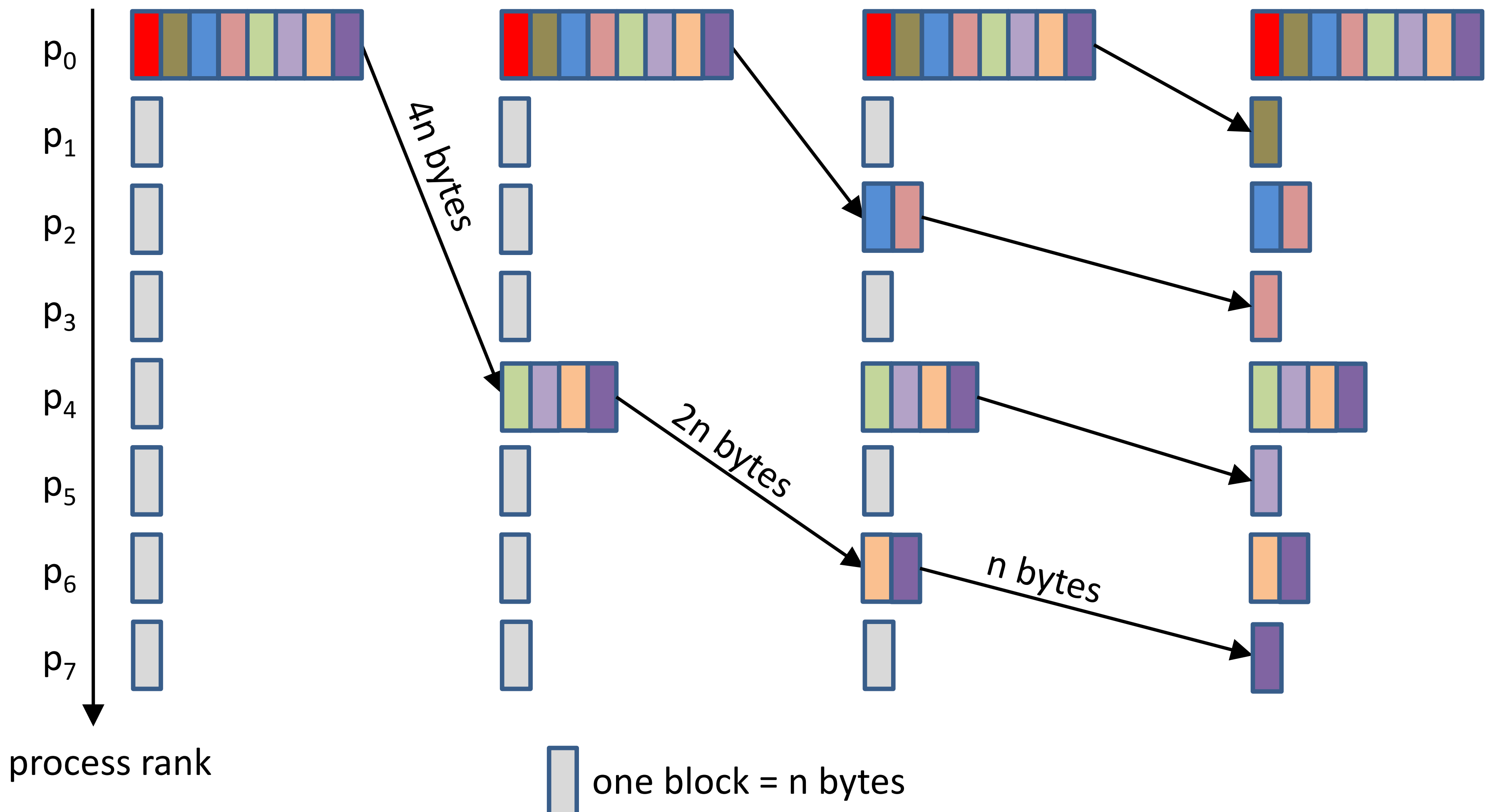
fill the send buffer at the root process only

first three parameters are ignored on non-root processes

```
john@doe ~]$ mpirun -np 4 ./scatter
Process 1 has s the  stored in the buffer.
Process 0 has This i stored in the buffer.
Process 3 has  data. stored in the buffer.
Process 2 has source stored in the buffer.
```

Scatter algorithm

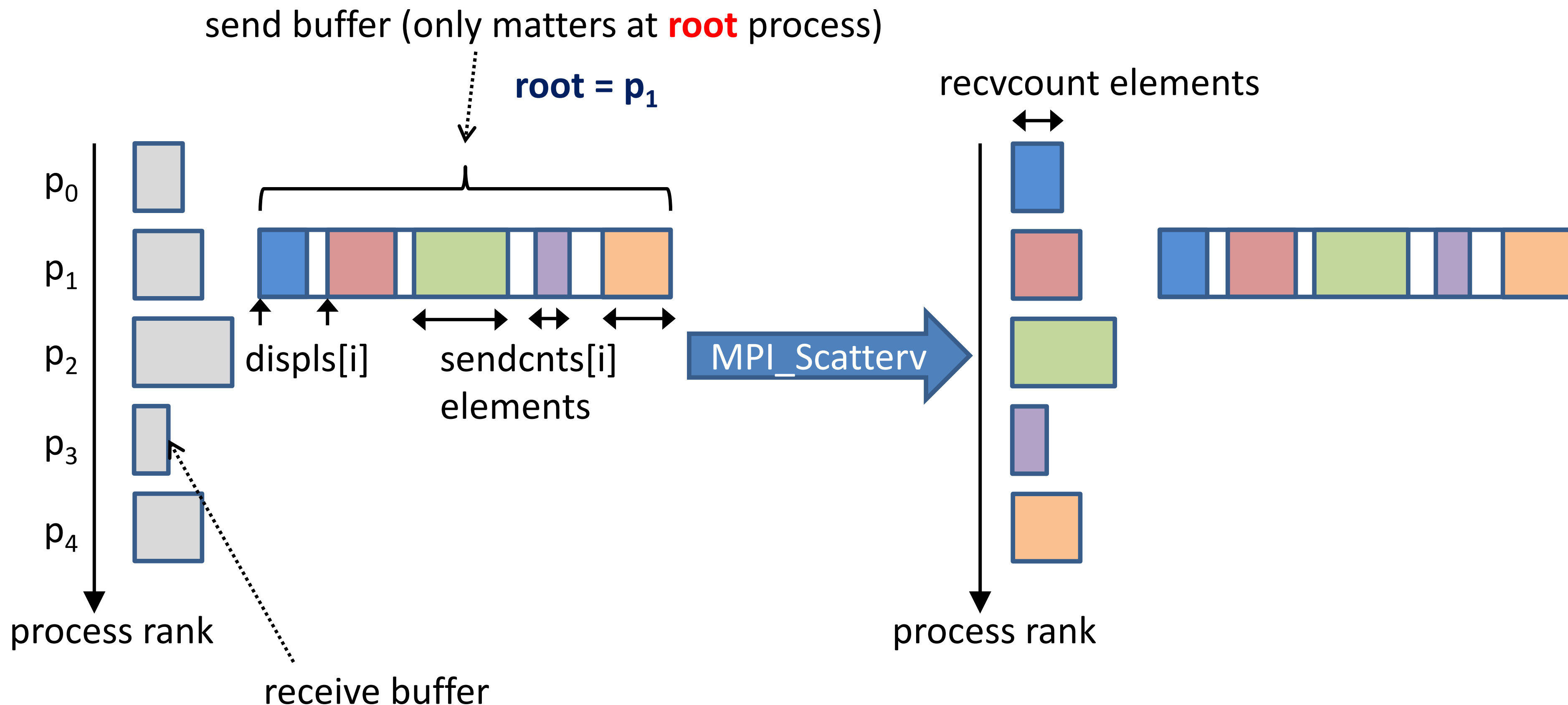
- **Linear algorithm**, subsequent sending of n bytes from root process to P-1 other processes takes $(\alpha + \beta n)(P-1)$ time.
- **Binary algorithm** takes only $\alpha \lceil \log_2 P \rceil + \beta n(P-1)$ time (reduced number of communication rounds!)



Scatter (vector variant)

```
MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs,
             MPI_Datatype sendType, void *recvbuf, int recvcnt,
             MPI_Datatype recvType, int root, MPI_Comm comm)
```

Partitions of `sendbuf` don't need to be of equal size and are specified per receiving process: the first index by the `displs` array, their size by the `sendcnts` array.

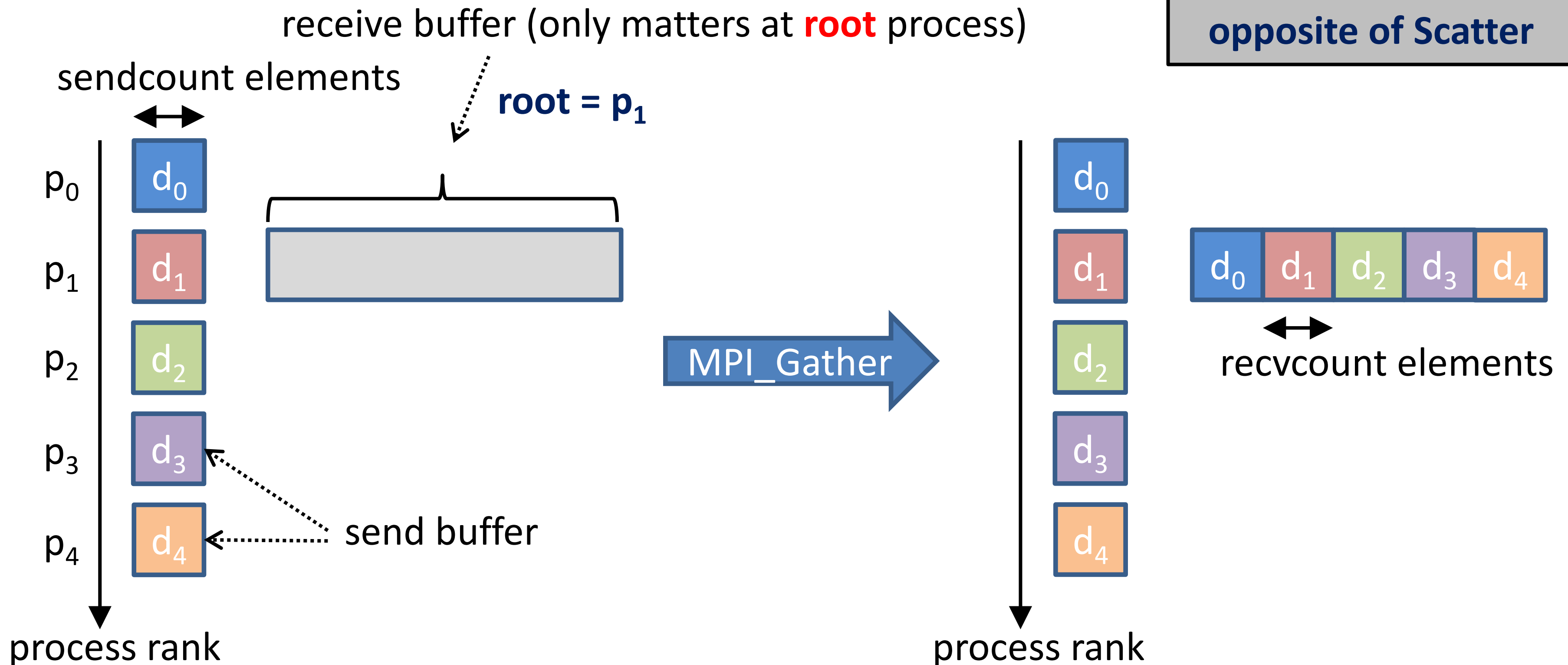


Gather

```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendType,
           void *recvbuf, int recvcount, MPI_Datatype recvType,
           int root, MPI_Comm comm)
```

MPI_Gather gathers equal partitions of size `recvcount` from each of the P processes in `comm` (including root) and stores them in `recvbuf` at the `root` process in rank order.

Gather is the opposite of Scatter



A vector variant, **MPI_Gatherv**, exists, a similar generalization as MPI_Scatterv

Gather example

```
int root = 0;
int sendBuf = rank;

if (rank == root) {
    int *recvBuf = new int[size];

    MPI_Gather(&sendBuf, 1, MPI_INT, recvBuf, 1, MPI_INT,
               root, MPI_COMM_WORLD);
    cout << "Receive buffer at root process: " << endl;
    for (size_t i = 0; i < size; i++)
        cout << recvBuf[i] << " ";
    cout << endl;

    delete [] recvBuf;
} else {
    MPI_Gather(&sendBuf, 1, MPI_INT, NULL, 1, MPI_INT,
               root, MPI_COMM_WORLD);
}
```

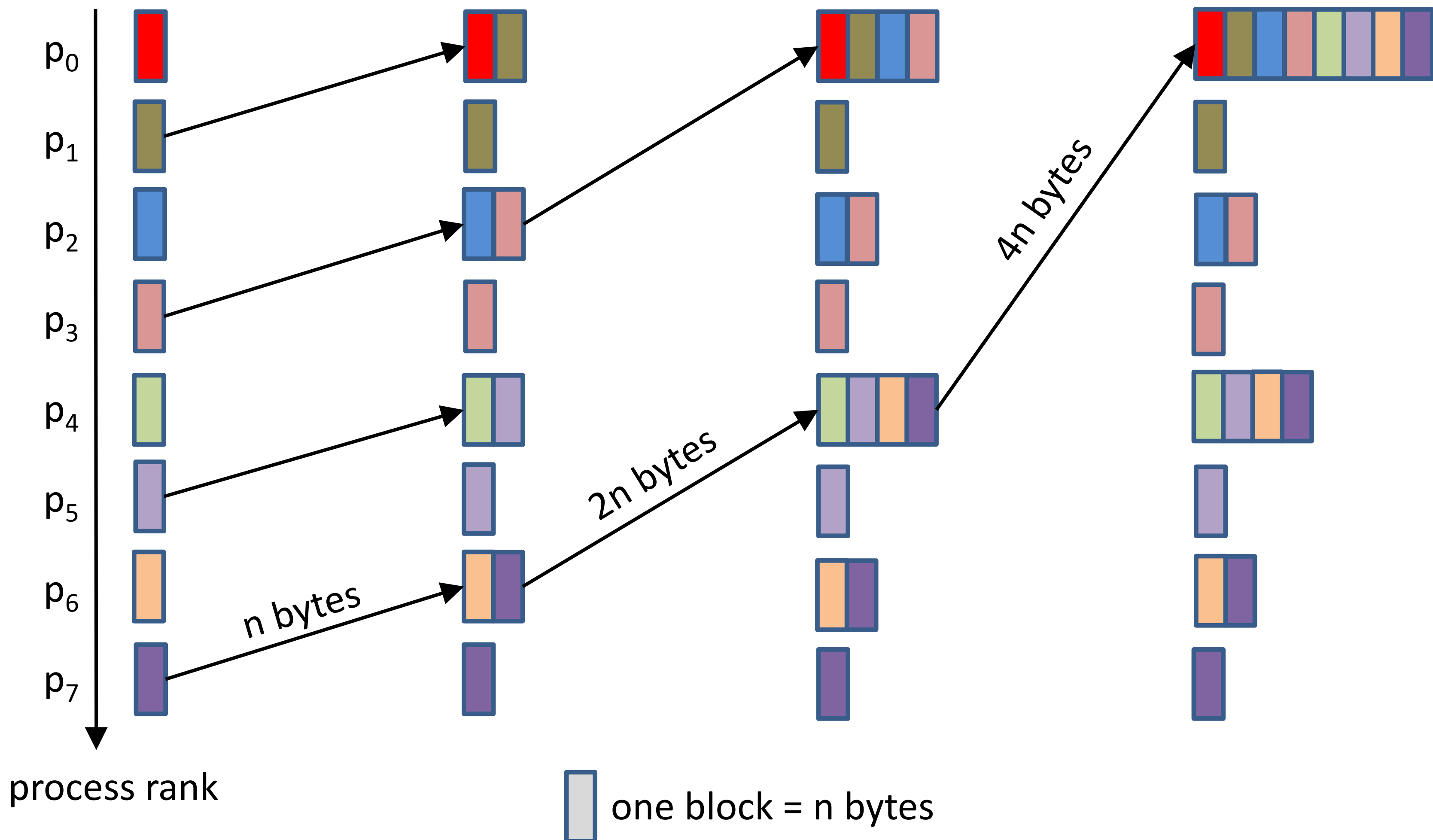
receive buffer exists at the root process only

receive parameters are ignored on non-root processes

```
john@doe ~]$ mpirun -np 4 ./gather
Receive buffer at root process:
0 1 2 3
```

Gather algorithm

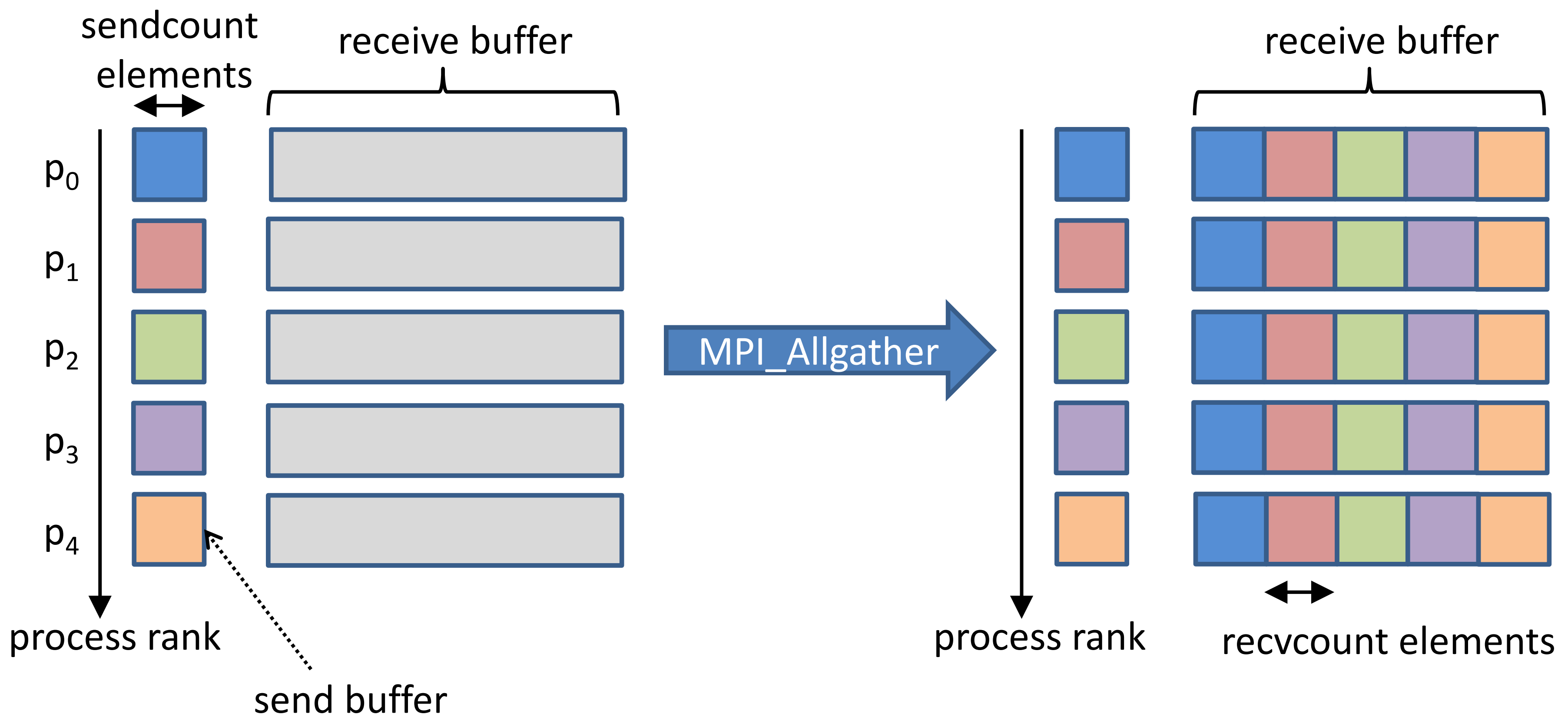
- **Linear algorithm**, subsequent sending of n bytes from P-1 processes to root takes $(\alpha + \beta n)(P-1)$ time.
- **Binary algorithm** takes only $\alpha\lceil \log_2 P \rceil + \beta n(P-1)$ time (reduced number of communication rounds!)



AllGather

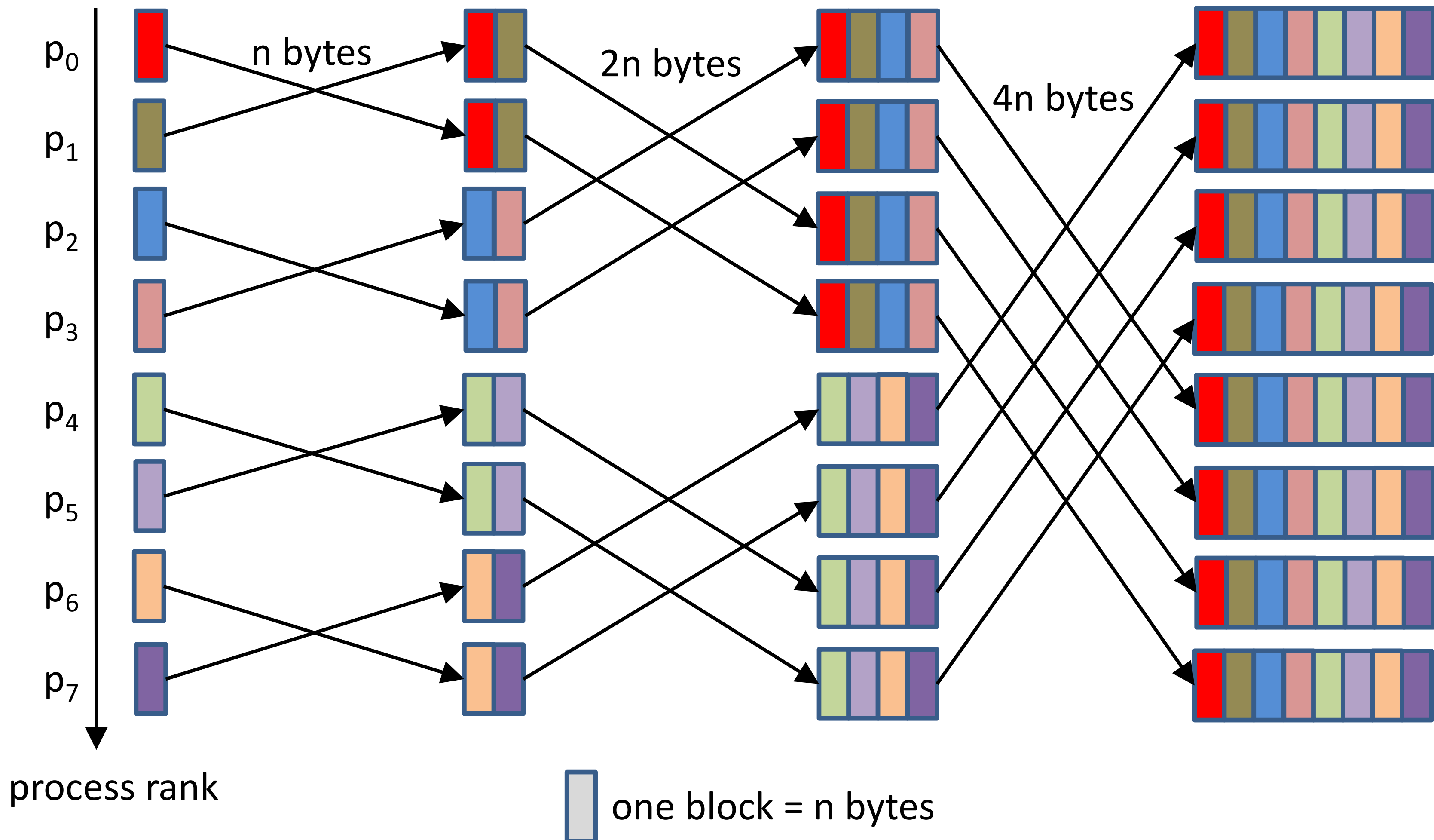
```
MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendType,
              void *recvbuf, int recvcnt, MPI_Datatype recvType,
              MPI_Comm comm)
```

MPI_Allgather is a generalization of MPI_Gather, in that sense that the data is gathered by all processed, instead of just the root process.



Allgather algorithm

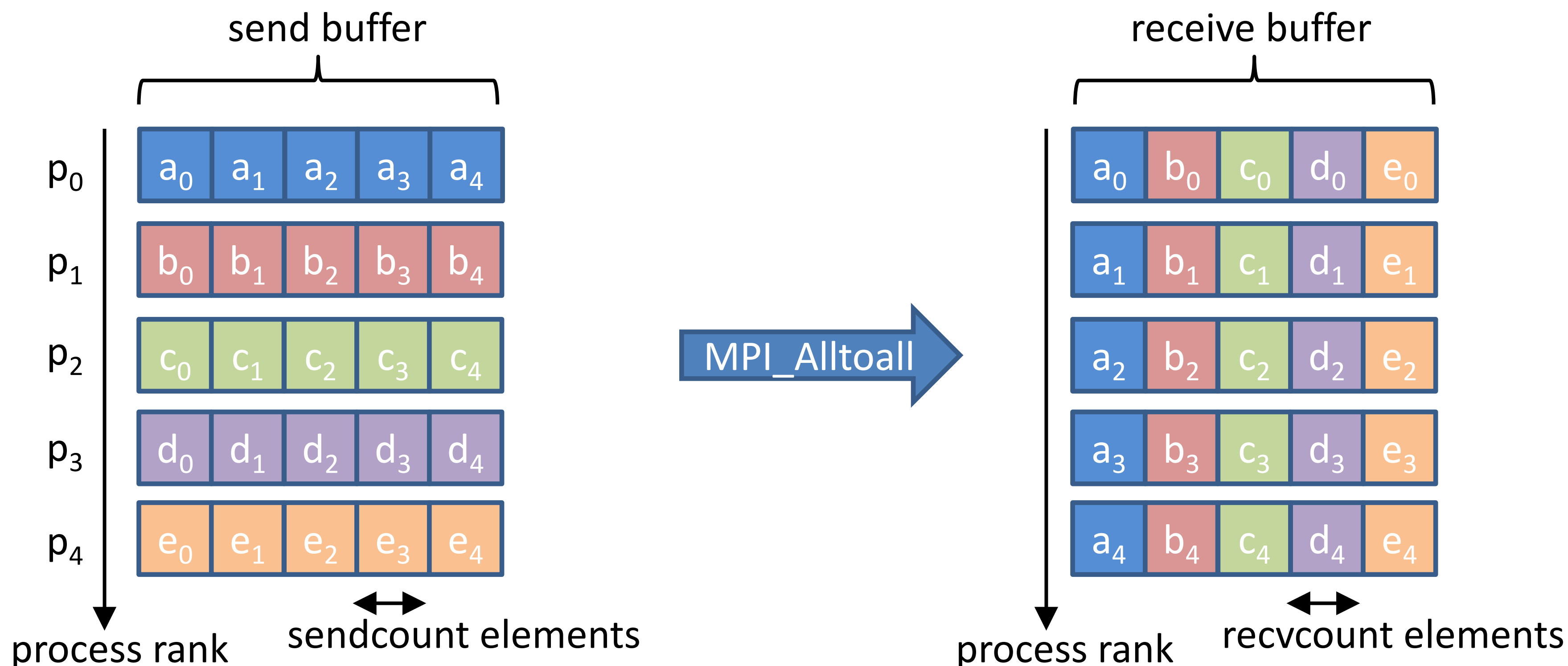
- **P calls to gather** takes $P[\alpha\lceil\log_2 P\rceil + \beta n(P-1)]$ time (using the best gather algorithm)
- **Gather followed by broadcast** takes $2\alpha\lceil\log_2 P\rceil + \beta n(P\lceil\log_2 P\rceil + P-1)$ time.
- **"Butterfly" algorithm** takes only $\alpha \log_2 P + \beta n(P-1)$ time (in case P is a power of two)



All to all communication

```
MPI_Alltoall(void *sendbuf, int sendcnt, MPI_Datatype sendType,
             void *recvbuf, int recvcnt, MPI_Datatype recvType,
             int root, MPI_Comm comm)
```

Using MPI_Alltoall, every process sends a distinct message to every other process.

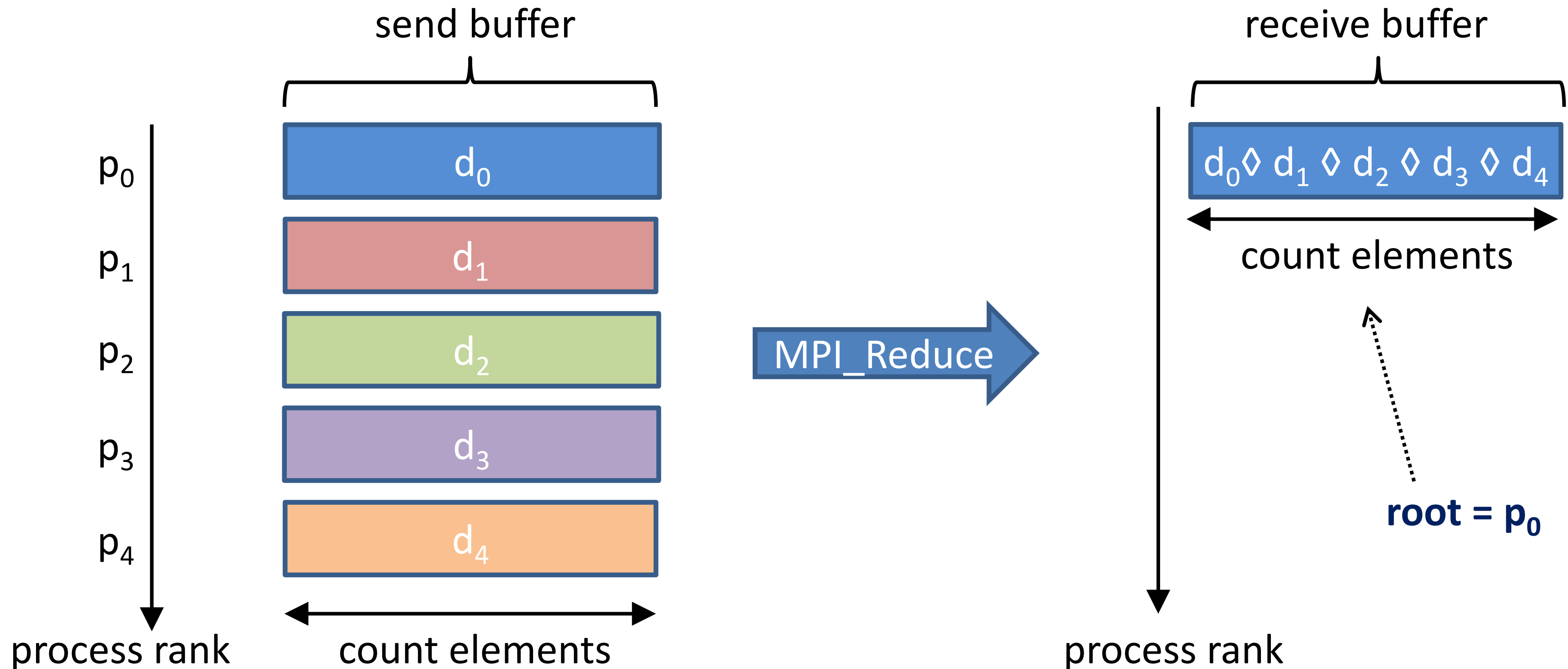


A vector variant, **MPI_Alltoallv**, exists, allowing for different sizes for each process

Reduce

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
        dataType, MPI_Op op, int root, MPI_Comm comm)
```

The reduce operation aggregates ("reduces") scattered data at the root process



◊ = operation, like sum, product, maximum, etc.

Reduce operations

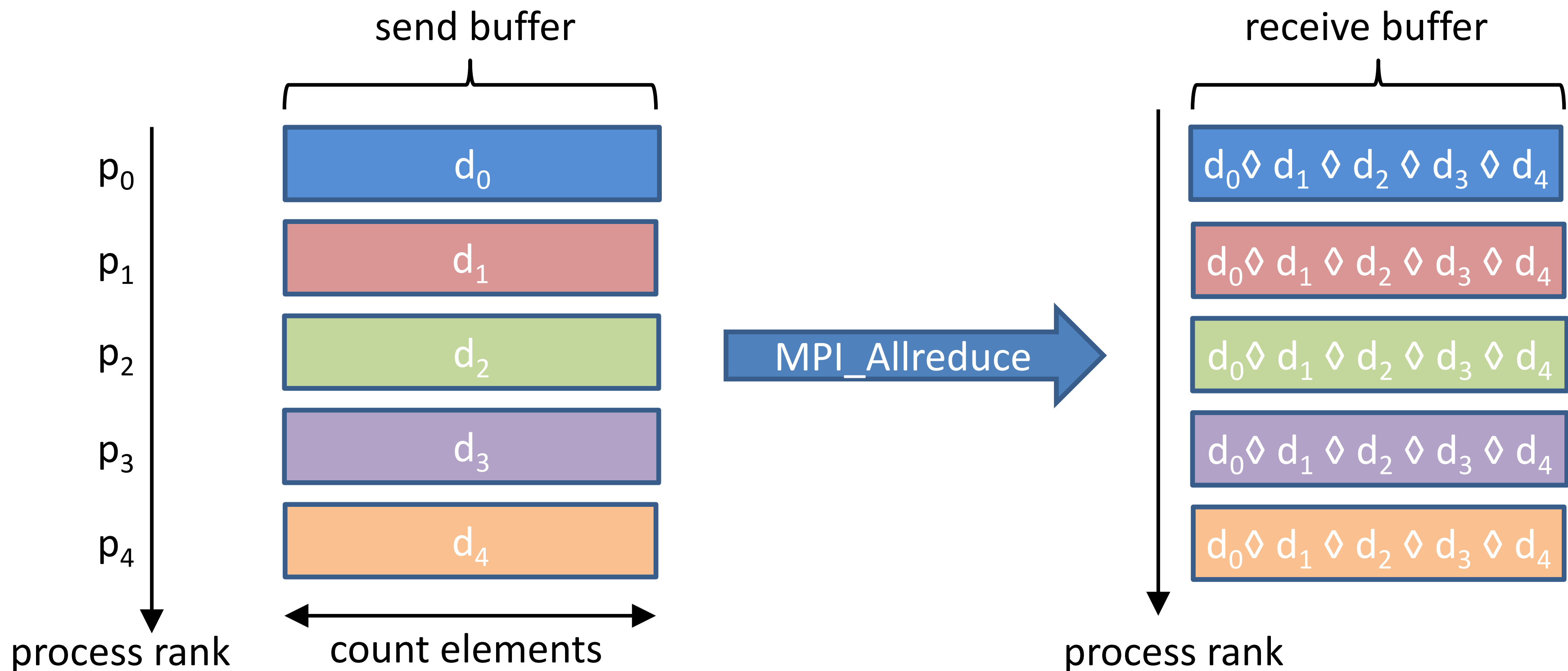
Available reduce operations (associative and commutative)
User defined operations are also possible

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bitwise AND
MPI_LOR	logical OR
MPI_BOR	bitwise OR
MPI_LXOR	logical exclusive OR
MPI_BXOR	bitwise exclusive OR
MPI_MAXLOC	maximum and its location
MPI_MINLOC	maximum and its location

Allreduce operation

```
MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
              MPI_Datatype dataType, MPI_Op op, MPI_Comm comm)
```

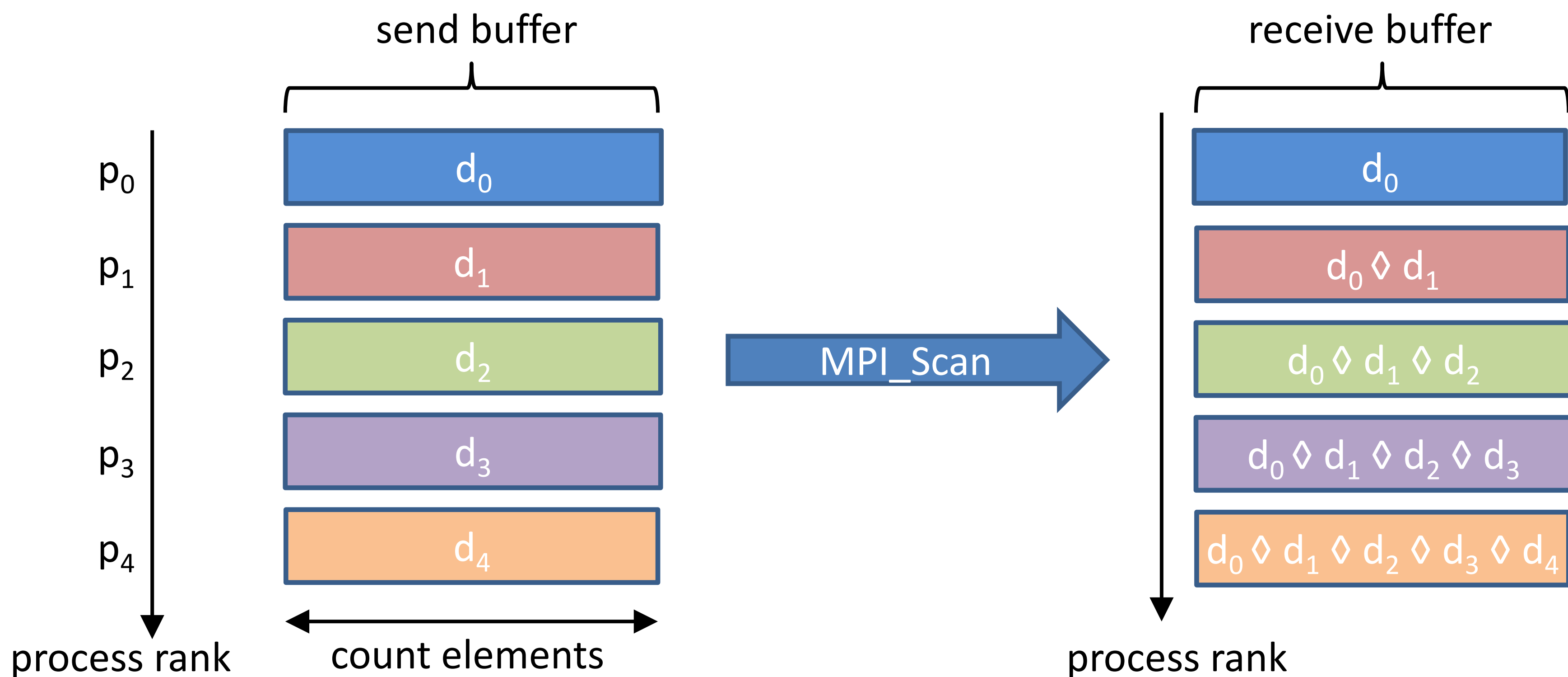
Similar to the reduce operation, but the result is available on every process.



Scan operation

```
MPI_Scan(void *sendbuf, void *recvbuf, int count,
         MPI_Datatype dataType, MPI_Op op, MPI_Comm comm)
```

A scan performs a partial reduction of data, every process has a distinct result



Hands-on

Matrix-vector multiplication

Master : Coordinates the work of others

Slave : does a bit of work

Task : compute A . b

A : double precision (m x n) matrix

b : double precision (n x 1) column matrix

Master algorithm

```
1. Broadcast b to each slave
2. Send 1 row of A to each slave
3. while ( not all m results received ) {
     Receive result from any slave s
     if ( not all m rows sent )
          Send new row to slave s
     else
          Send termination message to s
   }
4. continue
```

Slave algorithm

```
1. Broadcast b (in fact receive b)
2. do {
     Receive message m
      if( m != termination )
          compute result
          send result to master
  } while( m != termination )
3. slave terminates
```

Matrix-vector multiplication

```
int main( int argc, char** argv ) {
    int rows = 100, cols = 100;      // dimensions of a
    double **a;
    double *b, *c;
    int master = 0;                  // rank of master
    int myid;                        // rank of this process
    int numprocs;                    // number of processes
    // allocate memory for a, b and c
    a = (double**)malloc(rows * sizeof(double*));
    for( int i = 0; i < rows; i++ )
        a[i]=(double*)malloc(cols * sizeof(double));
    b = (double*)malloc(cols * sizeof(double));
    c = (double*)malloc(rows * sizeof(double));
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myid );
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs );
    if( myid == master )
        // execute master code
    else
        // execute slave code
    MPI_Finalize();
}
```

Matrix vector multiplication

```
// initialize a and b
for(int j=0;j<cols;j++) {b[j]=1.0; for(int i=0;i<rows;i++) a[i][j]=i;}
// broadcast b to each slave
MPI_Bcast( b, cols, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD );
// send row of a to each slave, tag = row number
int numsent = 0;
for( int i = 0; (i < numprocs-1) && (i < rows); i++ ) {
     MPI_Send(a[i], cols, MPI_DOUBLE_PRECISION, i+1,i,MPI_COMM_WORLD);
     numsent++;
}
for( int i = 0; i < rows; i++ ) {
    MPI_Status status; double ans; int sender;
    MPI_Recv( &ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    c[status.MPI_TAG] = ans;
    sender = status.MPI_SOURCE;
    if ( numsent < rows ) { // send more work if any
        MPI_Send( a[numsent], cols, MPI_DOUBLE_PRECISION,
                  sender, numsent, MPI_COMM_WORLD );
        numsent++;
    } else  // send termination message
        MPI_Send( MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION, sender,
                  rows, MPI_COMM_WORLD );
}
```

Matrix-vector multiplication

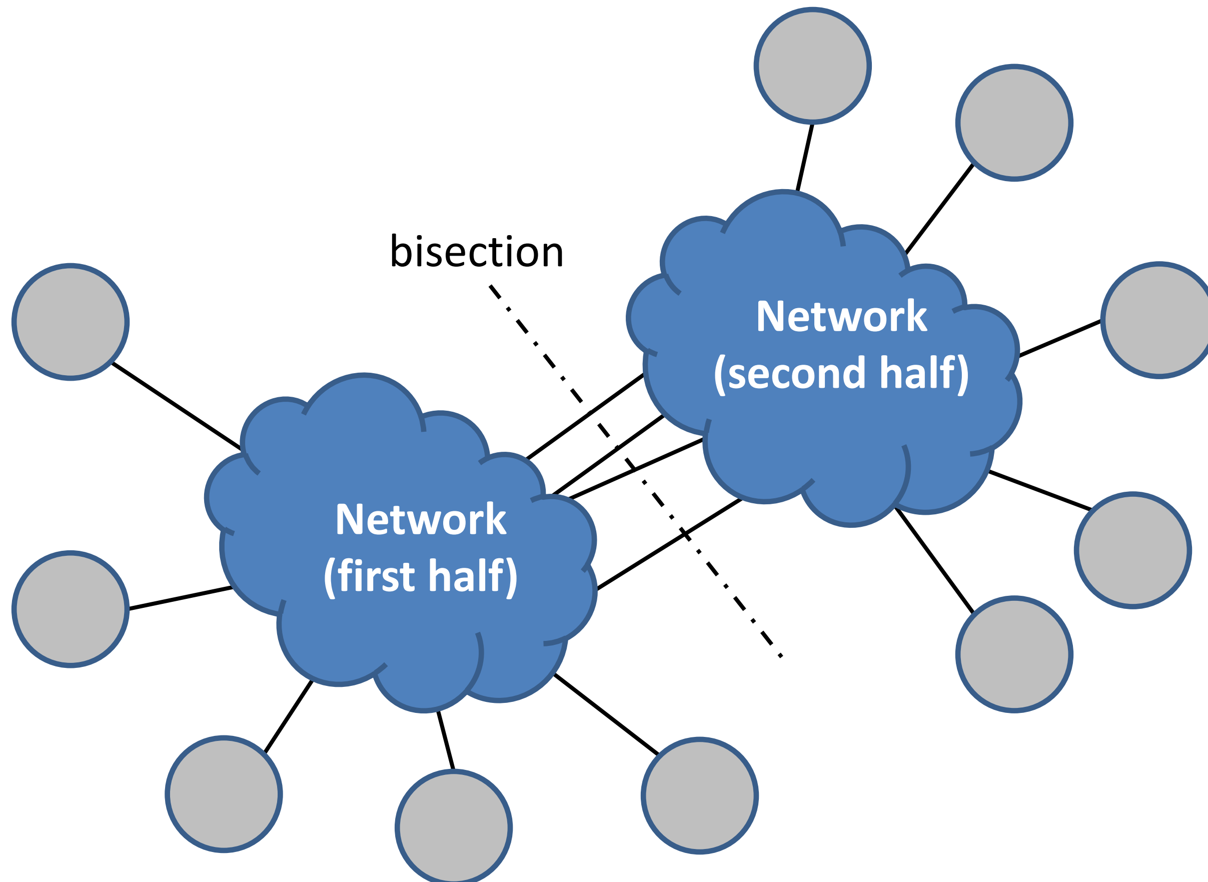
```
// broadcast b to each slave (receive here)
MPI_Bcast( b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD );
// send row of a to each slave, tag = row number
if( myid <= rows ) {
    double* buffer=(double*)malloc(cols*sizeof(double));
    while (true) {
        MPI_Status status;
        MPI_Recv( buffer, cols, MPI_DOUBLE_PRECISION, master,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status );
        if( status.MPI_TAG != rows ) { // not a termination message
            double ans = 0.0;
            for(int i=0; i < cols; i++)
                ans += buffer[i]*b[i];
            MPI_Send( &ans, 1, MPI_DOUBLE_PRECISION, master,
                      status.MPI_TAG, MPI_COMM_WORLD );
        } else
            break;
    }
} // more processes than rows => no work for some nodes
```

Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl’s law
  - Gustafson’s law
- Parallel program development: case studies

Network cost modeling

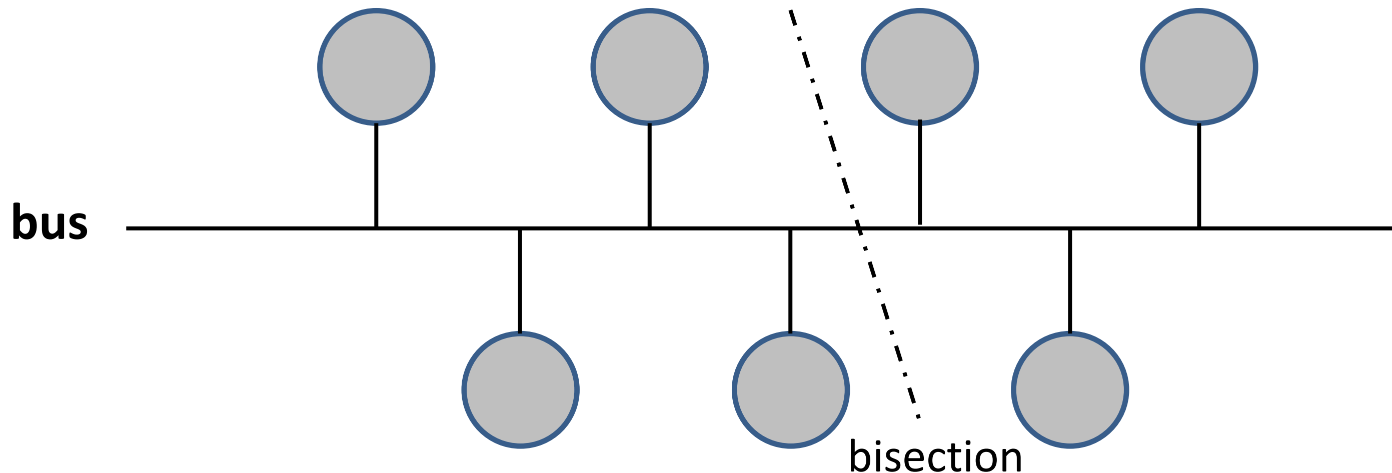
- **Simple performance model** for **multiple** communications
  - **Bisection bandwidth**: sum of the bandwidths of the (average number of) links to cut to partition the network in two halves.
  - **Diameter**: maximum number of hops to connect any two devices



Bus topology

Bus = communication channel that is **shared** by all connected devices

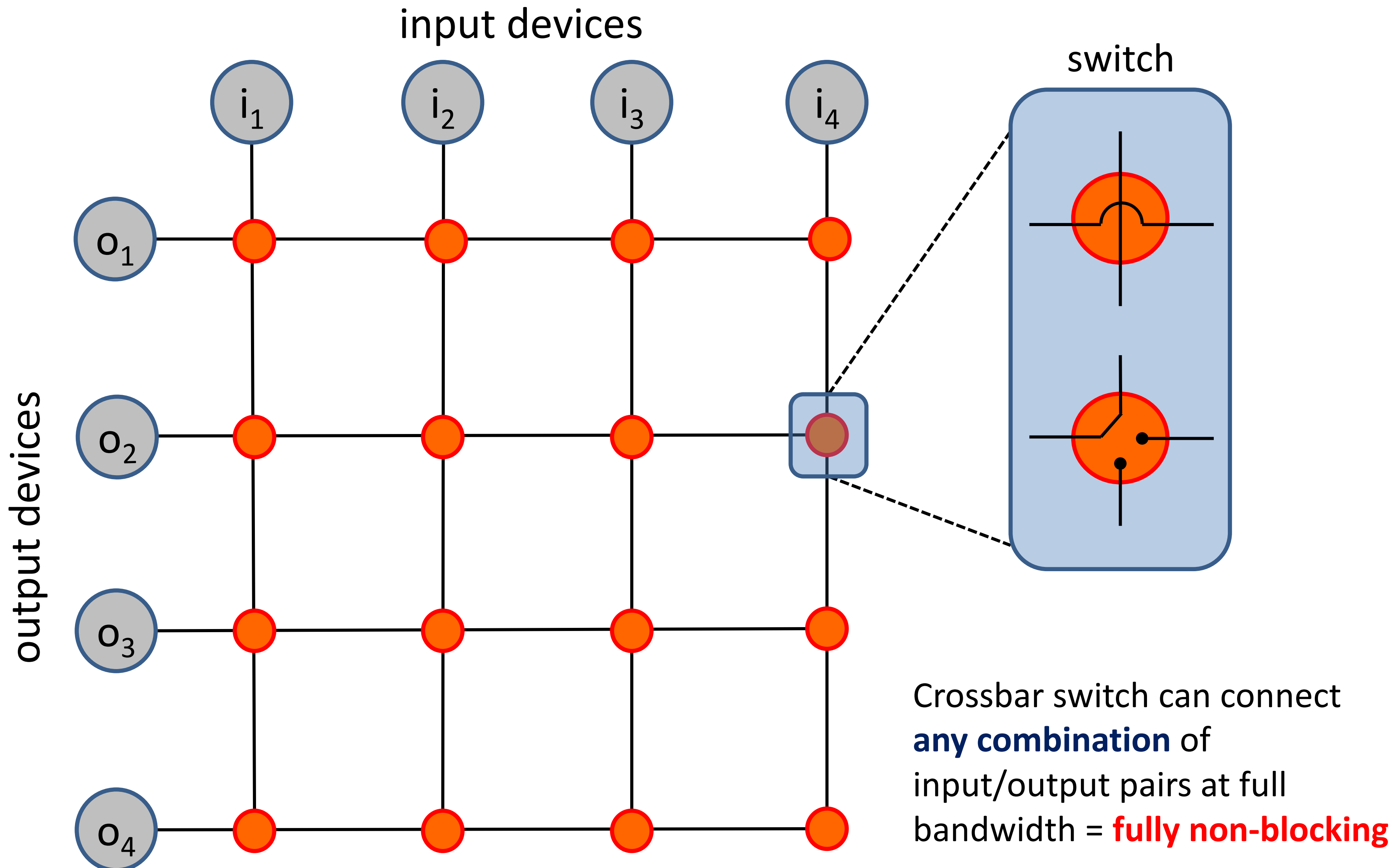
- No more than **two devices** can communicate at any given time
- Hardware controls which devices have access
- High risk of **contention** when multiple devices try to access the bus simultaneously. Bus is a "**blocking**" interconnect.



Bus network properties

- Bisection bandwidth = point-to-point bandwidth (independent of # devices)
- Diameter = 1 (single hop)

Crossbar switch



Crossbar switch can connect **any combination** of input/output pairs at full bandwidth = **fully non-blocking**

Static routing in crossbar switch

input devices

i_1 i_2 i_3 i_4

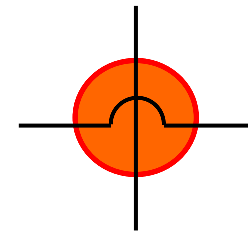
output devices

o_1 o_2 o_3 o_4

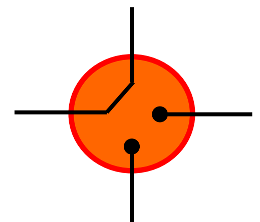
Input – output pairings

Input	Output
i_1	o_2
i_2	o_4
i_3	o_1
i_4	o_3

state A



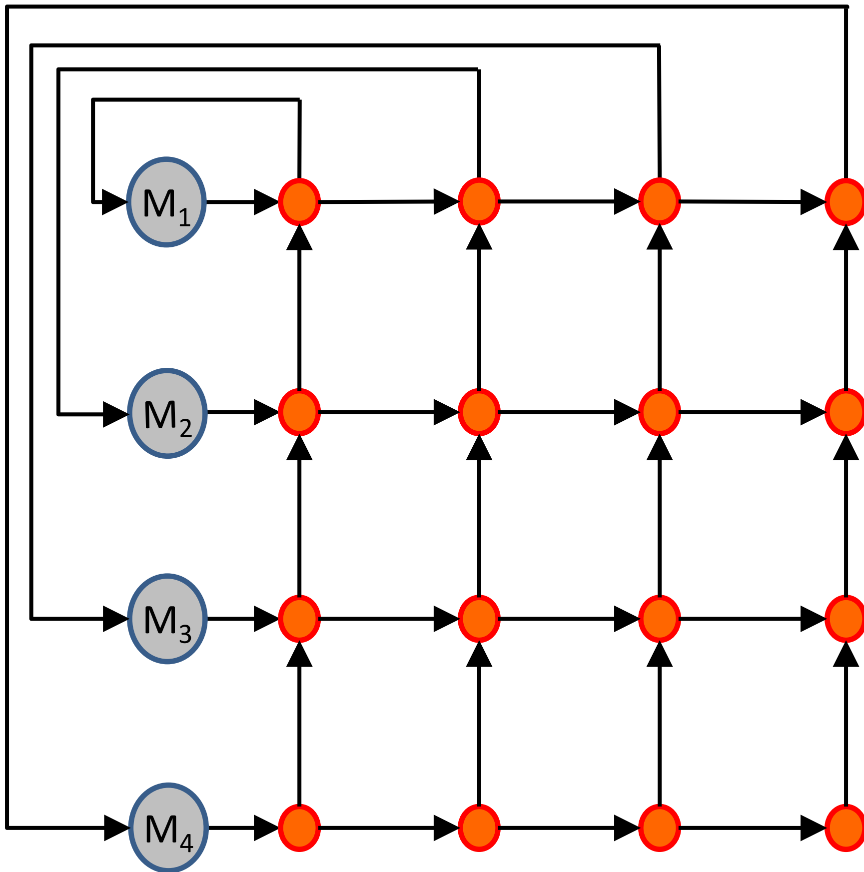
state B



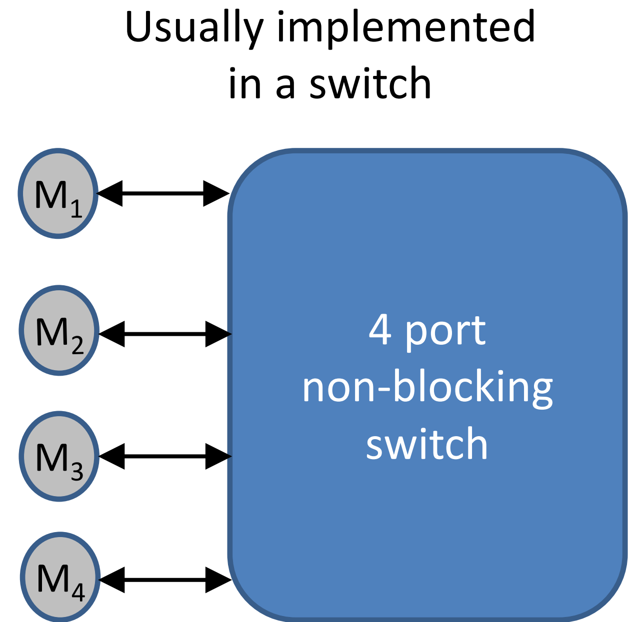
Switches at crossing of input/output should be in state B. Other switches should be in state A.

Path selection is fixed = static routing

Crossbar switch for distributed memory systems

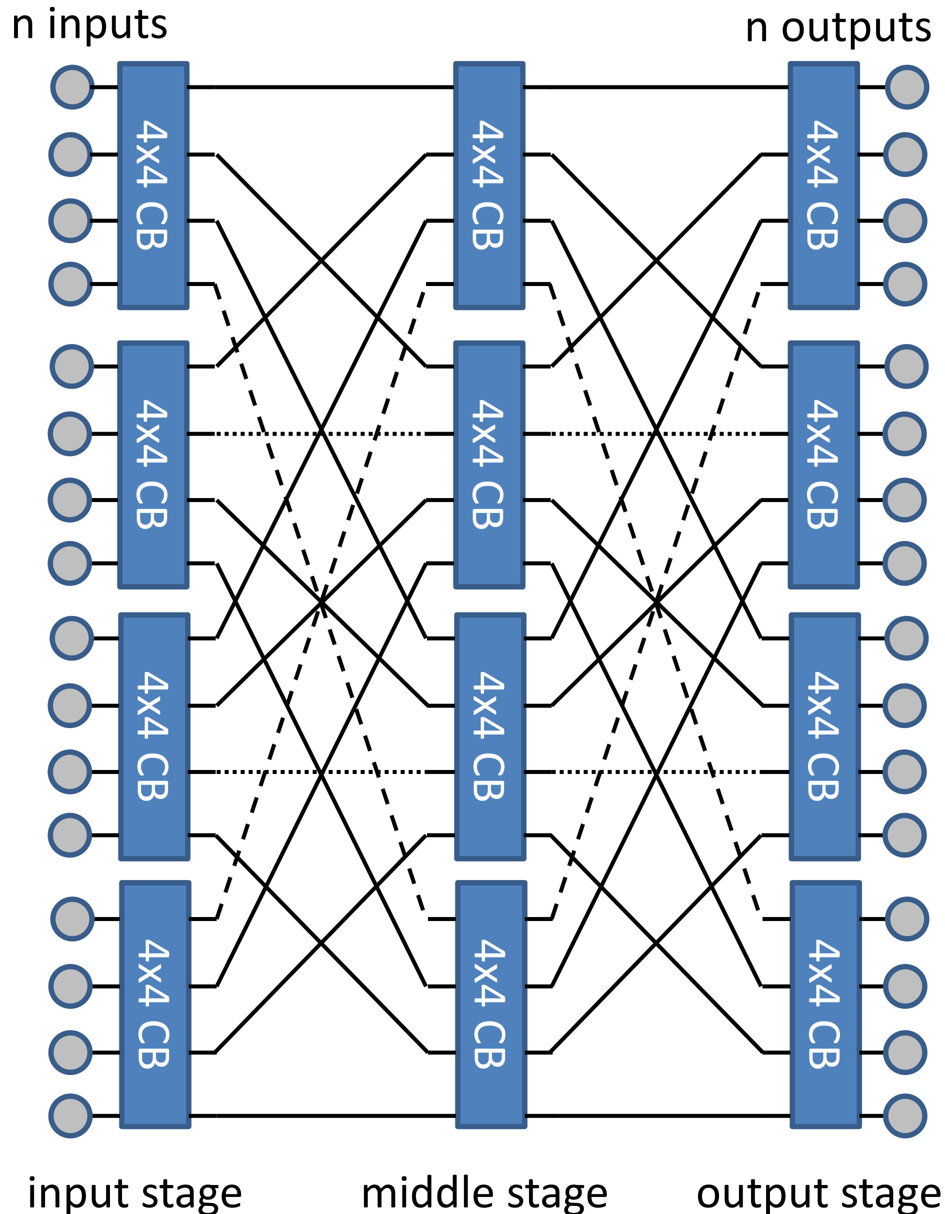


Usually implemented in a switch



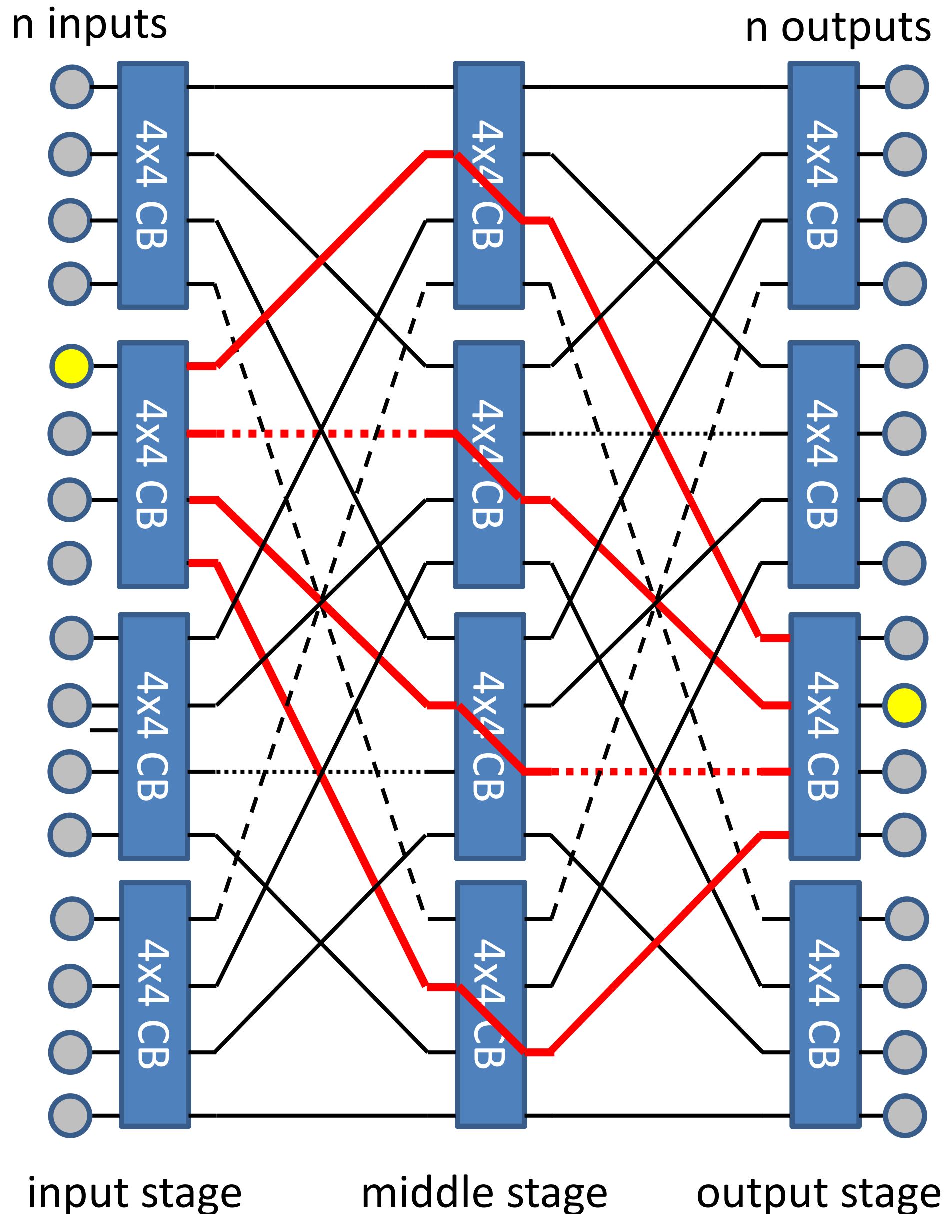
- Crossbar implementation of a switch with P = 4 machines.
  - **Fully non-blocking**
  - **Expensive: P^2 switches and $O(P^2)$ wires**

Clos switching networks



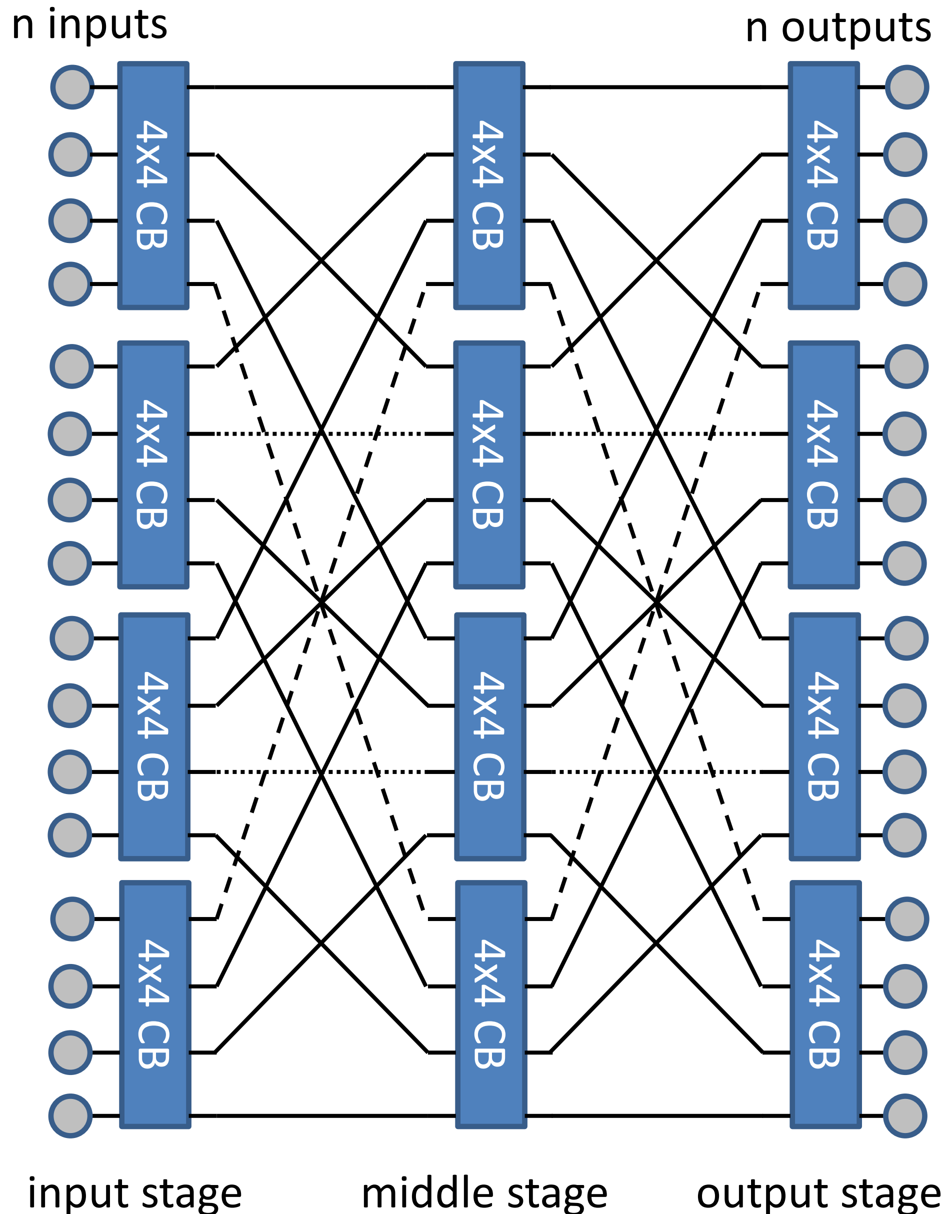
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.

Clos switching networks



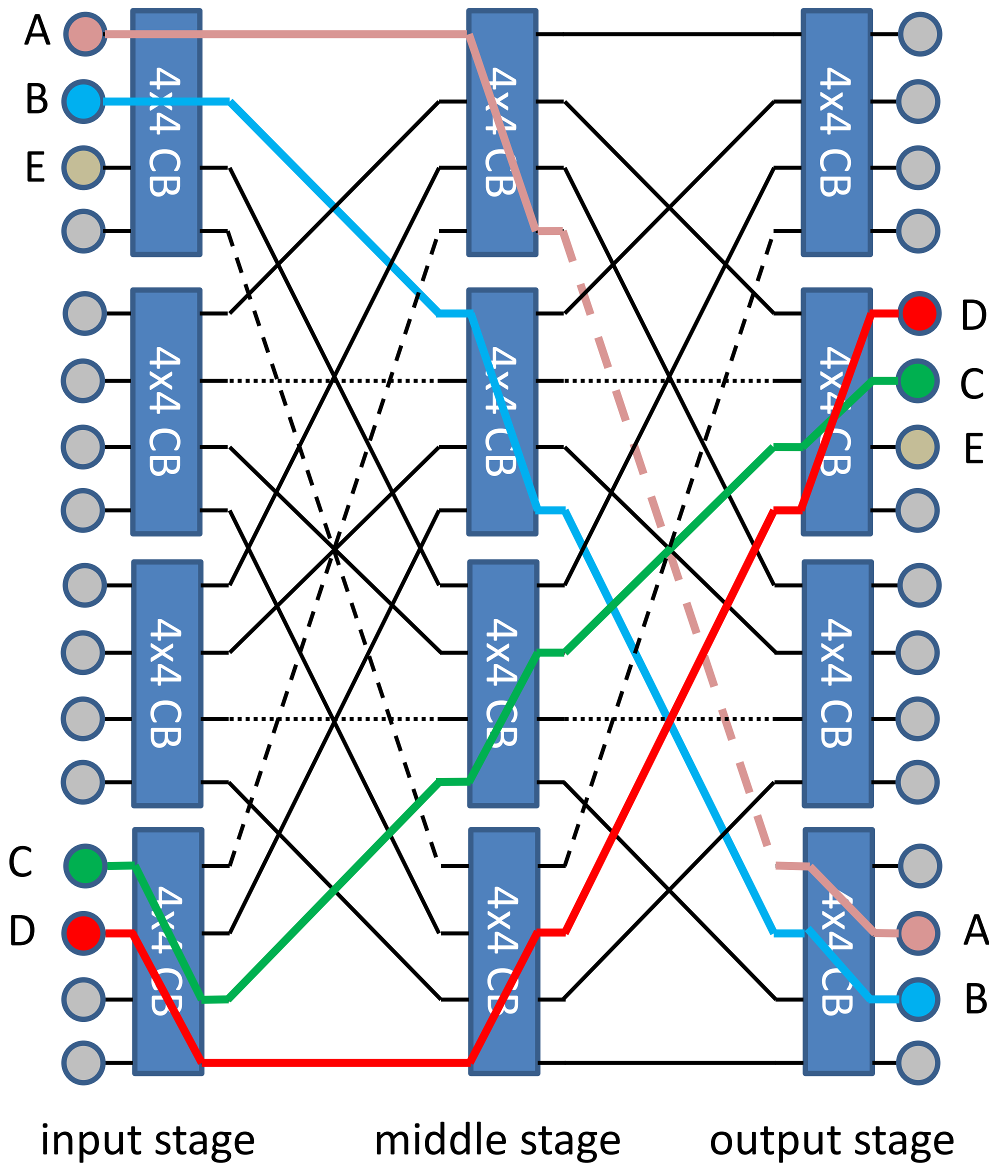
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.

Clos switching networks



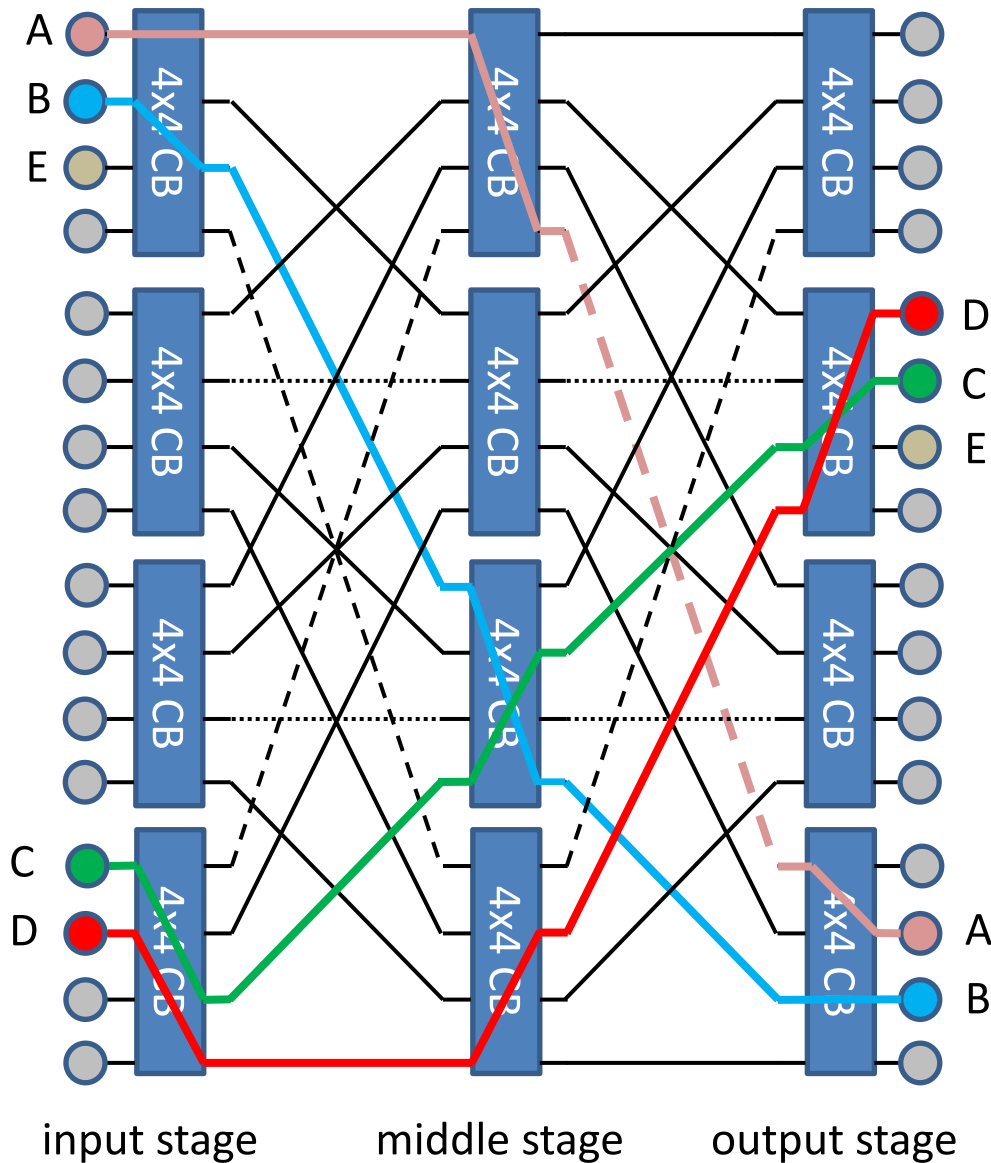
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.

Clos switching networks



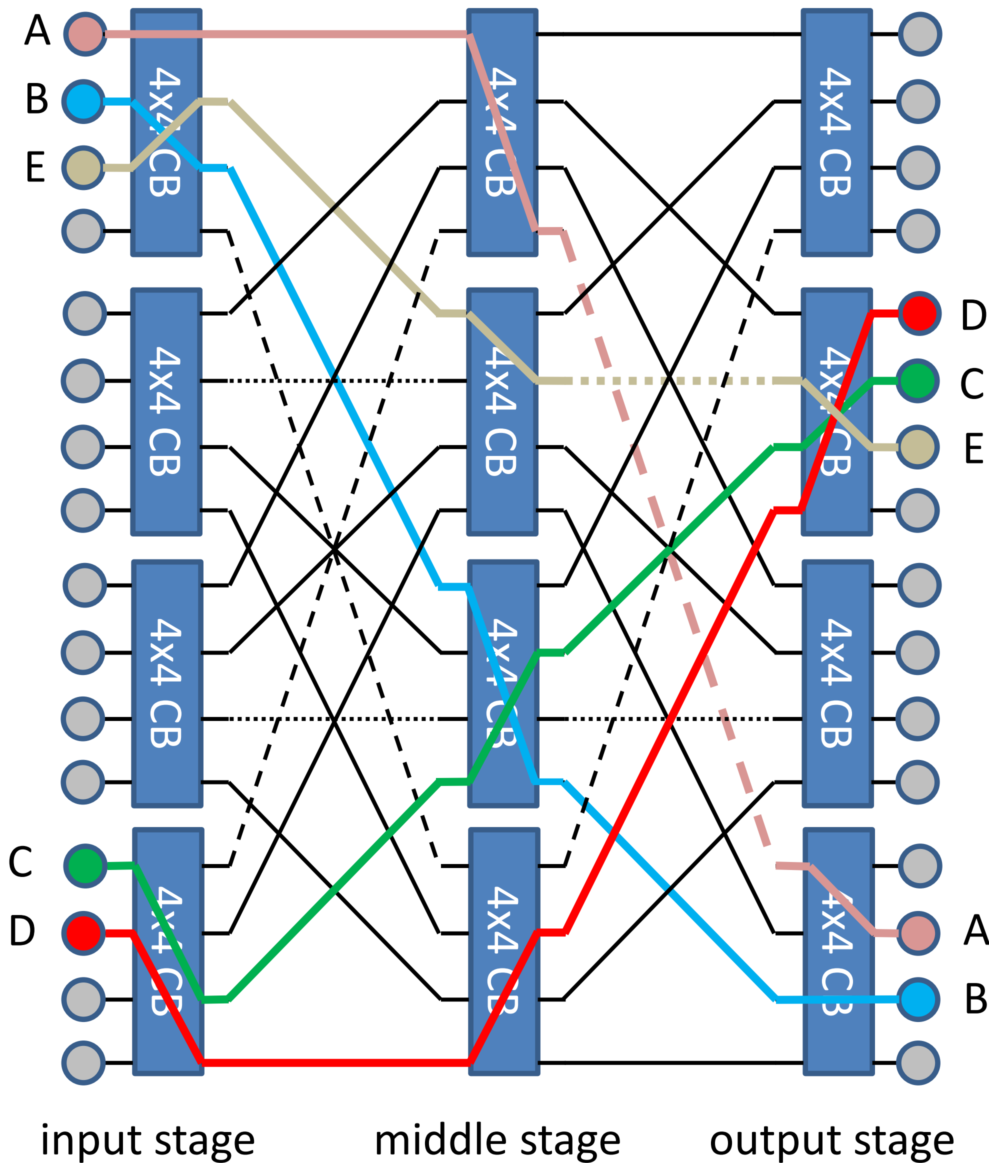
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.
- **Adaptive routing** may be necessary: **can not find a connection for E !**

Clos switching networks



- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.
- **Adaptive routing** may be necessary: **reroute an existing path**

Clos switching networks



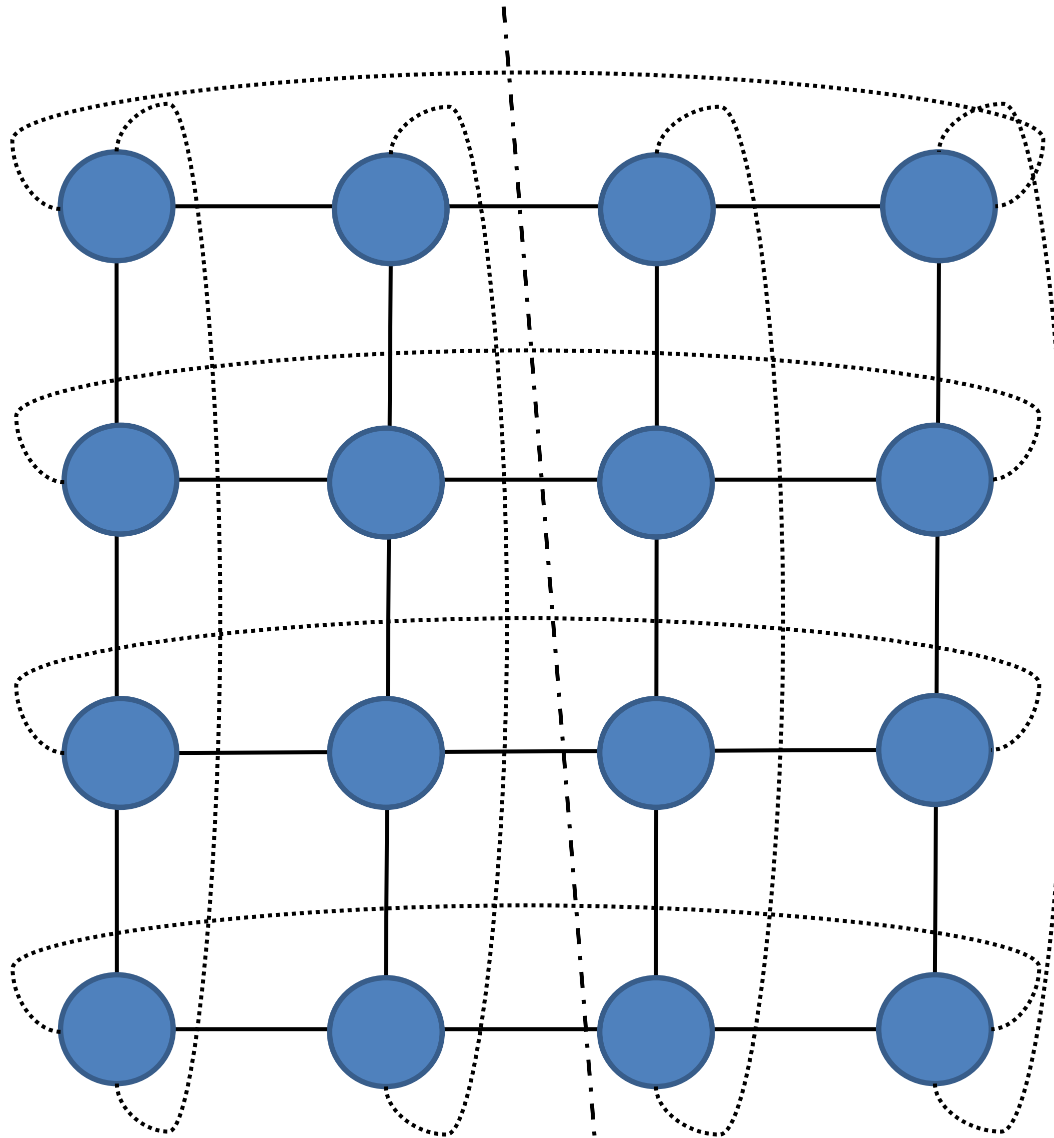
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate simultaneously (**non-blocking**)
- Requires far less wiring than conventional CB switches.
- **Adaptive routing** may be necessary: **E can now be connected**

Switch examples

Infiniband switch (24 ports)

Gigabit Ethernet switch (24 ports)

Mesh networks



Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl’s law
  - Gustafson’s law
- Parallel program development: case studies

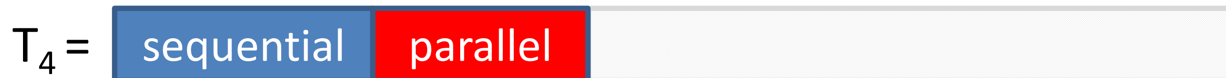
Basic performance terminology

- **Runtime** (“*How long does it take to run my program*”)
  - In practice, only wall clock time matters
  - Depends on the number of parallel processes **P**
  - $\mathbf{T_P}$ = runtime using P processes
- **Speedup** (“*How much faster does my program run in parallel*”)
  - $\mathbf{S_P = T_1 / T_P}$
  - In the ideal case, $S_P = P$
  - Super linear speedup are usually due to cache effects
- **Parallel efficiency** (“*How well is the parallel infrastructure used*”)
  - $\boldsymbol{\eta_P = S_P / P}$ $\quad (0 \leq \eta_P \leq 1)$
  - In the ideal case, $\eta_P = 1 = 100\%$
  - Depends on the application what is considered acceptable efficiency

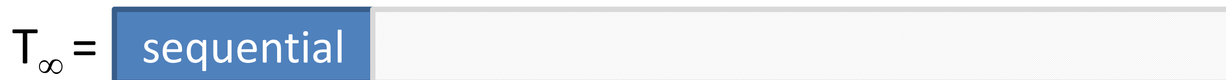
Strong scaling: Amdahl's law

- **Strong Scaling** = increasing the number of parallel processes for a fixed-size problem
- Simple model: partition sequential runtime T_1 in a parallelizable fraction (1-s) and inherently sequential fraction (s).
  - $\mathbf{T_1 = sT_1 + (1\text{-}s)T_1}$ $(0 \leq s \leq 1)$

  - Therefore, $\mathbf{T_P = sT_1 + \frac{(1-s)T_1}{P}}$



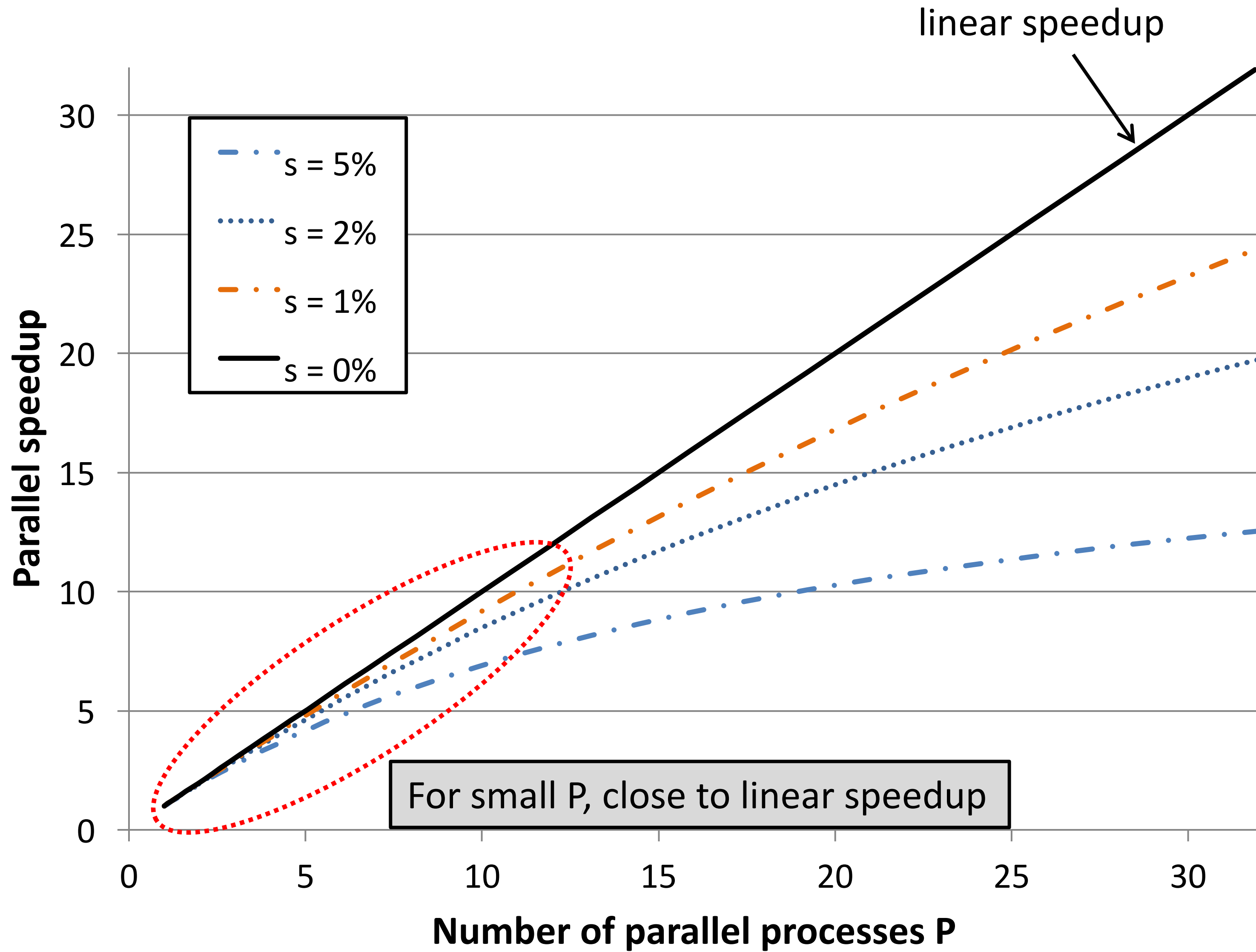
…



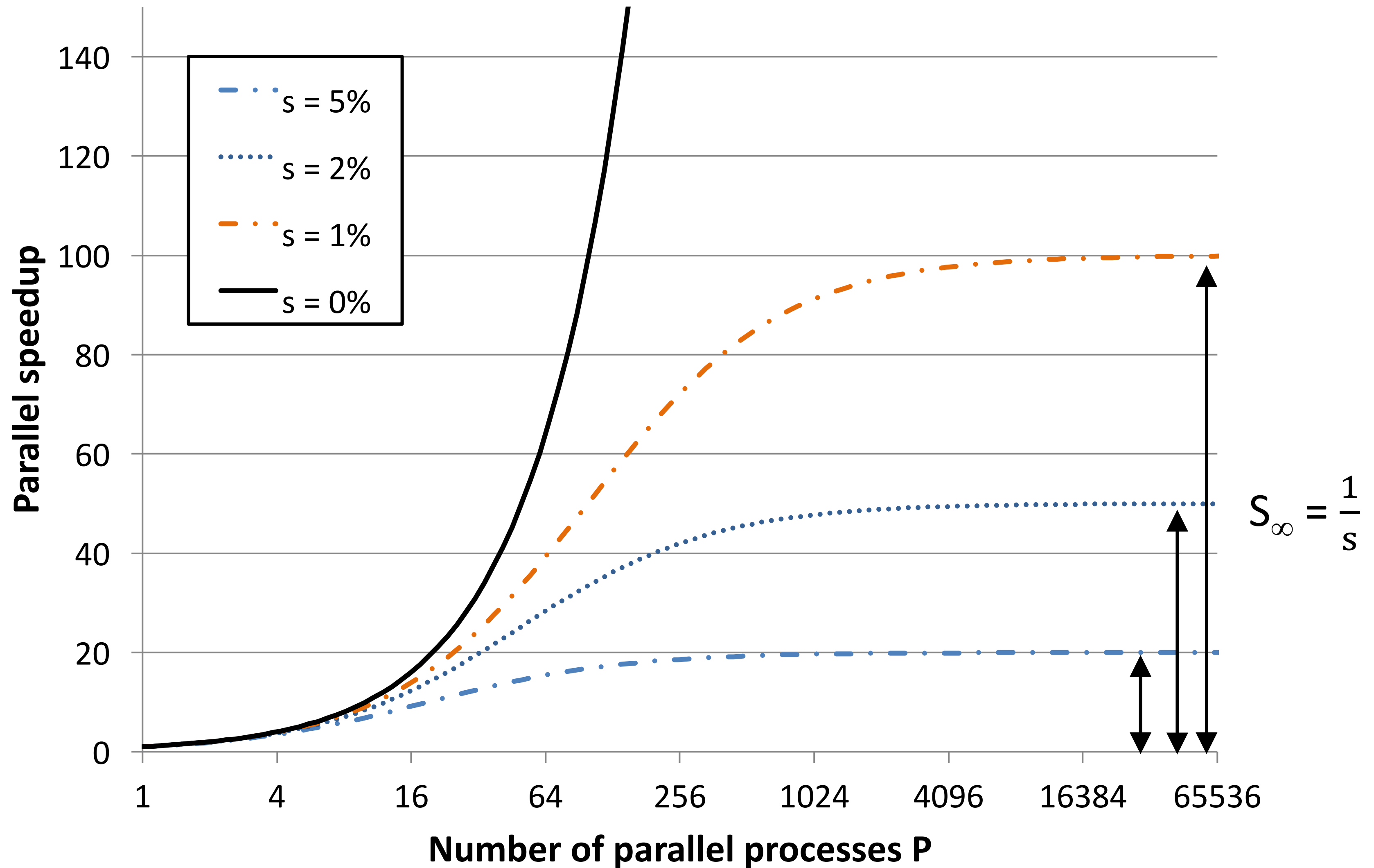
Strong scaling: Amdahl's law

- Consequently, $S_P = \frac{T_1}{T_P} = \frac{T_1}{sT_1 + \frac{(1-s)T_1}{P}} = \frac{1}{s + \frac{1-s}{P}}$ (**Amdahl's law**)
- Speedup is **bounded** $S_\infty = \frac{1}{s}$
  - e.g. $s = 1\%$, $S_\infty = 100$
  - e.g. $s = 5\%$, $S_\infty = 20$
- Sources of sequential fraction sT_1
  - Process startup overhead
  - Inherently sequential portions of the code
  - Dependencies between subtasks
  - Communication overhead
  - Function calling overhead
  - Load misbalance

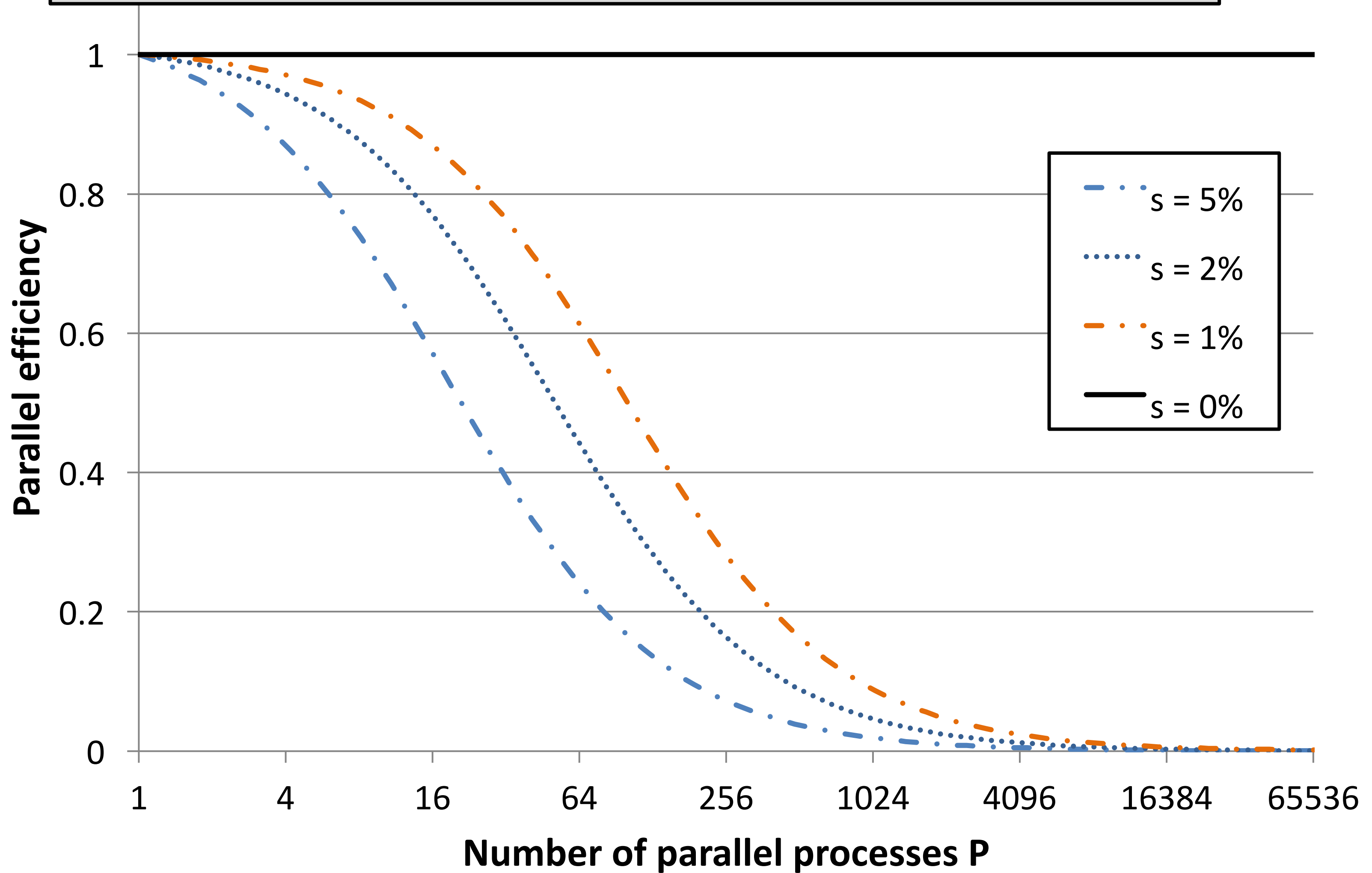
Amdahl's law



Amdahl's law

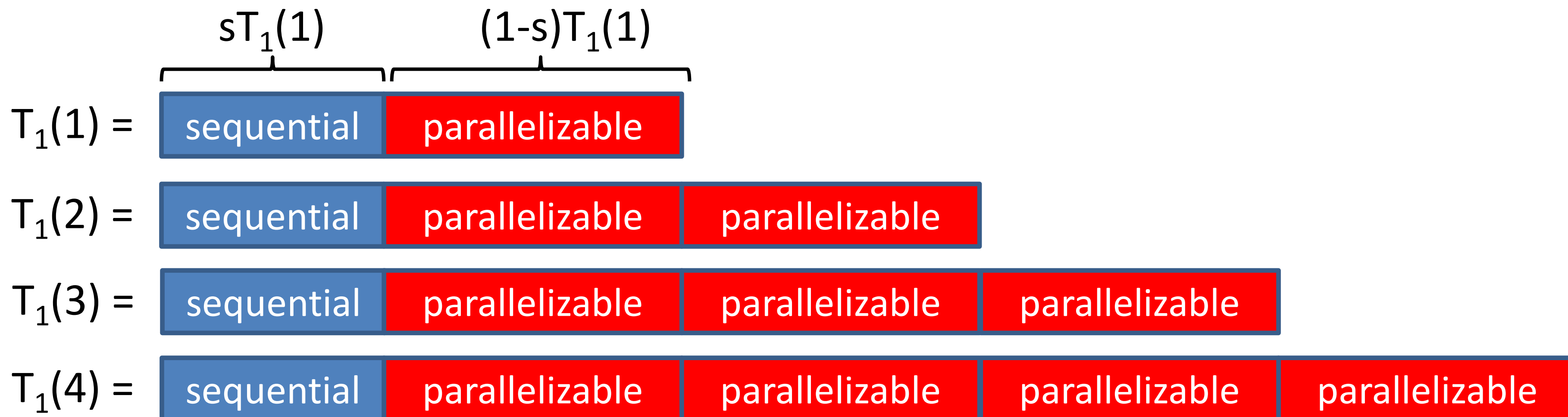


Amdahl's law



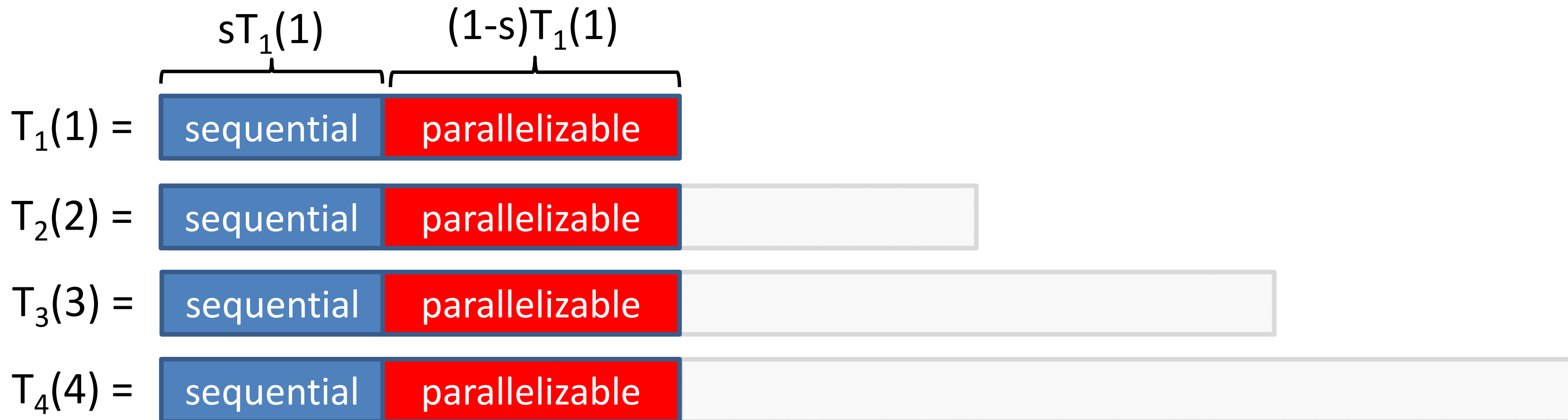
Weak scaling: Gustafson's law

- **Weak scaling**: increasing both the number of parallel processes P and the problem size N.
- Simple model: assume that the sequential part is constant, and that the parallelizable part is proportional to N.
  - $T_1(N) = T_1(1)[\ s + (1-s)N\ ]$ $(0 \leq s \leq 1)$



Gustafson's law

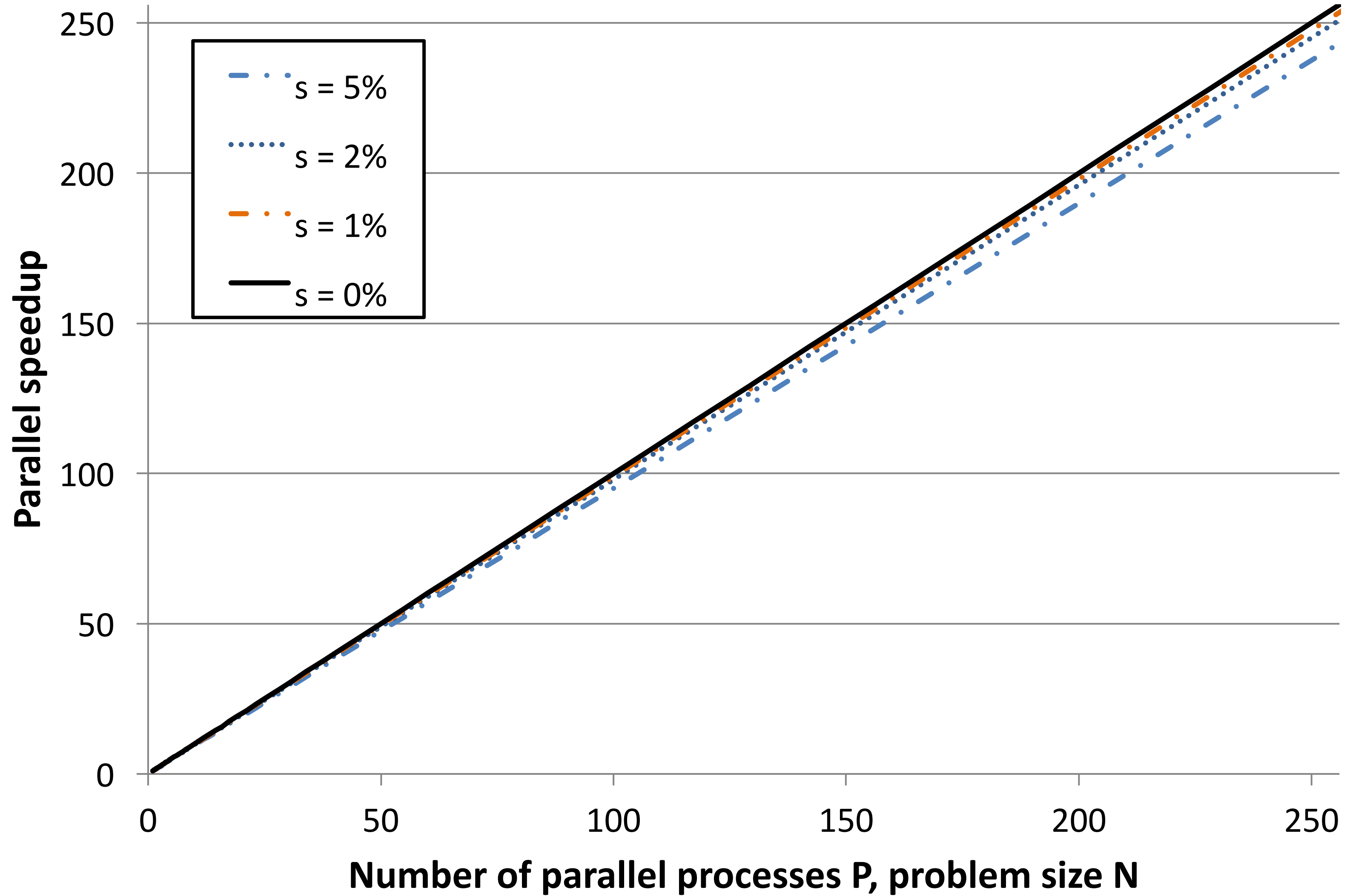
- Solve increasingly larger problems using a proportionally higher number of processes (N = P).



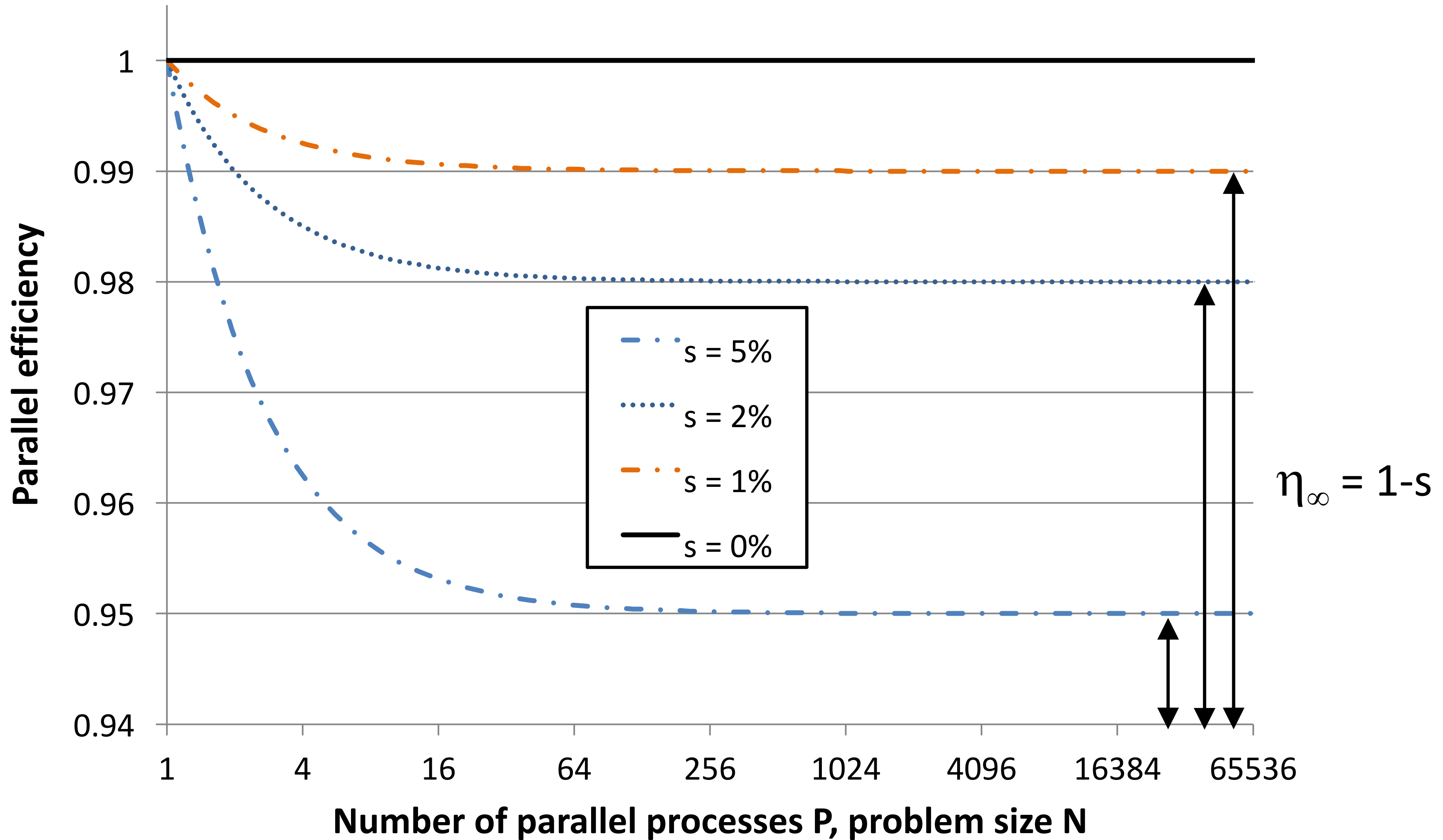
Therefore, $T_P(N) = T_1(1)[\, s + \frac{(1-s)N}{P} \,]$ and hence $T_P(P) = T_1(1)$

Speedup $S_P(P) = s + (1\text{-}s)P = P - s(P\text{-}1)$ **(Gustafson's law)**

Gustafson's law



Gustafson's law



Outline

- Distributed-memory architecture: general considerations
- Programming model: Message Passing Interface (MPI)
  - Point-to-point communication
    - Blocking communication
    - Point to point network performance
    - Non-blocking communication
  - Collective communication
    - Collective communication algorithms
    - Global network performance
- Parallel program performance evaluation
  - Amdahl's law
  - Gustafson's law
- Parallel program development: case studies

Case Study 1: Parallel matrix-matrix product

Case study: parallel matrix multiplication

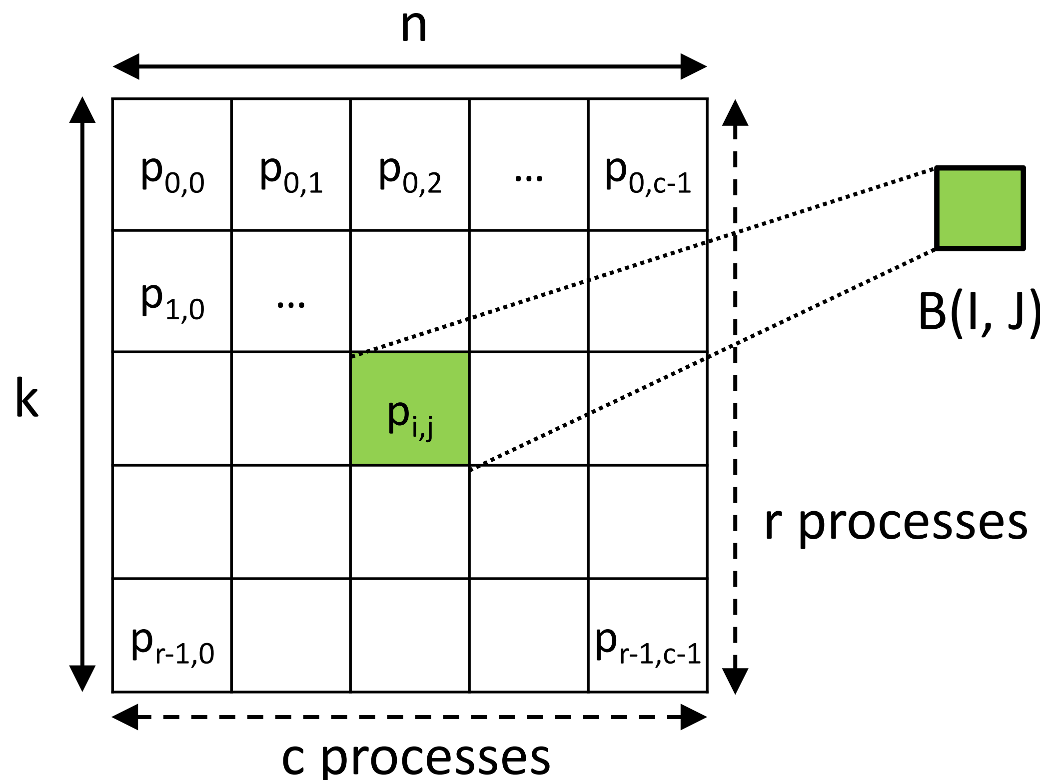
- Matrix-matrix multiplication: **$C = \alpha * A * B + \beta * C$** (BLAS xgemm)
  - Assume C = m x n; A = m x k; B = k x n matrix.
  - α, β = scalars; assume $\alpha = \beta = 1$ in what follows, i.e. C = C + A*B
- Initially, **matrix elements are distributed** among P processes
  - Assume **same scheme for each matrix** A, B and C
- Each process computes values for C that are **local to that process**
  - Required data from A and B that is not local needs to be **communicated**
  - Performance modeling assuming the **α, β, γ** model
    - α = latency
    - β = per element transfer time
    - γ = time for single floating point computation

Case study reproduced from J. Demmel

Case study: parallel matrix-matrix product

- **Two dimensional partitioning**
  - Partition matrices in 2D in an r x c mesh (P = r*c)
  - X(I, J) refers to block (I, J) of matrix X (X = {A, B, C})
  - Process p_n is also denoted by $p_{i,j}$ (n = i*c + j) and holds X(I,J)

e.g. matrix B



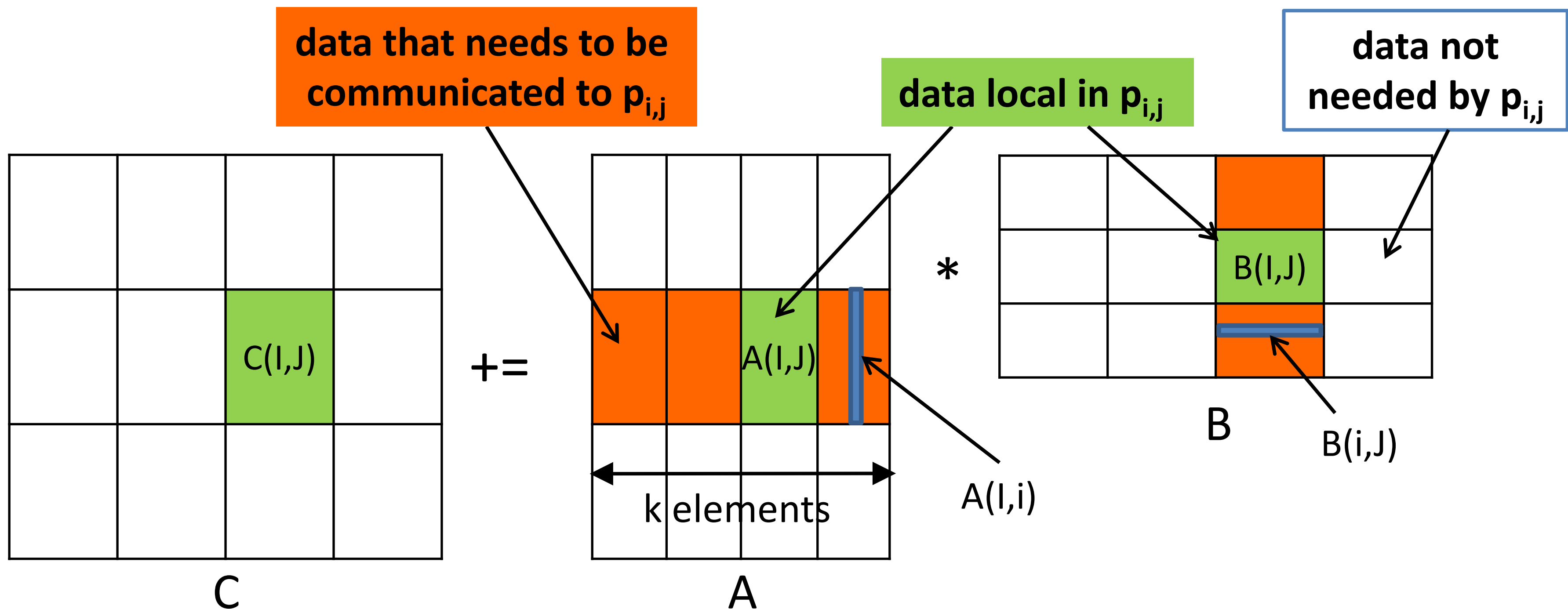
Case study reproduced from J. Demmel

Case study: parallel matrix-matrix product

- Second approach: **two dimensional partitioning (SUMMA)**
  - Process $p_{i,j}$ needs to compute C(I, J):

$$C(I,J) = C(I,J) + \sum_{i=0}^{k-1} A(I,i) * B(i,J) \qquad \overbrace{\forall\, I = 0 \ldots r \quad \forall\, J = 0 \ldots c}^{\text{do this in parallel}}$$

index i refers to a single column of A or single row of B



Case study reproduced from J. Demmel

Case study: parallel matrix-matrix product

- Second approach: **two dimensional partitioning**
  - SUMMA algorithm (all I and J in parallel)

```
for i = 0 to k-1
    broadcast A(I, i) within process row
    broadcast B(i, J) within process column
    C(I,J) += A(I, i) * B(i, J)
endfor
```

  - Cost for inner loop (executed k times):
    - $\log_2 c\,(\alpha + \beta(m/r)) + \log_2 r\,(\alpha + \beta(n/c)) + 2mn\gamma/P$
  - Total $T_P = 2kmn\gamma/P + k\alpha(\log_2 c + \log_2 r) + k\beta((m/r)\log_2 c) + (n/c)\log_2 r)$
  - For n = m = k and r = c = sqrt(P) we find:

  Parallel efficiency η_P $= 1 / (1 + (\alpha/\gamma)(P\log_2 P)/(2n^2) + (\beta/\gamma)\sqrt{P}\,\log P/n)$

  Isoefficiency when n grows as $\sqrt{P}$ (constant memory per node!)

Case study reproduced from J. Demmel

Case study: parallel matrix-matrix product

- Even more efficient: use **“blocking”** algorithm

```
for i = 0 to k-1 step b
    end = min(i+b-1, k-1)
    broadcast A(I, i:end) within process row
    broadcast B(i:end, J) within process column
    C(I,J) += A(I, i:end) * B(i:end, J)
endfor
```

Perform this product using Level-3 BLAS

- SUMMA algorithm is implemented in **PBLAS = Parallel BLAS**
- Algorithm can be extended to block-cyclic layout (see further)

Case study: parallel Gaussian Elimination

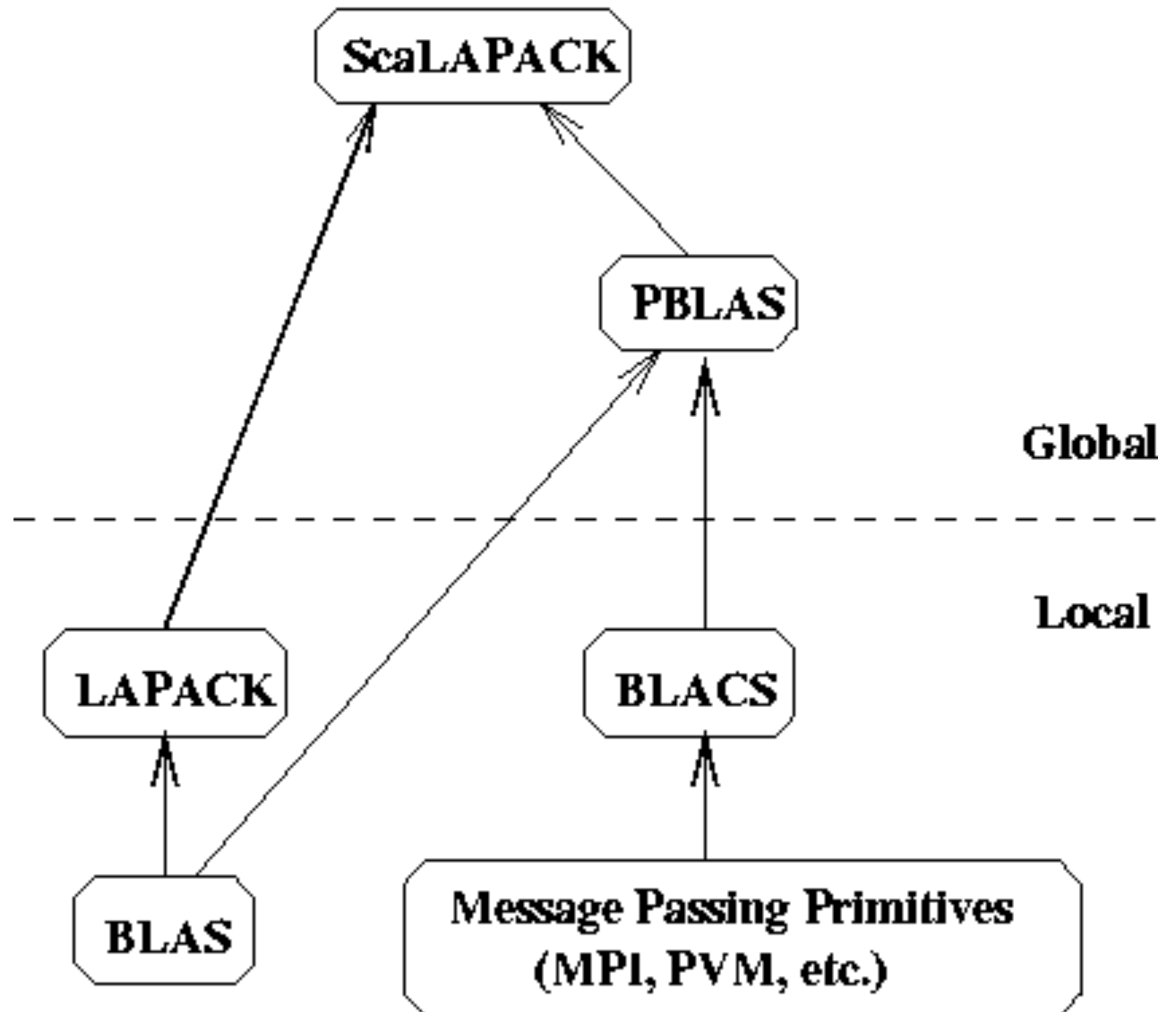


Image take from J. Demmel

Case Study 2: Parallel Sorting

Case study: parallel sorting algorithm

- **Sequential** sorting of n keys (1st Bachelor)
  - Bubblesort: **$O(n^2)$**
  - Mergesort: **$O(n \log n)$**, even in worst-case
  - Quicksort: **$O(n \log n)$** expected, $O(n^2)$ worst-case, fast in practice
- **Parallel** sorting of n keys, using P processes
  - Initially, each process holds n/p keys (unsorted)
  - Eventually, each process holds n/p keys (sorted)
    - Keys per process are sorted
    - If $q < r$, each key assigned to process q is less than or equal to every key assigned to process r (sort keys in rank order)

Case study: parallel sorting algorithm

Bubblesort algorithm: $O(n^2)$

```
void Bubble_sort(int *a, int n) {
    for (int listLen = n; listLen >= 2; listLen--)
        for (int i = 0; i < listLen-1; i++)
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1]
                a[i+1] = temp;
            }

}
```

"Compare-swap" operation (lines `if (a[i] > a[i+1]) {` to `a[i+1] = temp;`)

Example: 5 2 4 8 1 → 2 4 5 1 **8** after iteration 1
→ 2 4 1 **5 8** after iteration 2
→ 2 1 **4 5 8** after iteration 3
→ 1 **2 4 5 8** after iteration 4

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

- **Bubblesort**
  - Result of current step (a[i] > a[i+1]) depends on previous step
    - Value of a[i] is determined by previous step
    - Algorithm is **"inherently serial"**
    - Not much point in trying to parallelize this algorithm
- **Odd-even transposition sort**
  - Decouple algorithm in two phases: even and odd
    - **Even phase**: compare-swap on following elements:
      (a[0], a[1]), (a[2], a[3]), (a[4], a[5]), …
    - **Odd phase**: compare-swap operations on following elements:
      (a[1], a[2]), (a[3], a[4]), (a[5], a[6]), …

Case study: parallel sorting algorithm

Even-odd transposition sort algorithm: $O(n^2)$

```
void Even_odd_sort(int *a, int n) {
    for (int phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { // even phase
            for (int i = 0; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1]
                    a[i+1] = temp;
                }
        } else {                // odd phase
            for (int i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1]
                    a[i+1] = temp;
                }
        }
}
```

"Compare-swap" operation (annotates the if block in the even phase)

"Compare-swap" operation (annotates the if block in the odd phase)

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

Even-odd transposition sort algorithm: $O(n^2)$

Example:	5 2 4 8 1	→	[2 5] [4 8] 1	even phase
		→	2 [4 5] [1 8]	odd phase
		→	[2 4] [1 5] 8	even phase
		→	2 [1 4] [5 8]	odd phase
		→	[1 2] [4 5] 8	even phase

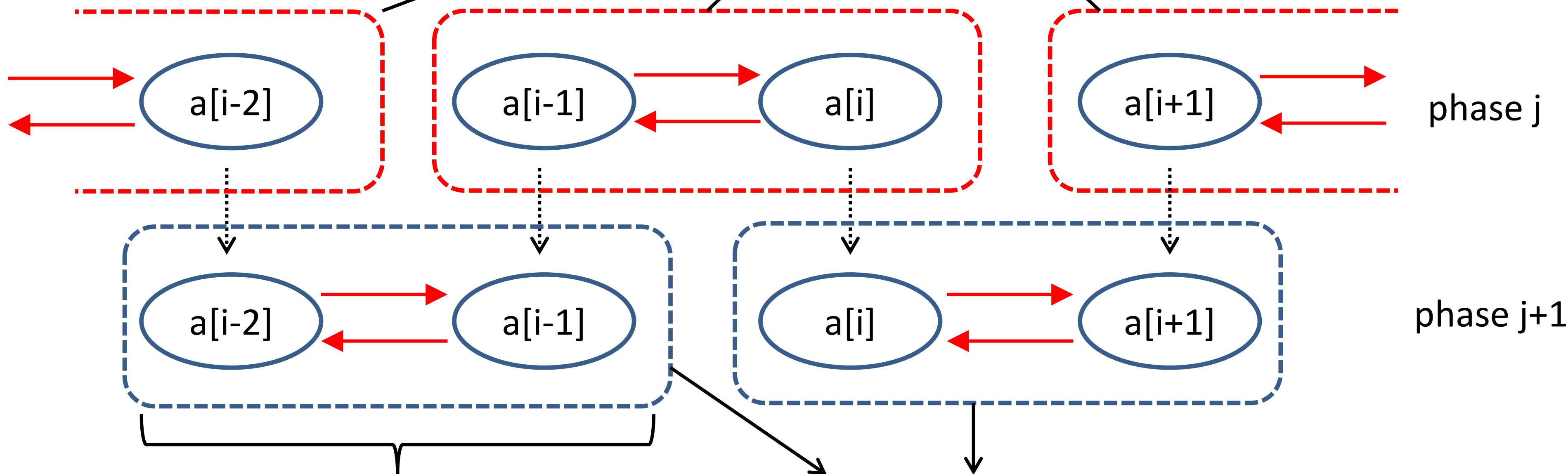
Parallelism within each even or odd phase is now obvious:
Compare-swap between (a[i], a[i+1]) independent from (a[i+2], a[i+3])

Case study: parallel sorting algorithm

execute in parallel during **phase j**

Parallel algorithm

- First, assume P == n (one element per process)



execute in parallel during **phase j+1**

Communicate value with neighbor

- Right process (highest rank) keeps largest value
- Left process (lowest rank) keeps smallest value

Image reproduced from P. Pacheco

Case study: parallel sorting algorithm

Parallel algorithm

- Now, assume **n/P >> 1** (as is typically the case)
- Example: P = 4; n = 16

	process 0	process 1	process 2	process 3
initial values	15,11,9,16	3,14,8,7	4,6,12,10	5,2,13,1
local sorting	9,11,15,16	3,7,8,14	4,6,10,12	1,2,5,13
phase 0 (even)	3,7,8,9	11,14,15,16	1,2,4,5	6,10,12,13
phase 1 (odd)	3,7,8,9	1,2,4,5	11,14,15,16	6,10,12,13
phase 2 (even)	1,2,3,4	5,7,8,9	6,10,11,12	13,14,15,16
phase 3 (odd)	1,2,3,4	5,6,7,8	9,10,11,12	13,14,15,16

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

Theorem: Parallel odd-even transposition sort algorithm will sort the input list after P (= number of processes) phases.

Parallel even-odd transposition sort **pseudocode**

```
sort local keys
for (int phase = 0; phase < P; phase++) {
    neighbor = computeNeighbor(phase, myRank);
    if (I'm not idle) { // first and/or last process may be idle
        send all my keys to neighbor
        receive all keys from neighbor
        if (myRank < neighbor)
            keep smaller keys
        else
            keep larger keys
     }
}
```

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

Implementation of computeNeighbor (**MPI**)

```
int computeNeighbor(int phase, int myRank) {
    int neighbor;
    if (phase % 2 == 0) {
        if (myRank % 2 == 0)
            neighbor = myRank + 1;
        else
            neighbor = myRank - 1;
    } else {
        if (myRank % 2 == 0)
            neighbor = myRank - 1;
        else
            neighbor = myRank + 1;
    }
    if (neighbor == -1 || neighbor == P-1)
        neighbor = MPI_PROC_NULL;
    return neigbor;
}
```

When used as destination or source rank in MPI_Send or MPI_Recv, no communication takes place

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

Implementation of **data exchange in MPI**

- Be careful of **deadlocks**
- In both even and odd phases, communication always takes place between a process with even, and a process with odd rank

```
if (myRank % 2 == 0) {
    MPI_Send(...)
    MPI_Recv(...)
} else {
    MPI_Recv(...)
    MPI_Send(...)
}
```

Exchange order to prevent deadlocks !

OR

```
MPI_Sendrecv(...)
```

Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

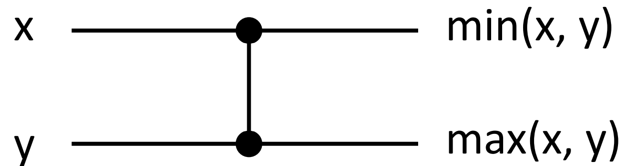
Parallel odd-even transposition sort algorithm **analysis**

- **Initial sorting**: $O(n/P \log(n/P))$ time
  - Use an efficient sequential sorting algorithm, e.g. quicksort or mergesort
- **Per phase**: $2(\alpha + n/P\ \beta) + \gamma n/P$
- **Total runtime** $T_P(n) = O(n/P \log(n/P) + 2(\alpha P + n\beta) + \gamma n$
  $= 1/P\ O(n \log n) + O(n)$
- Linear speedup when P is small and n is large
- However, **bad asymptotic behaviour**
  - When n and P increase proportionally, runtime per process is $O(n)$
  - What we really want: $O(\log n)$
  - Difficult! (but possible!)

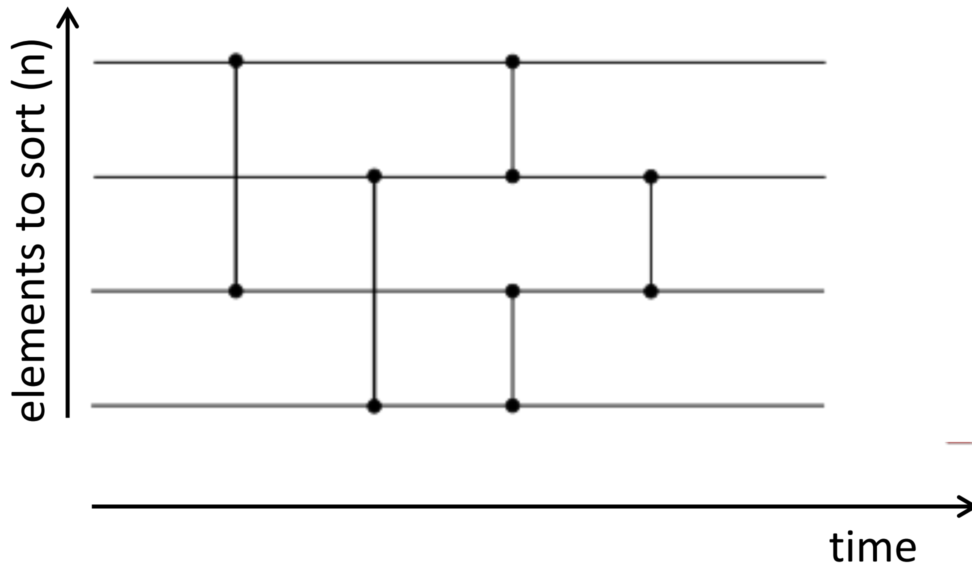
Algorithm reproduced from P. Pacheco

Case study: parallel sorting algorithm

- **Sorting networks** (= graphical depiction of sorting algorithms)
  - Number of horizontal "**wires**" (= elements to sort)
  - Connected by vertical "**comparators**" (= compare and swap)

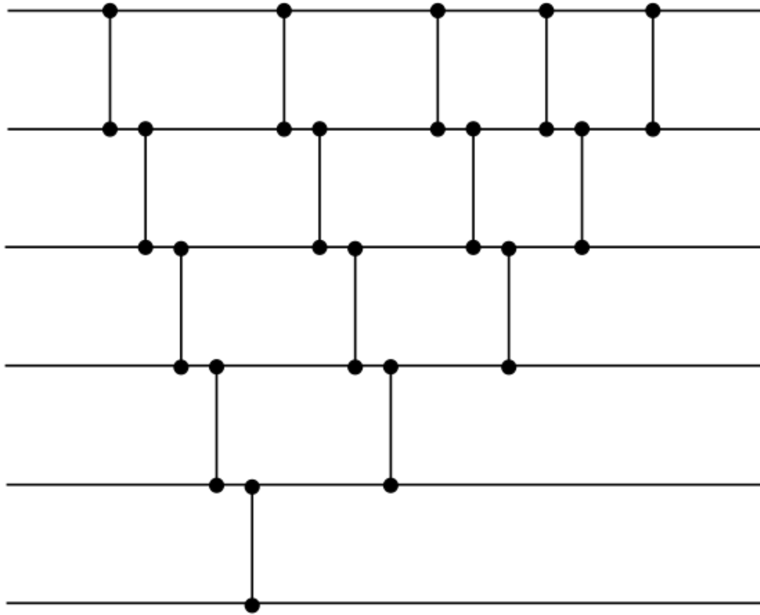


Example (4 elements to sort)

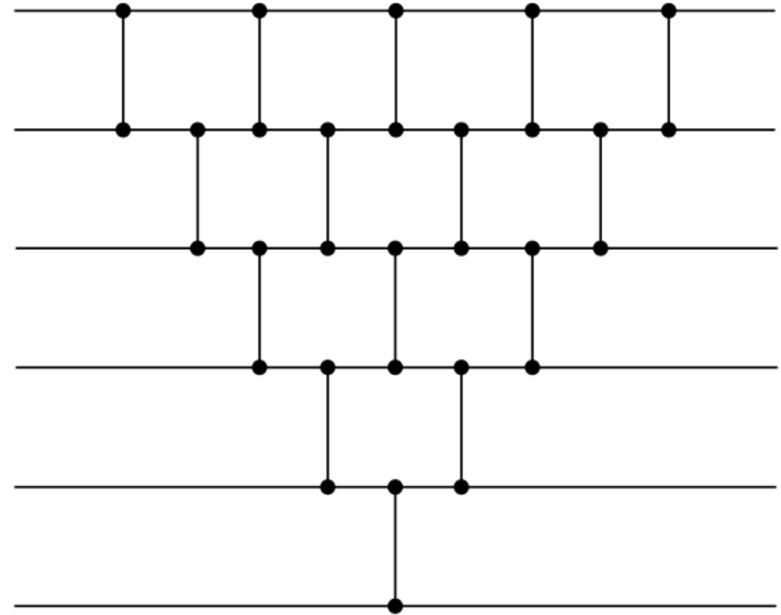


Case study: parallel sorting algorithm

Bubblesort algorithm (sequential)



Same bubblesort algorithm (**parallel**)



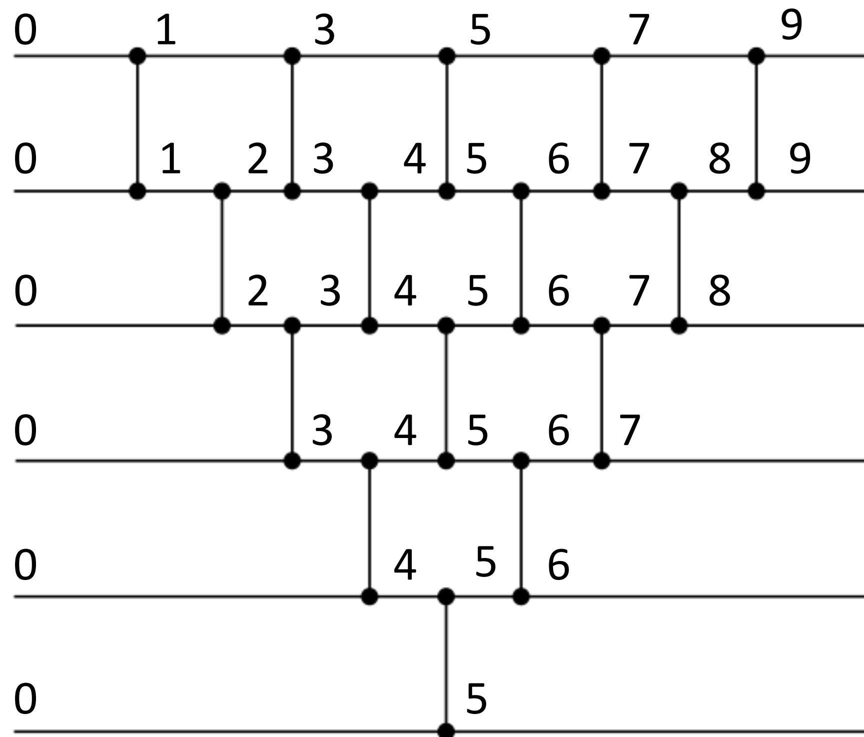
Sequential runtime
= number of comparators
= "**size** of the sorting network"
= n * (n – 1) / 2

Parallel runtime
(assume P == n processes)
= "**depth** of sorting network"
= 2n – 3

Case study: parallel sorting algorithm

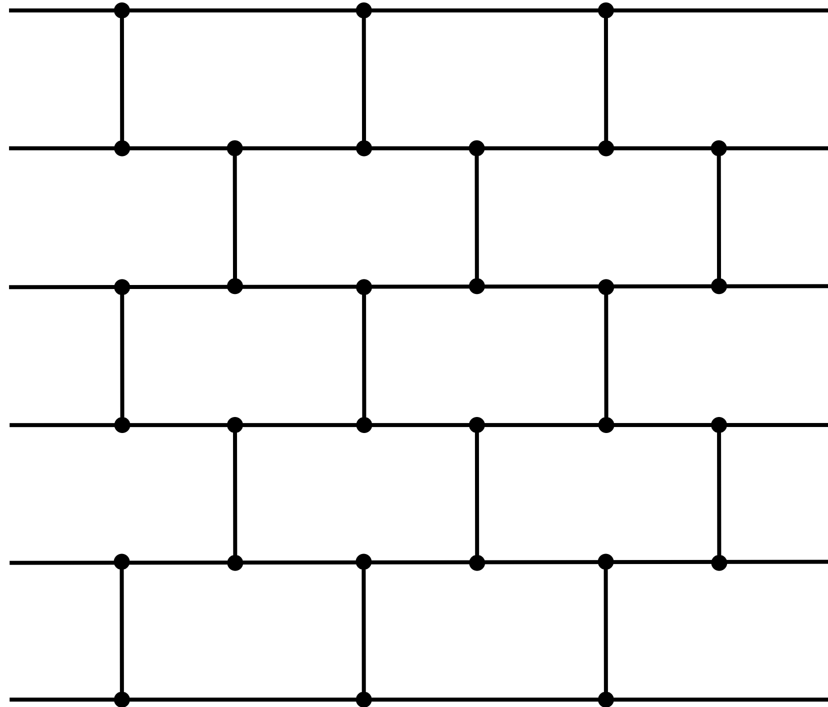
Definition: **depth of a sorting network** (= parallel runtime)

- Zero at the inputs or each wire
- For a comparator with inputs with depth d_1 and d_2, the depth of its outputs is **$1 + \max(d_1, d_2)$**
- Depth of the sorting network = maximum depth of each output



Case study: parallel sorting algorithm

- Sorting network of **odd-even transposition sort**



- Parallel runtime = "**depth** of sorting network" = n
- … or P (we assume n == P)

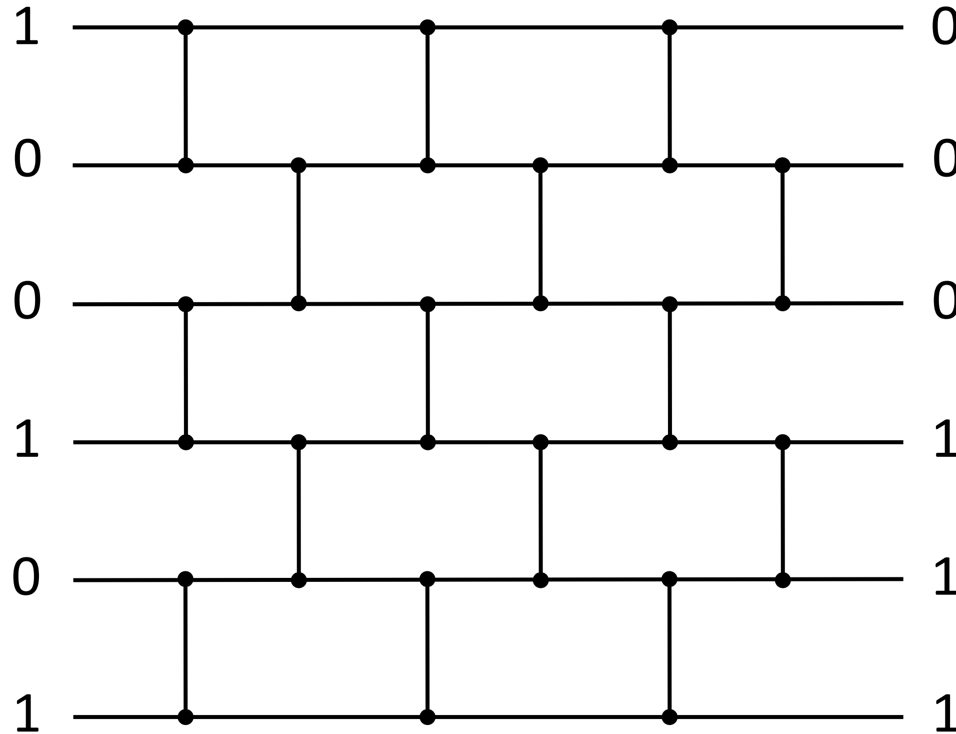
Case study: parallel sorting algorithm

- **Can we do better?**
  - Sequential sorting algorithms are **$O(n \log n)$**
  - Ideally, P and n can scale proportionally: **$P = O(n)$**
  - That means that we want to **sort n numbers in $O(\log n)$ time**
    - This is possible (!), however, big constant pre-factor
  - We will describe an algorithm that can sort n numbers in **$O(\log^2 n)$** parallel time (using $P = O(n)$ processes)
    - This algorithm has **best performance in practice**
    - … unless n becomes huge ($n > 2^{2000}$)
    - Nobody wants to sort that many numbers

Case study: parallel sorting algorithm

- **Theorem**: If a sorting network with n inputs sorts all 2^n binary strings of length n correctly, then it sorts all sequences correctly *(proof: see references)*.

Example



We will design an algorithm that can sort binary sequences in **$O(\log_2^2 n)$** time

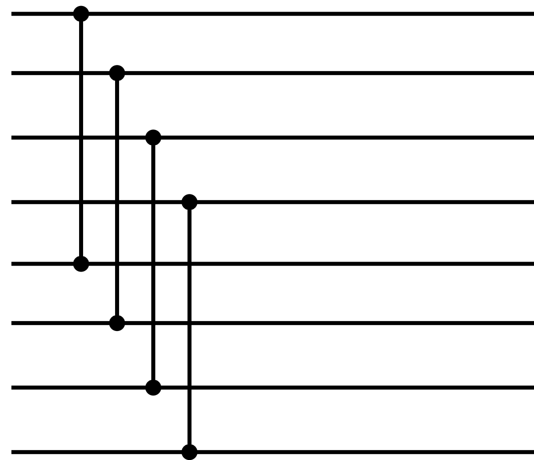
Case study: parallel sorting algorithm

- **Step 1:** create a sorting network that sorts bitonic sequences
- **Definition**: A **bitonic sequence** is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.
  - (1, 2, 3, 3.14, 5, 4, 3, 2, 1) is bitonic
  - (4, 5, 4, 3, 2, 1, 1, 2, 3) is bitonic
  - (1, 2, 1, 2) is not bitonic
- Over zeros and ones, a bitonic sequence is of the form
  - $0^i1^j0^k$ or $1^i0^j1^k$ (with e.g. 0^i = 0000…0 = i consecutive zeros)
  - i, j or k can be zero

Case study: parallel sorting algorithm

- Now, let's create a sorting network that sorts a **bitonic sequence**
- A **half-cleaner** network connects line i with line i + n/2

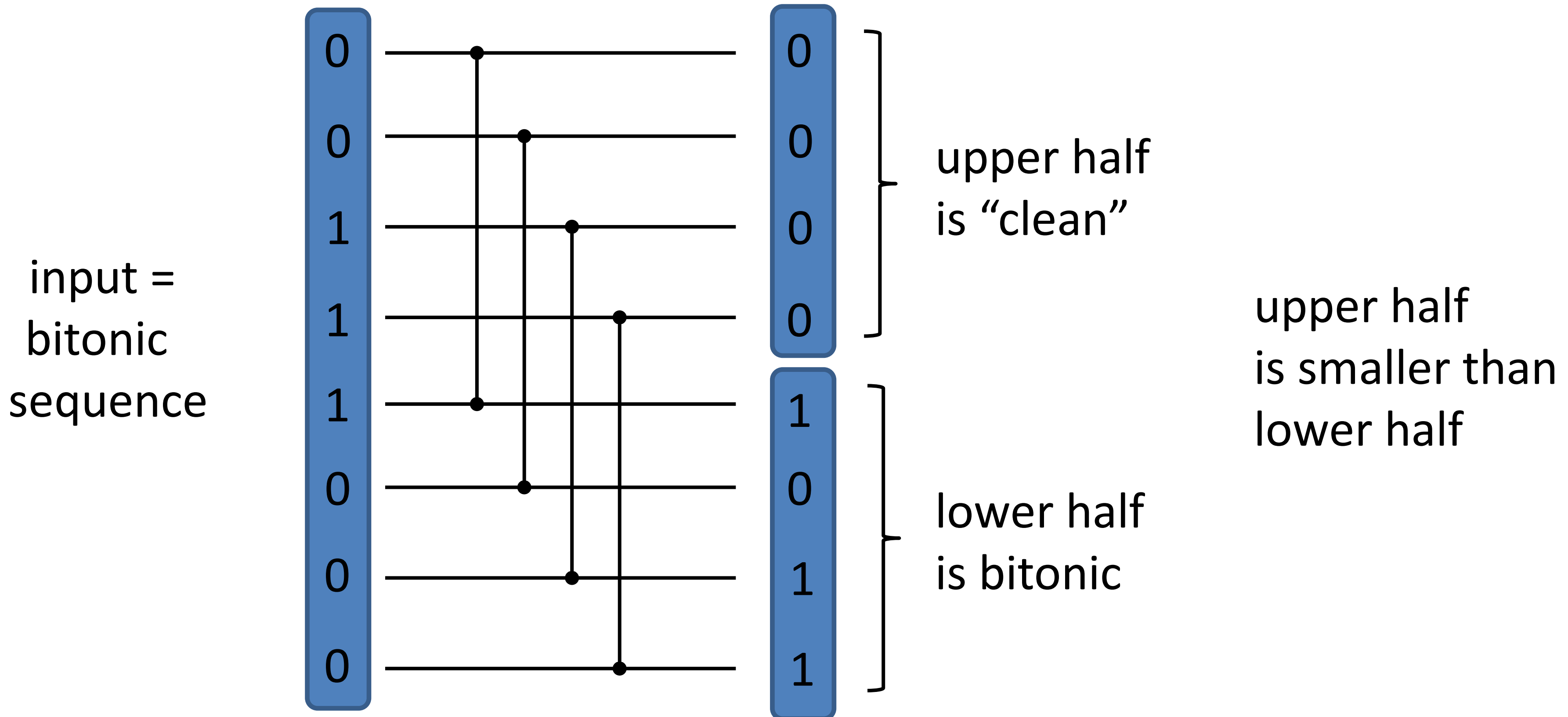
Half-cleaner
example (n=8)



- If the input is a binary bitonic sequence then for the output
  - Elements in the **top half are smaller** than the corresponding elements in the **bottom half,** i.e. halves are relatively sorted.
  - One of the halves of the output consists of **only zeros or ones** (i.e. is "clean"), the other half is bitonic.

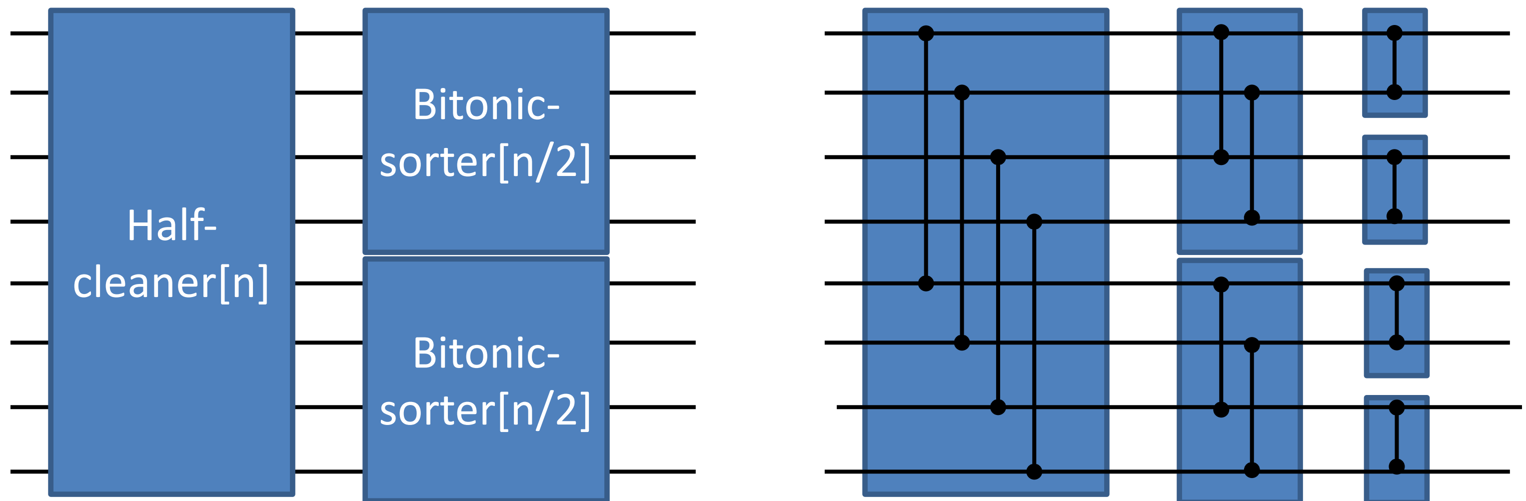
Case study: parallel sorting algorithm

- Example of a **half-cleaner network**



Case study: parallel sorting algorithm

- Therefore, a **bitonic sorter[n]** (i.e. network that sorts a bitonic sequence of length n) is obtained as

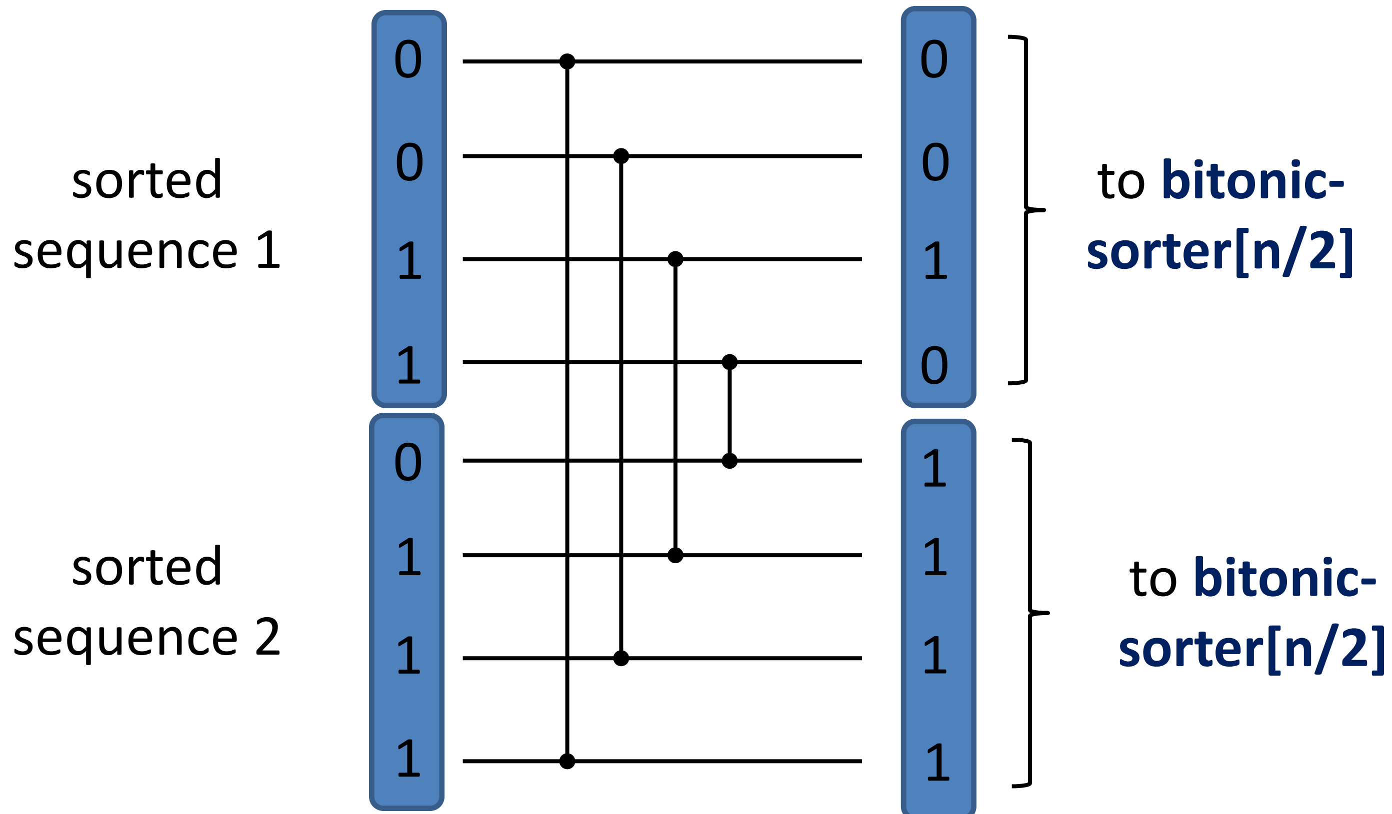


A bitonic sorter sorts a bitonic sequence of length $n = 2^k$ using

- size = $nk/2 = n/2 \log_2 n$ comparators (= sequential time)
- depth = $k = \log_2 n$ (= parallel time)

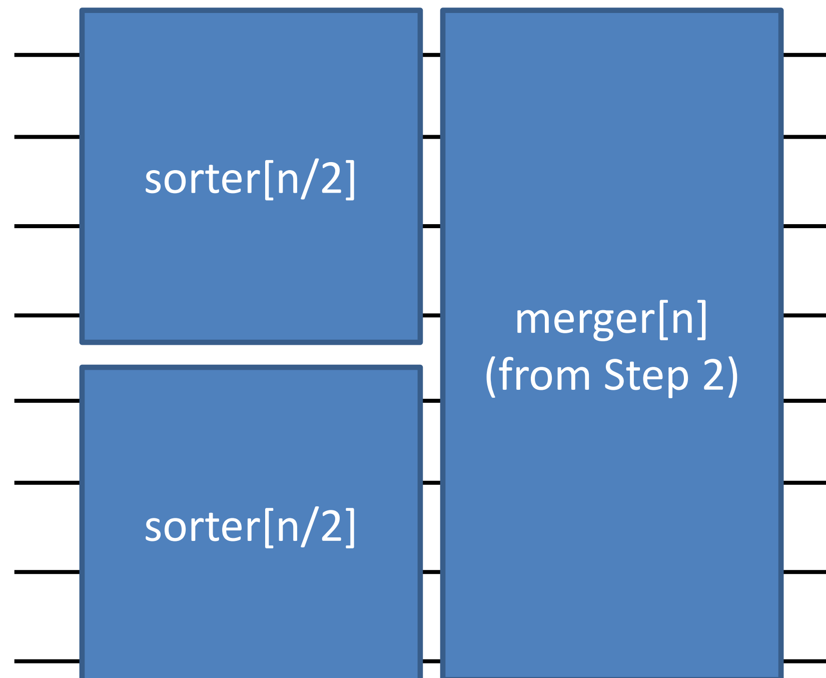
Case study: parallel sorting algorithm

- **Step 2**: Build a network **merger[n]** that merges two sorted sequences of length n/2 so that the output is sorted
  - Flip second sequence and concatenate first and flipped second
  - Concatenated sequence is bitonic, sort using step 1

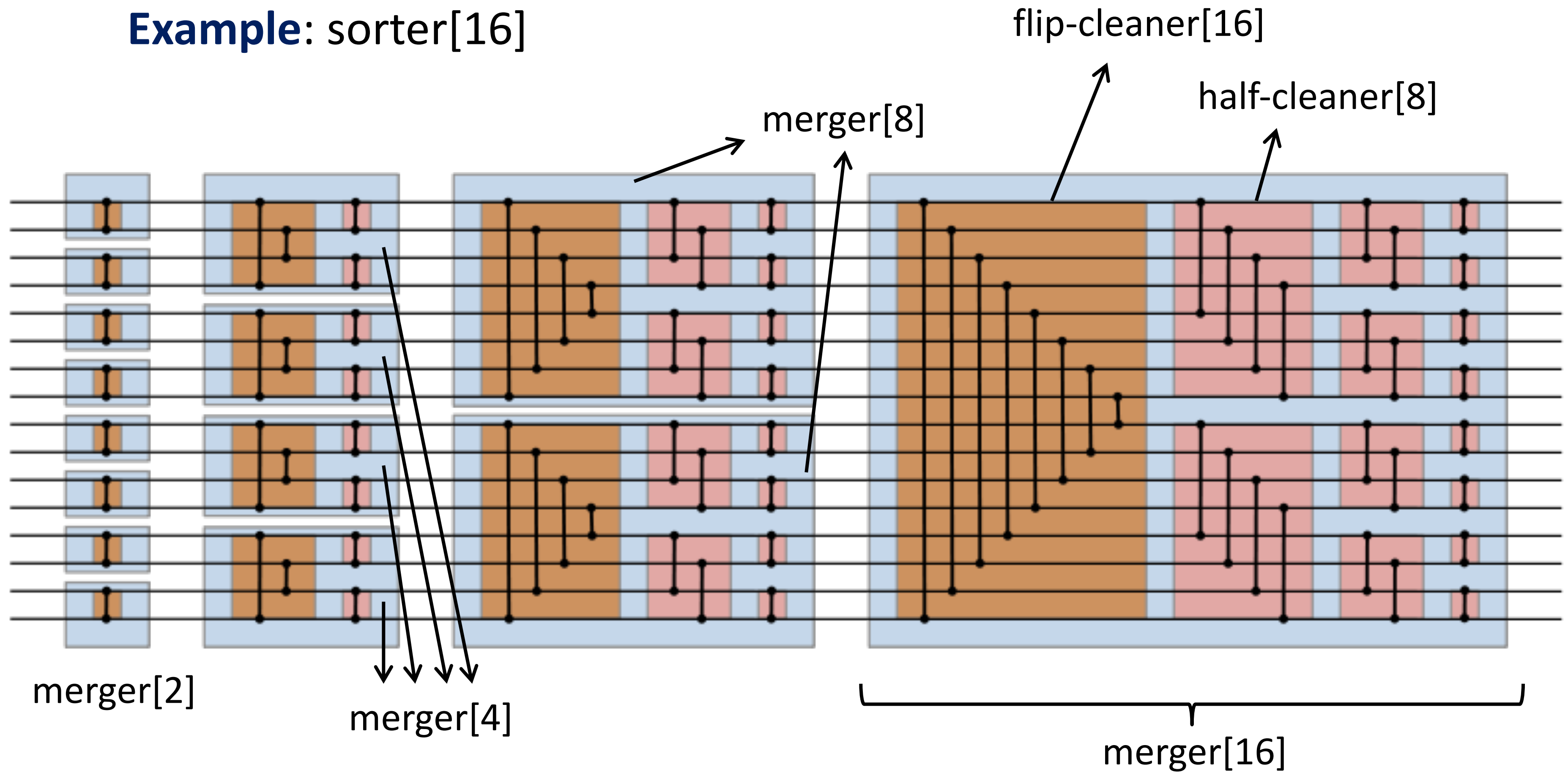


Case study: parallel sorting algorithm

- **Step 3**: Build a **sorter[n]** network that sorts arbitrary sequences
  - Do this recursively from merger[n] building blocks
  - Depth: $D(1) = 0$ and $D(n) = D(n/2) + \log_2 n = O(\log_2^2 n)$



Case study: parallel sorting algorithm



Case study: parallel sorting algorithm

- **Further reading** of bitonic networks:
  - http://valis.cs.uiuc.edu/~sariel/teach/2004/b/webpage/lec/14_sortnet_notes.pdf
- In case **n is not a power of two**:
  - http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm