Introduction to the Message Passing Interface (MPI)

Jan Fostier

May 8th 2013

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

Transistor count

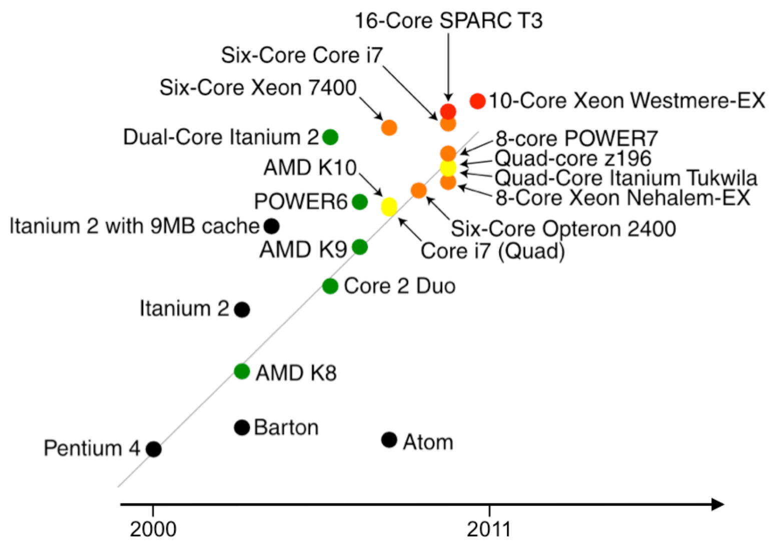
Date of introduction

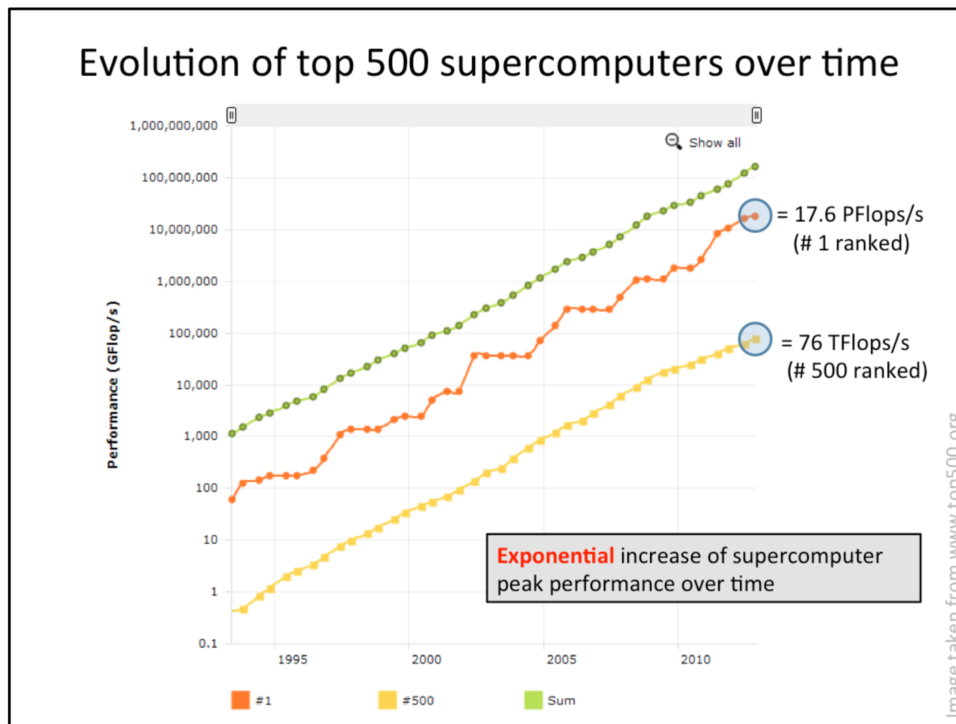
curve shows transistor count doubling every two years

"Transistor count doubles every two years"

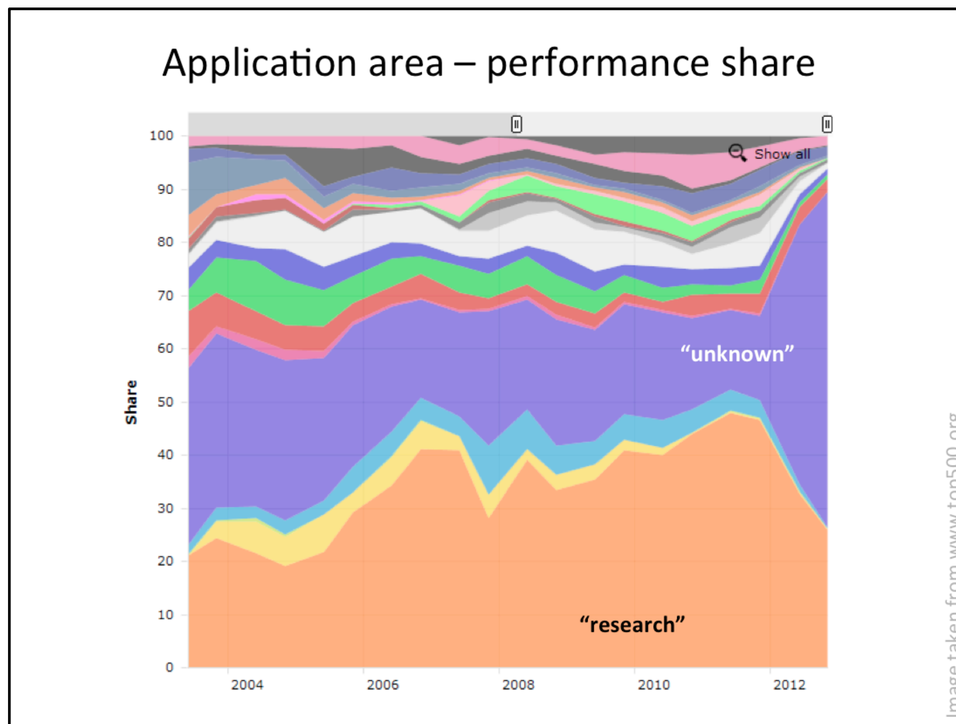
Illustration from Wikipedia

While Moore's law is still valid today (and is project to remain valid until at least 2020), the power of a single CPU chip has progressed mainly by incorporating more and more CPU cores, rather than increasing the speed of a single core.

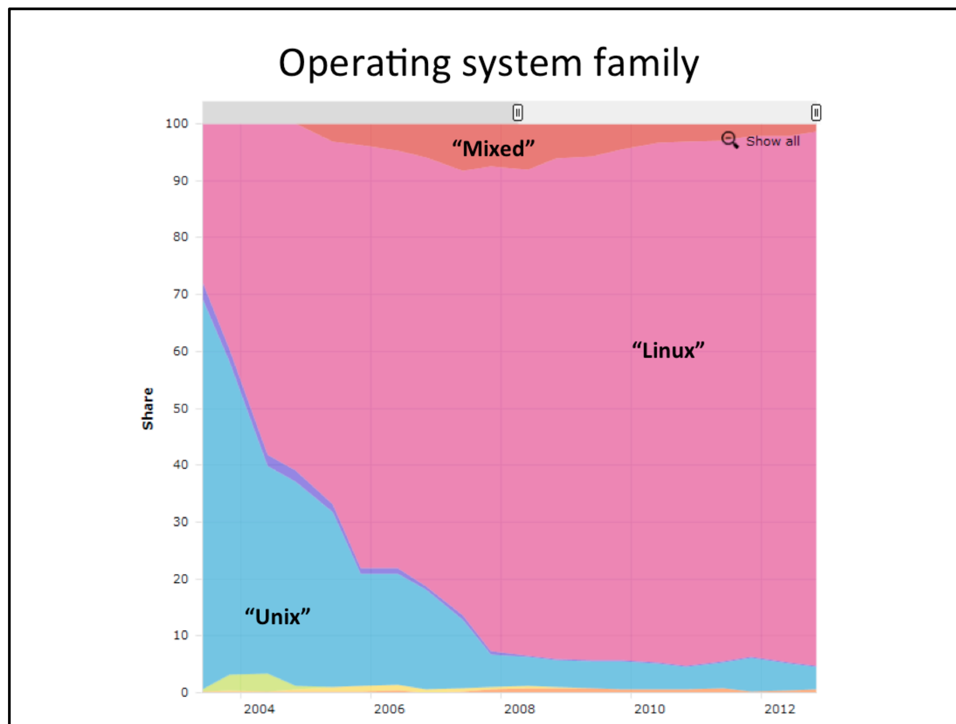

Moore's Law
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
8-Core Xeon Nehalem-EX
AMD K10
POWER6
Itanium 2 with 9MB cache
Six-Core Opteron 2400
Core i7 (Quad)
AMD K9
Core 2 Duo
Itanium 2
AMD K8
Barton
Atom
Pentium 4
2000
2011
Illustration from Wikipedia



In past 15 years, we've seen an exponential growth in the performance of high performance computing systems. For example, the Titan – Cray XK7 is currently ranked the number 1 supercomputer in the world (list of November 2012, see www.top500.org for more recent updates) and has a measured Linpack (linear algebra package) peak performance of 17,59 PFlops/s. The system contains 560640 cores; a combination of 16 core AMD Opteron 6274 CPUs and Nvidia Tesla K20x GPUs.



The different colors represent different application areas over time. A big portion of the supercomputers are used for scientific research. Other application areas are weather and climate simulation, financial models, aerospace, geophysics, etc. Note that in the last update in November 2012, it was unknown for over 60% over the top 500 supercomputers for what kind of purpose they are intended.



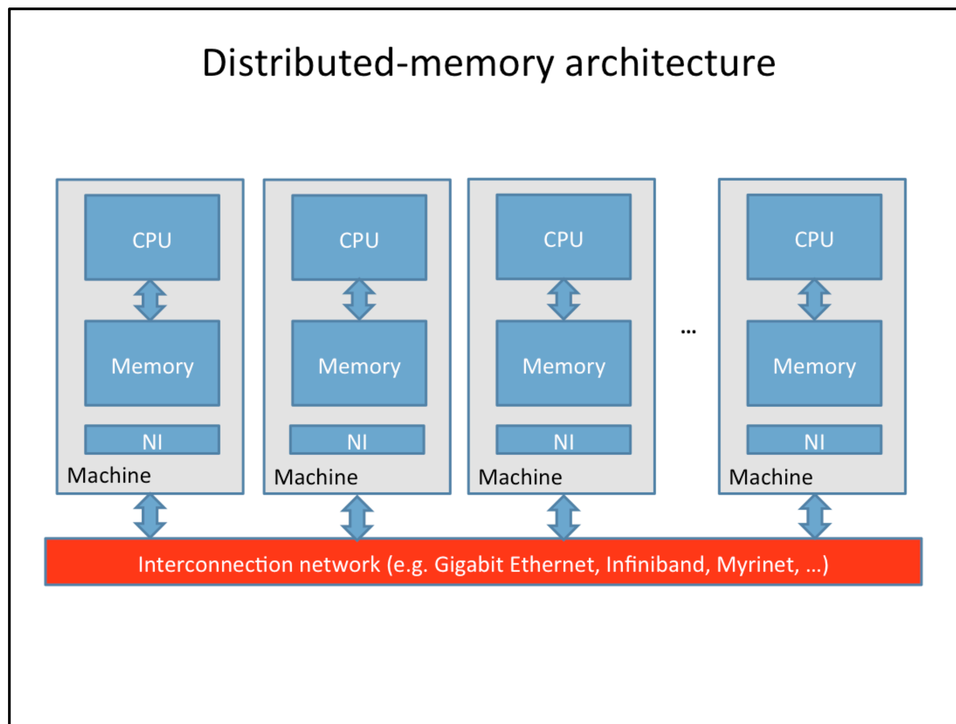
Note that the use of the Linux operating system is dominant on the High Performance Computing market.

Other interesting evolutions can be observed at www.top500.org. For instance, there is recent trend to create hybrid CPU / GPU systems (roughly 10% of the current performance is delivered by incorporating GPUs). Also, Infiniband interconnect technology is predominantly used as in interconnect (~ 45%), next to Gigabit Ethernet (~38%).

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**

SETI @ home = Search for Extraterrestrial Intelligence
Folding @ home = Predict protein folding
GIMPS = Great Internet Mersenne Prime Search (Mersenne prime = prime of the form $2^p – 1$)
R&D = Research & Development
CFD = Computational Fluid Dynamics
CEM = Computational Electromagnetics
Algorithmic trading = buying and selling of financial stock by computers.



In the next chapters, we will focus our attention on parallel computing. We start by considering distributed-memory systems, consisting of a number of machines connected by an interconnection network. In its most simple form, each machine runs a single process. This process has direct access only to the local memory of that machine. In order to access data stored in the memory of another machine, the data must be communicated through the interconnection network using **message-passing**. For this, each machine has at least one **network interface** (NI) or **network interface controller** (NIC) (these are the same things).

In High Performance Computing, the de facto standard for message-passing is the Message Passing Interface (MPI).

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 open implementation (GPL):
  - Open MPI (http://www.open-mpi.org, MPI-2 standard)
- Other important vendor-specific implementations:
  - Qlogic MPI
  - Intel MPI
- 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
```

Source code “helloworld.cpp” on Minerva.

Basic MPI routines

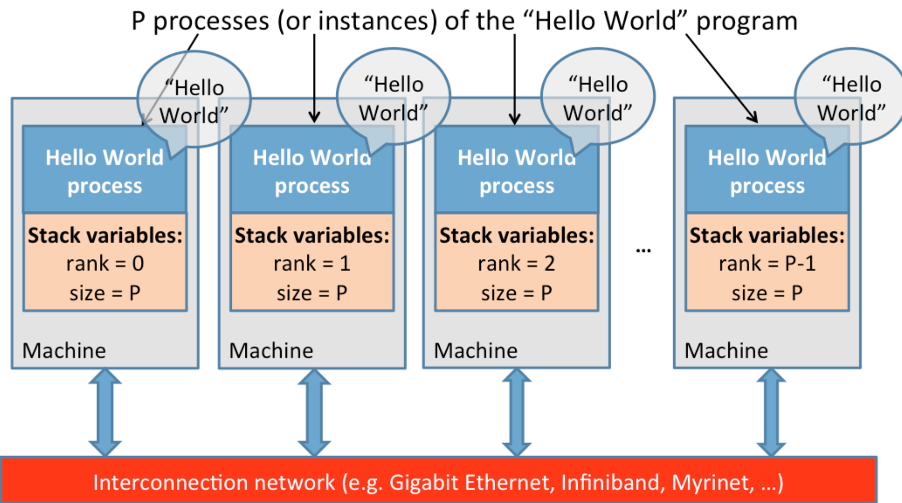
- `int MPI_Init(int *argc, char ***argc)`
  - Initialization: all processes must call this prior to any other MPI routine.
  - Strips of (possible) arguments provided by "mpirun".
- `int MPI_Finalize(void)`
  - Cleanup: all processes must call this routine at the end of the program.
  - All pending communication 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]

MPI standard:

MPI programs require that library routines that are part of the basic language environment (such as printf and malloc in ISO C) and are executed after MPI_Init and before MPI_Finalize. All MPI programs must contain exactly one call to an MPI initialization routine (MPI_Init).

MPI_Finalize cleans up all MPI state. If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then each process must call MPI_Finalize before it exits. Before an MPI process invokes MPI_Finalize, the process must perform all MPI calls needed to complete its involvement in MPI communications.

Message Passing Interface Mechanisms



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

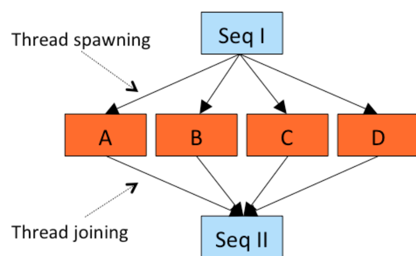
- Each process is a instance of the **same program**

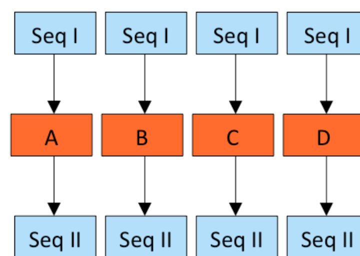
Terminology

- **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 ictce
  - 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

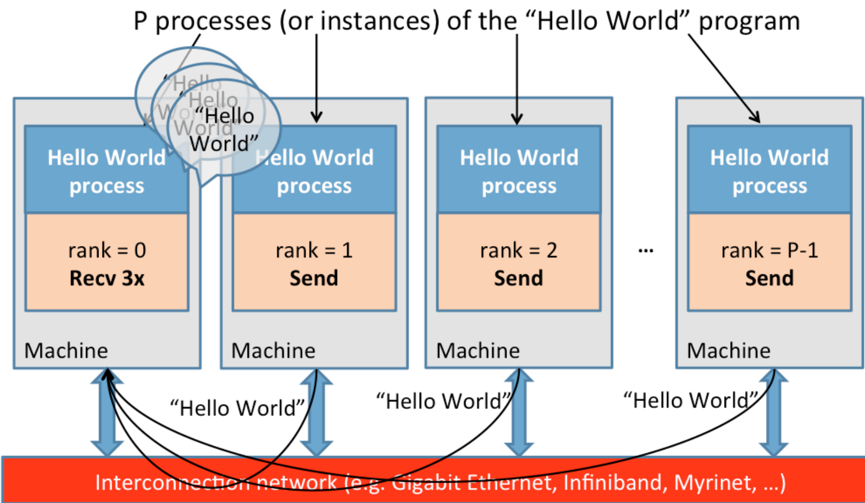
```
...
int rank, size, count;
char b[40];
MPI_Status status;

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

if (rank != 0) {
  MPI_Send("Hello World", 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, cout, 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
```

Source code “ptpcomm.cpp” on Minerva.

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 }

MPI Standard: The send buffer specified by the MPI_Send operation consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of elements, not number of bytes. Note that count may be zero, in which case the data part of the message is empty.

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the message envelope. These fields are

- source
- destination
- tag
- communicator

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation. The message destination is specified by the dest argument. The integer-valued message tag is specified by the tag argument. This integer can be used by the program to distinguish different types of messages.

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

See previous slide.

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

MPI Standard:

One can think of message transfer as consisting of the following three phases.
1. Data is pulled out of the send buffer and a message is assembled.
2. A message is transferred from sender to receiver.
3. Data is pulled from the incoming message and disassembled into the receive buer.

Type matching has to be observed at each of these three phases: (1) The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; (2) the type specified by the send operation has to match the type specified by the receive operation; (3) and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is erroneous.

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the

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;
}
```

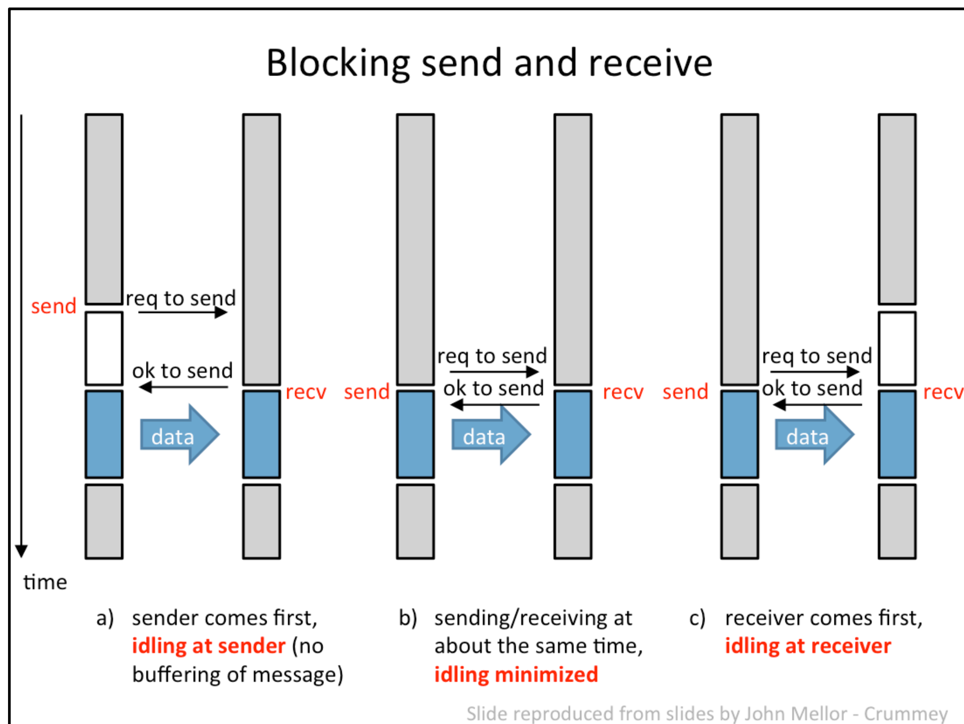
  - Does not contain the size of the received message

- `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

MPI Standard: The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

MPI_Get_count returns the number of entries received. (Again, we count entries, each of type datatype, not bytes.) The datatype argument should match the argument provided by the receive call that set the status variable.



MPI Standard: The send call described before is blocking: it does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, **or** it **might** be copied into a temporary system buffer. In standard mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI **may** buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. **(end of standard citation)** This corresponds to (a) in the slide.

In case the send and receive operation are posted (almost) simultaneous, idling is guaranteed to be minimized (b).

In case a (blocking) receive operation is posted before a matching send is posted (c), the receiver side will block until the message is received.

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

See deadlock.cpp example on Minerva.

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

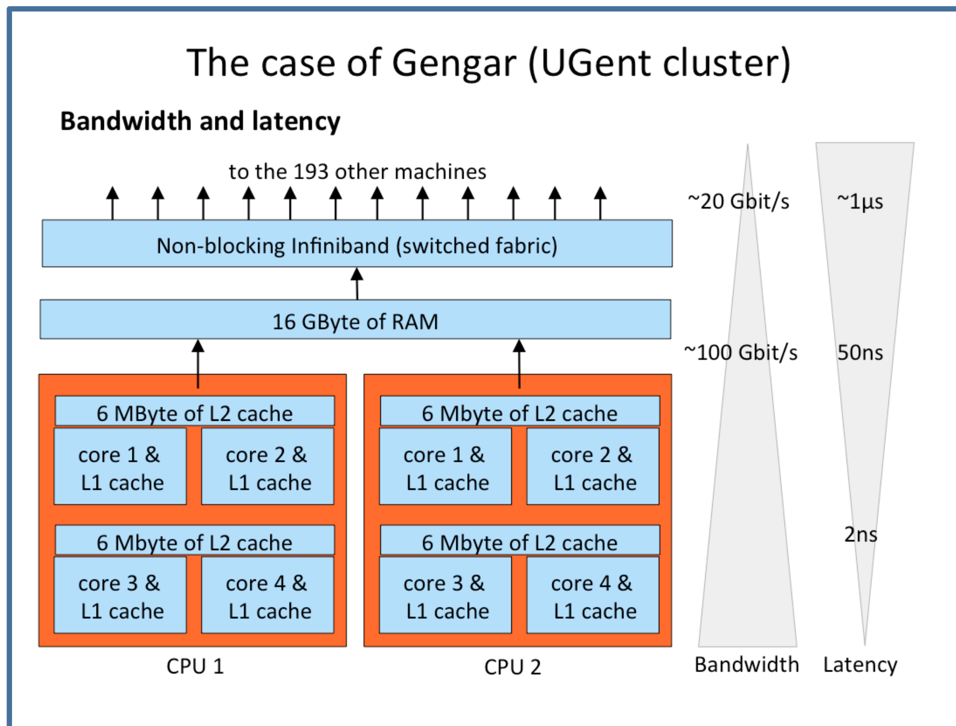
$$T_{comm} = \alpha + \beta * n$$

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

$$B_{eff} = n/a + b*n = n/a + n/B$$

Note that in the most general case, αλπηα and beta depend on the message size.

In case alpha and beta are constant (this is the most simple model), the effective bandwidth is only a function of the message size n. Note that when the latency -> 0 or n -> infinity, the effective bandwidth is equal to the saturation bandwidth B.

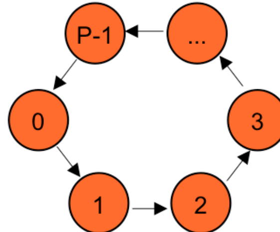


A Gengar machine consists of two quad-core CPUs (Intel Xeon L5420), hence 8 cores in total. Each core has 32 Kbyte L1 cache memory; 6 Mbyte of L2 cache is shared by two cores only. In fact, this is one the earliest quad-core models that was introduced by Intel, and in fact consists of two dual-core CPUs that are packed together. Because e.g. core 1 and core 3 inside a single CPU do not share any cache memory, the exchange of data will always occur by reading / writing to the main memory. On the other hand, core 1 and core 2 share L2 cache, therefore, exchange of data between these cores can happen at a faster rate, by reading / writing to the shared L2 cache.

Each machine has 16 Gbyte of RAM (2 Gbyte / core). The machines are connected by a high-speed Infiniband network (fully non-blocking, see further). Additionally, the machines are connected by a Gigabit Ethernet network (not shown on the slide). This network is used for console login and non-HPC related access (e.g. maintenance).

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 -N SWPC13
#PBS -o output.file
#PBS -e error.file
#PBS -q short/long/debug   (choose one)
#PBS -l nodes=2:ppn=all
#PBS -l walltime=00:02:00
#PBS -m n

cd $VSC_SCRATCH/<yourdirectory>
module load scripts
module load ictce
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

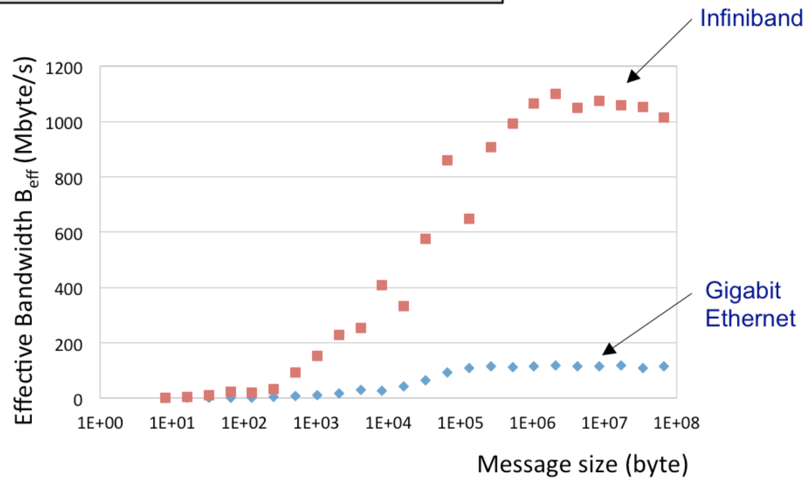
MPI standard

MPI_Wtime returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred. The times returned are local to the node that called them. There is no requirement that different nodes return "the same time.“

MPI_Wtick returns the resolution of MPI_Wtime in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_Wtick should be 10e-3.

Bandwidth on Gengar (Ugent cluster)

Effective BW increases for larger messages



Measured effective bandwidth (unidirectional) using the ringtest example for both the Gigabit Ethernet network and Infiniband network. Measurements were done on the Gengar cluster (each machine has both an Infiniband and a Gigabit Ethernet interface). The effective bandwidth increases for larger message size, in both cases, consistent with the formulas from earlier slides.

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
```

We observed a big difference in available bandwidth between the two networks (previous slide). A second major difference is the fact that the Gigabit Ethernet requires quite a lot of CPU cycles when communication at full speed. These CPU cycles are needed to constantly pack small portions of data into packets and copy them across memory locations (tcp/ip stack protocol overhead). These CPU instructions are encoded in the Gigabit Ethernet device driver. The CPU is constantly signaled by means of interrupt requests to prepare the next portion of data to send or receive. This gives rise to a combined fraction of over 80% of the CPU cycles that are spent in “system time (= device driver)” (sy: 78,3%) and handling software interrupt requests (si: 1,8%). Only 19.8% of the cycles can be spent in user space (actual program).

The Infiniband network on the other hand can transmit at full speed, with very little CPU interaction, leaving 99% of the CPU cycles to the user space. Therefore, Infiniband networks are much more suited to overlap computations and communications (see further: non-blocking send / receive + asynchronous progress).

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( ... )
```

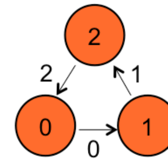
- Buffer is replace by received data

MPI standard:

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

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

MPI_Sendrecv can help prevent deadlocks, because from a programmer's point of view, the send and receive operation are posted simultaneously. It is safe to mix MPI_Send and MPI_Recv on one process with MPI_Sendrecv on another process.

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

```
MPI_Send(...)    MPI_Isend(..., MPI_Request *request)
MPI_Recv(...)    MPI_Irecv(..., status, MPI_Request *request)
```

- 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)

MPI Standard: One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use nonblocking communication. A nonblocking send start call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

Non-blocking communications

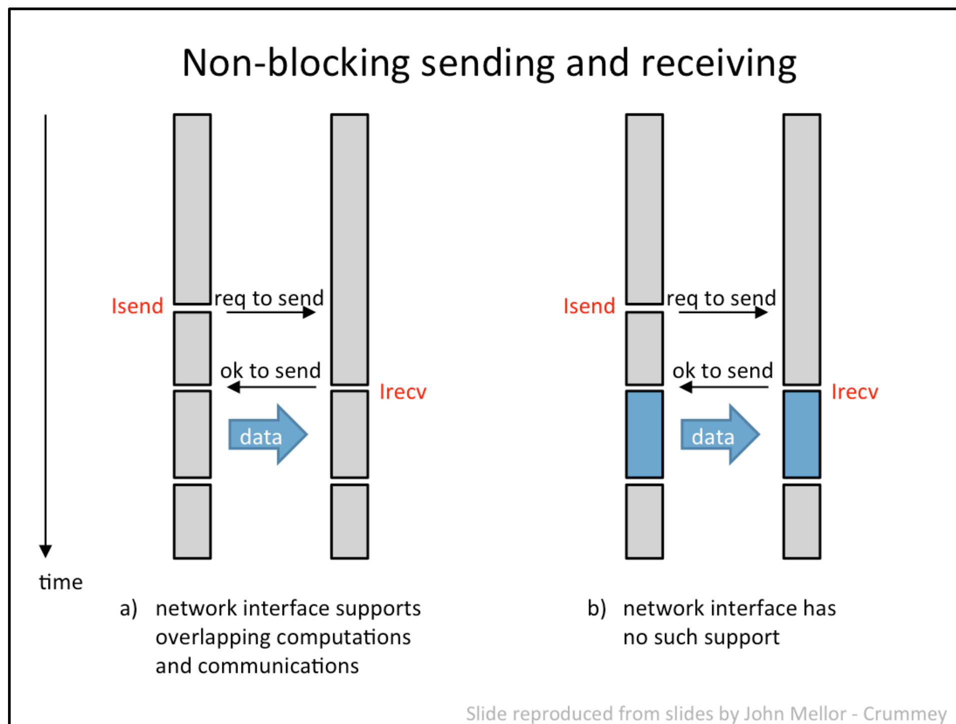
Asynchronous progress

- = 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

Asynchronous progress is the ability of a parallel architecture to overlap communications and calculations. When performing data transfer over a Gigabit Ethernet network, the CPU is actively involved in copying memory from the source buffer to the buffers that can be directly accessed by the network interface (NIC). This is handled by the device driver software of the network (usually a module of the OS kernel) . At high data transfer rates (i.e. close to the maximum bandwidth of one Gigabit per second), the CPU is flooded with interrupt requests, in order to copy a new segment of data into the buffer. This means that the CPU has relatively few cycles left to perform actual computations and the abilities for asynchronous progress are very limited.

Inifiniband on the other hand, implements a so-called **zero-copy protocol**, in which the Inifiniband hardware can directly access a certain memory block (**direct memory access** or DMA). This greatly improves performance, as the CPU is greatly offloaded by this protocol. Nowadays, Open MPI has very good support for Infiniband. On the UGent HPC, vendor-specific implementations are used, e.g. the Intel MPI and Qlogic MPI implementations of MPI. These have full support for all hardware features available.

The **eager protocol** is a data transfer protocol where the data is immediately transmitted to the receiver side, without any prior acknowledgement from the



Left: Gray boxes denote useful computations. Almost no CPU cycles are needed to transfer a message between two nodes. The network interface uses e.g. direct memory access (DMA) to access the send/receive buffers, largely bypassing the CPU.

Right: Blue means that the CPU is actively involved in the data transfer (e.g. by copying data from memory to the network interface). CPU cycles that are spent during communications cannot be used for asynchronous progress. Therefore, overlapping communications and computations possibilities are very limited in this case.

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

MPI Standard: Non-blocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

The functions MPI_Wait and MPI_Test are used to complete a nonblocking communication. The completion of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has completed (but indicates, of course, that the send was initiated).

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to

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] );
    }
}
```

Source code “clientserver2.cpp” is available on Minerva.

In this simple example, all processes produce data of some sort (assume producing data is time-consuming). In a simple while(true) loop, data is produced and sent to the process with rank 0.

A single process (rank == 0) collects all the data and consumes it (assume consuming this data is fast). This process should be able to receive data from any (other) process. This could be implemented using e.g. a blocking receive operation with the MPI_ANY_SOURCE specification, however, in this example, we choose to use non-blocking receive operations. We post a non-blocking receive operation for each of the other processes. These non-blocking receive operations return immediately (before data is actually received). Next, in a while(true) loop, the root process accepts waits for exactly one message to be received. This data is then consumed and a new receive operation is posted for that process. The root process knows which of the pending receive requests was satisfied by the i-index. In this simple example, this index corresponds to the source rank. However, this rank can also be retrieved from the status structure (status.MPI_SOURCE).

A problem with this example is that MPI_Waitany does not guarantee fairness. Because MPI_Waitany waits for exactly one request to be received, MPI_Waitany can

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

MPI_Waitall blocks until **all** communication operations associated with active handles in the list complete, and returns the status of all these operations (this includes the case where no handle in the list is active). MPI_Testall returns flag = true if all communications associated with active handles in the array have completed.

MPI_Waitsome waits **until at least one** of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations.

Example: improved client-server code

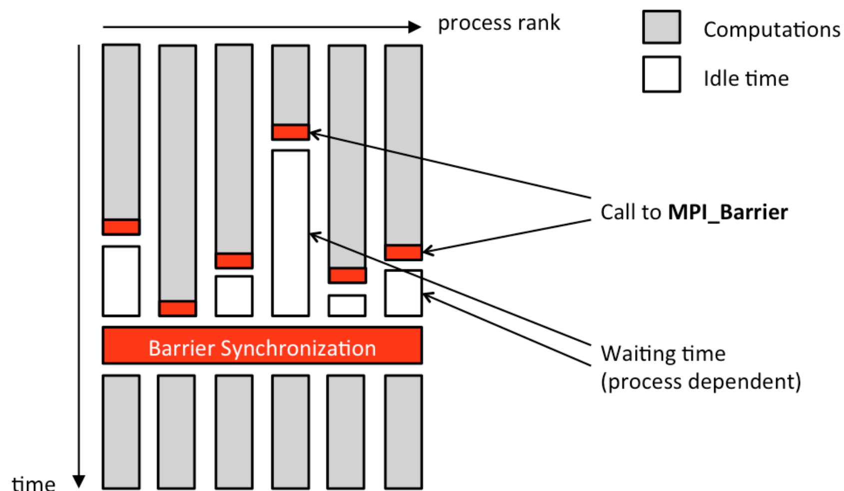
```
if ( rank != 0 ) {      // same client code
    ...
} else {                // server code (rank == 0)
    MPI_Request *reqList = new MPI_Request[nProc-1];
    MPI_Status *status = new MPI_Status[nProc-1];
    int *reqIndex = new MPI_Request[nProc];

    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]] );
        }
    }
}
```

Source code “clientserver3.cpp” is available on Minerva.

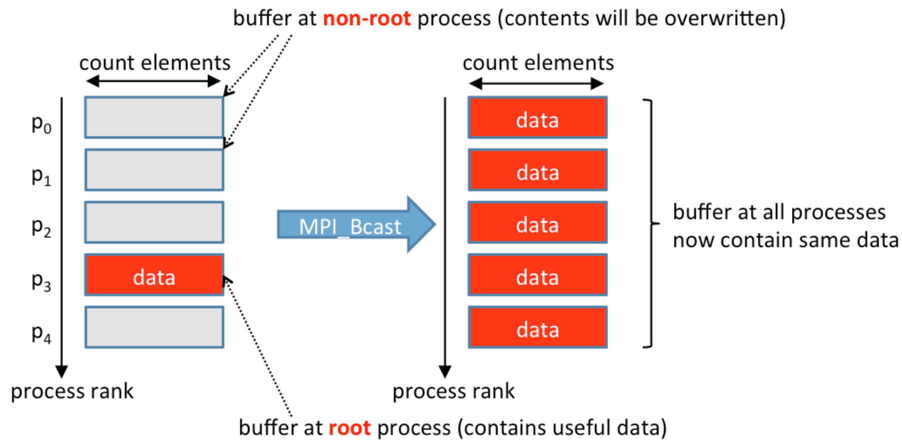
Same client-server example. This time however, MPI_Waitsome is used instead of MPI_Waitany. This guarantees that every client will eventually be provided by the server side (fairness).

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



Blocks the caller until all processes in the communicator have called it; that is, the call returns at any process only after all members of the communicator have entered the call.



MPI_Bcast broadcasts "`count`" elements of type "datatype" from root to all other processes in comm. **All processes in `comm` have to call this function** (both root and non-root processes). Before calling the function, the buffer at the root process must contain useful data. Hence, at the root process, this buffer acts as a send buffer. This data will be transmitted to the other processes and will remain unchanged. The buffers at the other processes acts as a receive buffer; it needs to be pre-allocated, any contents will be overwritten by the data at the root process.

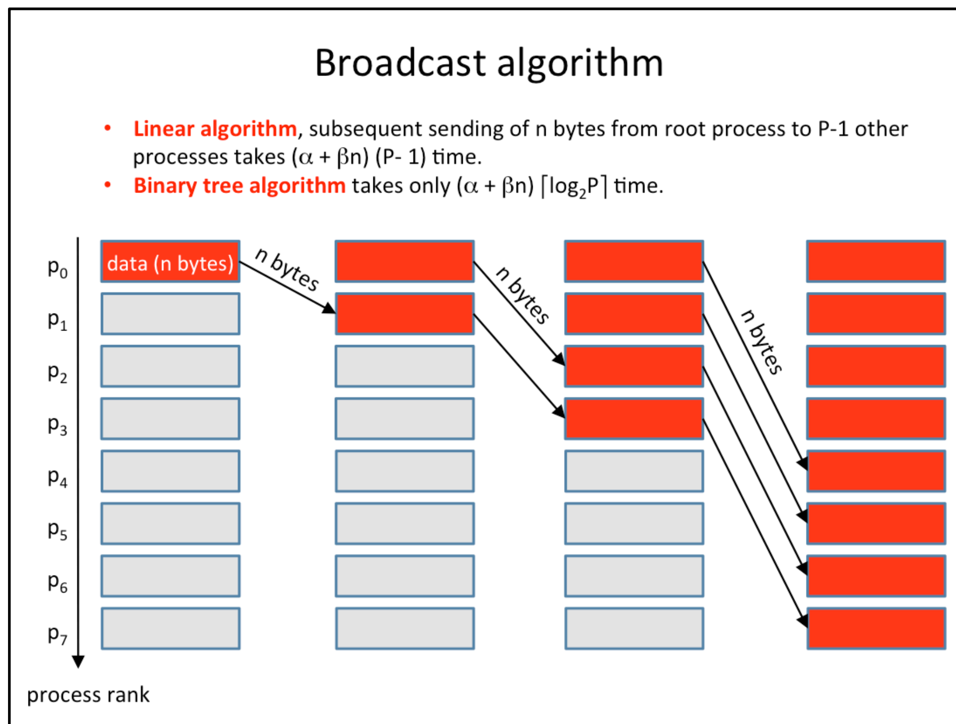
Broadcast example

```
...
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.
```



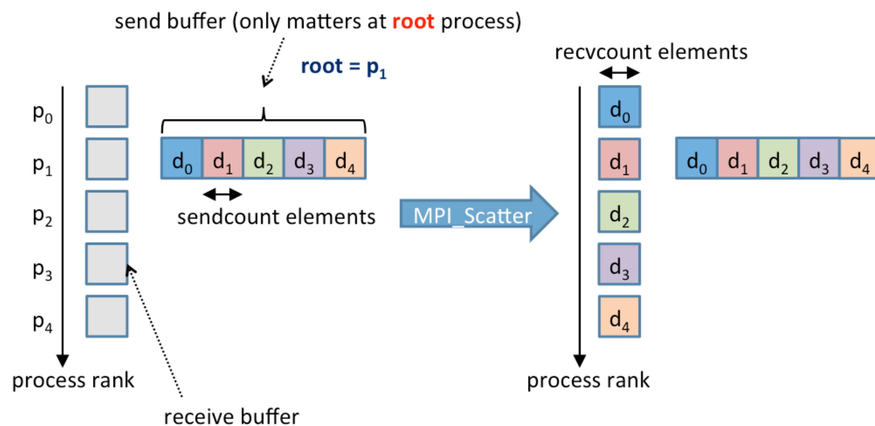
MPI_Bcast broadcasts a message from the process with rank root to all processes of the group. It is called by all members of group using the same arguments for comm and root. On return, the contents of root's communication buffer has been copied to all processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_Bcast and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

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.



MPI_Scatter is the inverse operation to MPI_Gather.

The outcome is *as if* the root executed n send operations:

MPI_Send(sendbuf + i * sendcount * extent(sendType), sendcount, sendType, i, …)

and each process executed a receive:

MPI_Recv(recvbuf, recvcount, recvType, i, …)

The send buffer is ignored for all non-root processes.

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The arguments root and comm must have identical values on all processes.

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);

...
```

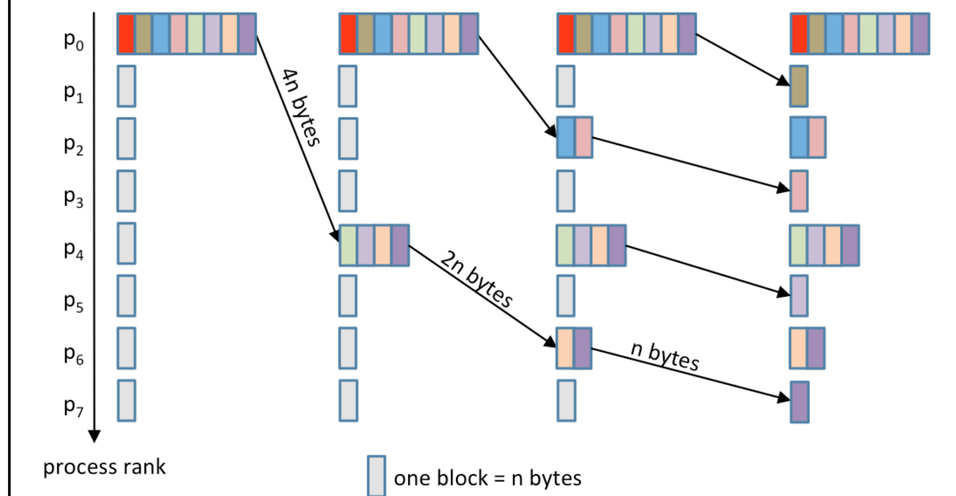
fill the send buffer at the root process only

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.
```

See scatter.cpp on Minerva.

The code shown in this slide is a simplification of scatter.cpp and will fail when running it with more than 4 processes, because the send buffer is too small. The example on Minerva is more generic in that sense.

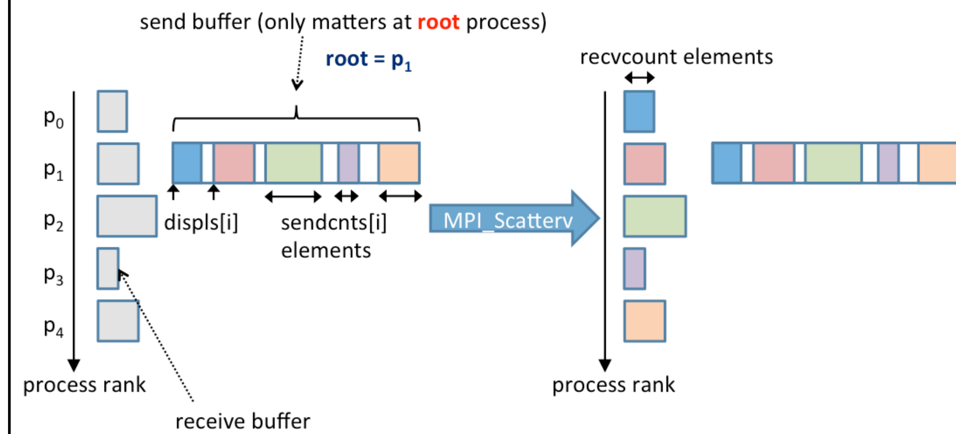


Note that the binary algorithm does not reduce the amount of data sent from the root process, compared to the linear algorithm. It does however, reduce the number of communication rounds from (P-1) to $\log_2 P$. Note however that certain data need to be buffered by certain non-root processes. This buffer is not provided by the used and must therefore be allocated by the MPI implementation itself.

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.



MPI_Scatterv extends the functionality of MPI_Scatter by allowing a varying count of data to be sent to each process, since sendcnts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing the new argument, displs.

The outcome is as if the root executed P send operations

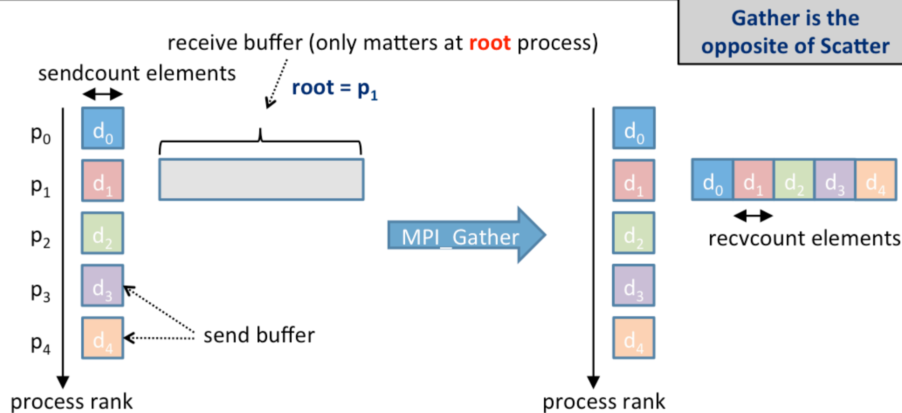
MPI_Send(sendbuf + displs[i] * extent(sendType), sendcnts[i], sendType, i, ...)

and each process executed a receive

MPI_Recv(recvbuf, recvcnt, recvType, i, ...)

The send buffer is ignored for all non-root processes.

The type signature implied by sendcnt[i], sendType at the root must be equal to the type signature implied by recvcnt, recvType at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.
All arguments to the function are significant on process root, while on other



Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the P processes in the group (including the root process) had executed a call to

MPI_Send(sendbuf, sendcount, sendtype, root, …)

and the root had executed P calls to

MPI_Recv(recvbuf + i * recvcount * extent(recvType), recvcount, recvType, i, …)

where extent(recvType) is the type extent obtained from a call to MPI_Type_extent().

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcnt, sendtype on process i must be equal to the type signature of recvcnt, recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed. All arguments to the function are significant on process root, while on other processes, only arguments

Gather example

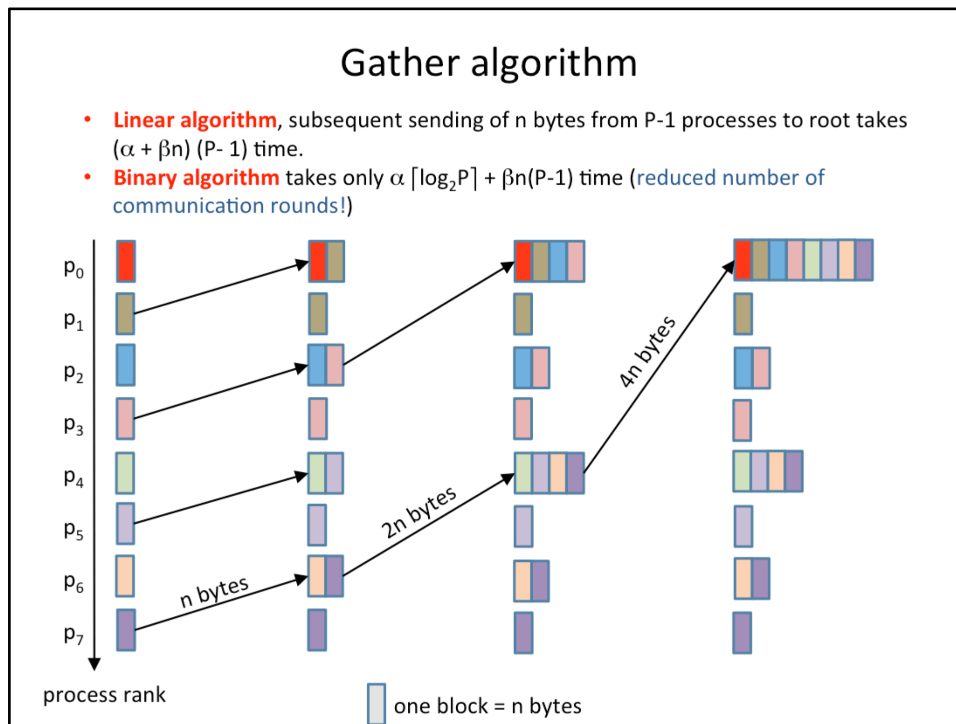
```
int root = 0;
int sendBuf = rank;

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
```

See gather.cpp on Minerva.

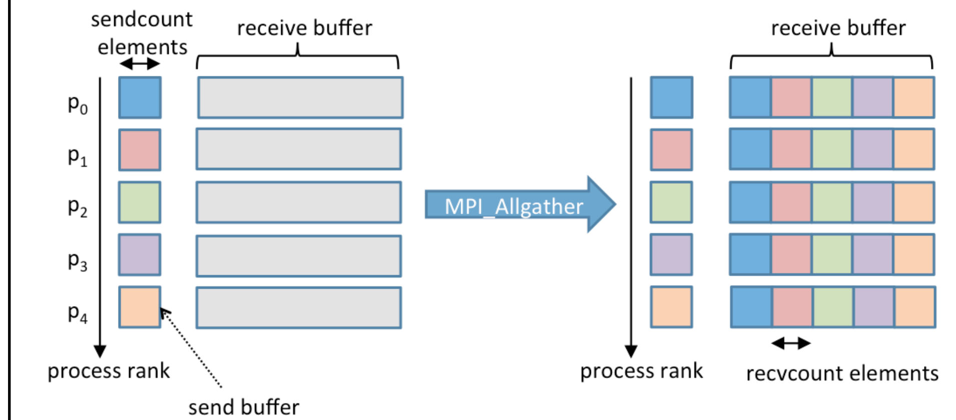


Note that the binary algorithm does not reduce the amount of data sent to the root process (this is dominant for large messages). It does however, reduce the number of communication rounds from (P-1) to $\log_2 P$. Note however that certain data need to be buffered by certain non-root processes. This buffer is not provided by the used and must therefore be allocated by the MPI implementation itself.

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.



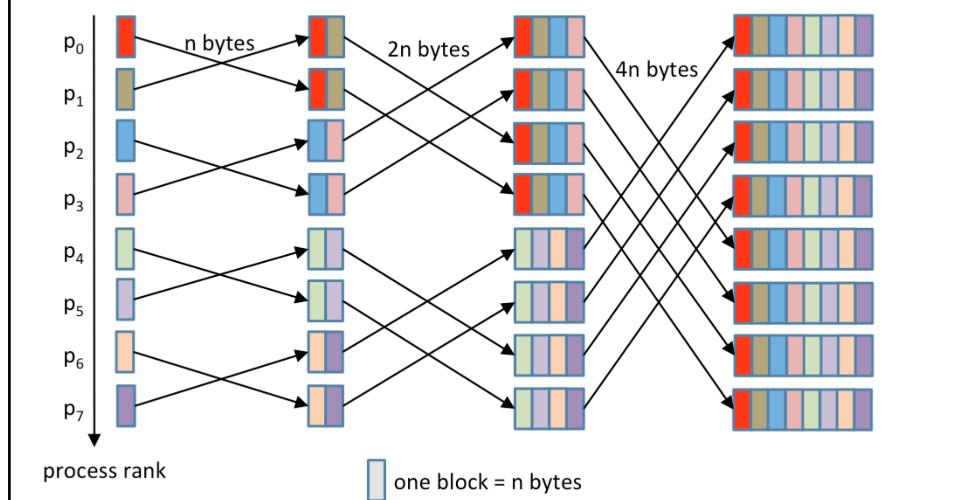
MPI_Allgather can be thought of as MPI_Gather, but where all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.
The outcome of a call to MPI_Allgather(...) is as if all processes executed P calls to

MPI_Gather(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)

for root = 0, …, P – 1.

The rules for correct usage of MPI_Allgather are easily found from the corresponding rules for MPI_Gather.



There are several ways to implement an allgather algorithm. A naïve solution would be to call the gather routine P times, each time with a different process as root. This is very inefficient as this leads to a dominant term of n*P*P.

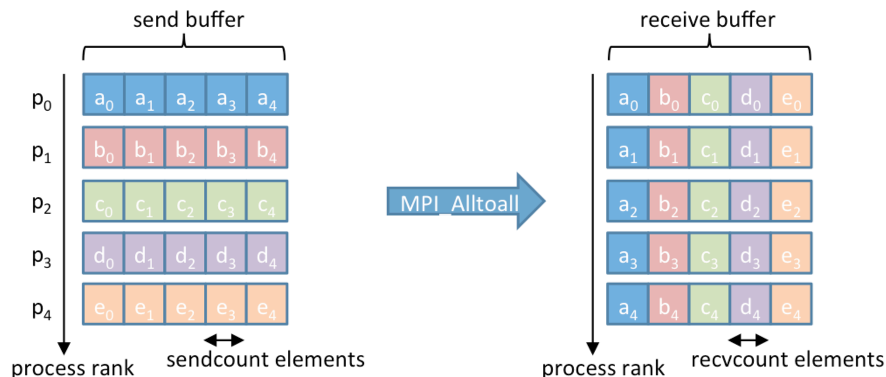
An improvement is to perform a single gather operation, followed by a broadcast from the root. This lead to a dominant term of n*P*log P.

However, a dedicated algorithm, based on a butterfly communication pattern leads to a domination term of only n*P.

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

MPI_Alltoall is an extension of MPI_Allgather to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

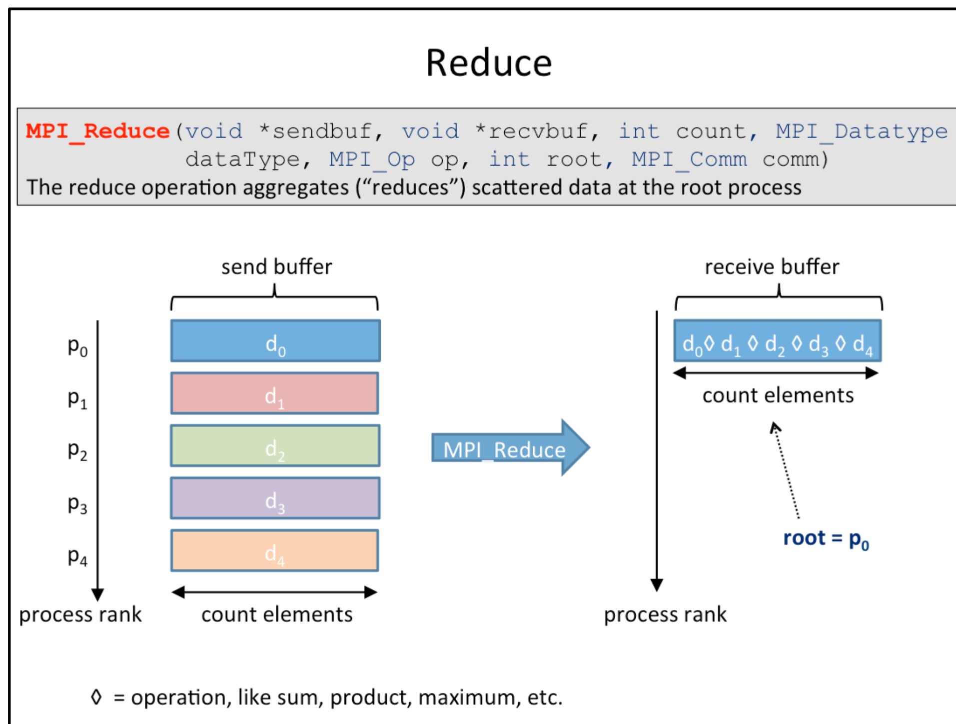
The outcome is as if each process executed a send to each process (itself included) with a call to

MPI_Send(sendbuf + i * sendcnt * extent (sendType), sendcnt, sendType, i, …)

and a receive from every other process with a call to

MPI_Recv(recvbuf + i * recvcnt * extent (recvType), recvcnt, recvType, i, …)

All arguments on all processes are significant. The argument comm must have identical values on all processes.



MPI_Reduce combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type.

The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, **all processes** provide input buffers **and output buffers** of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then

recvbuf[0] = global_maximum(sendbuf[0])

and

recvbuf[1] = global_maximum(sendbuf[1])

The operation op is always assumed to be **associative**. All predefined operations are

Reduce operations

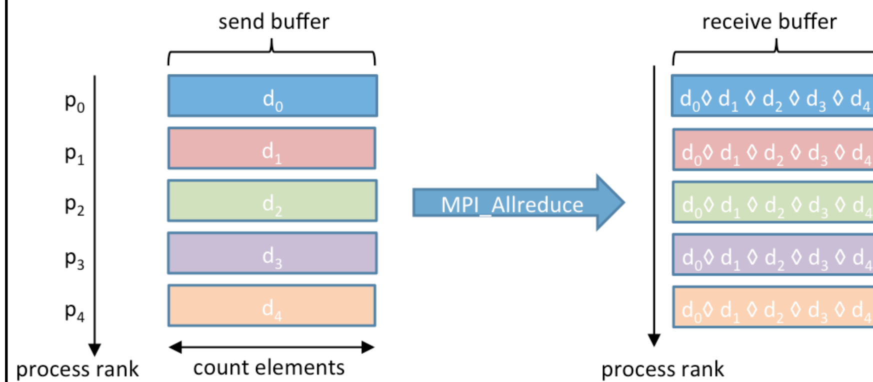
Available reduce operations (associative and commutative)
User defined operations are also possible

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUN	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.

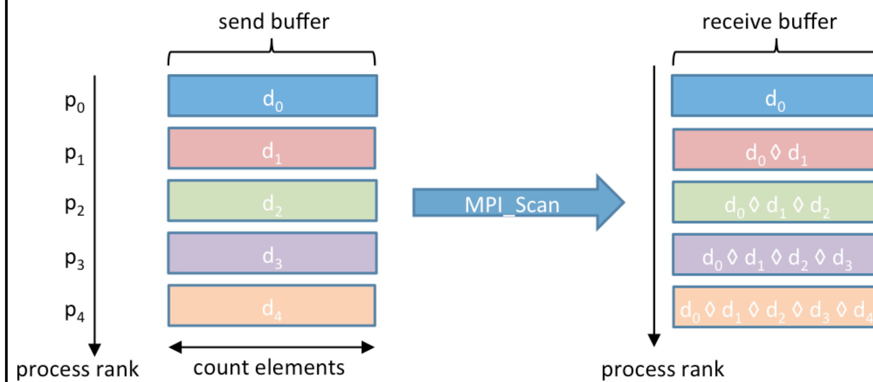


Same as MPI_Reduce except that the result appears in the receive buffer of **all** the group members.

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



MPI_SCAN is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks [0,...,i]. The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

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

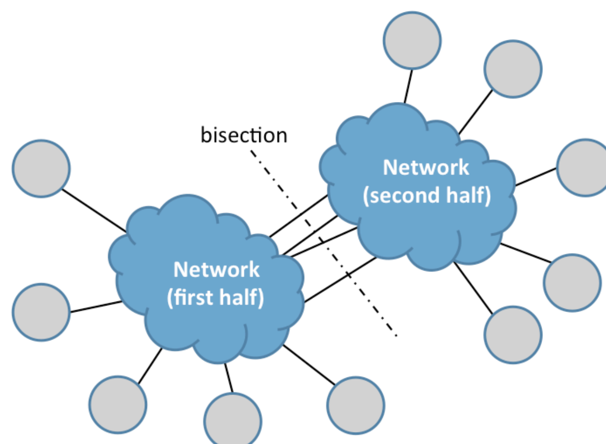
```
// initialize a and b
for(int j=0;j<cols;j++) {b[j]=1.0; for(int i=0;i<rows;i++) a[i][j]=i;}
// broadcast b to each slave
MPI_Bcast( b, cols, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD );
// send row of a to each slave, tag = row number
int numsent = 0;
for( int i = 0; (i < numprocs-1) && (i < 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



We have created simple model for point-to-point communication bandwidth in earlier slides. However, this model is not always valid in case several devices are communication mutually. For example, consider 4 machines and two scenarios:

a) Machine 0 is communicating with machine 1, while machine 2 and 3 are not communicating.
b) Machine 0 is communicating with machine 1, and simultaneously, machine 2 is communicating with machine 3.

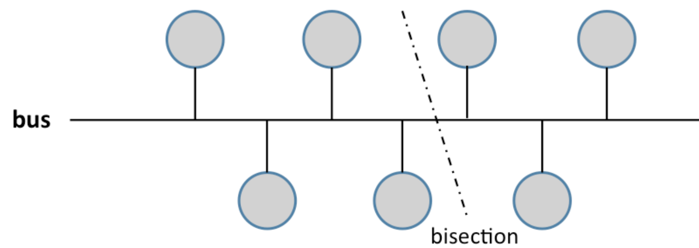
In scenario a), machine 0 and 1 can communicate at their full (effective) bandwidth. However, in scenario b), this bandwidth might be lower, because two communications are occurring simultaneously, and they might share a common communication path in the network. Therefore, we need to study and understand the network topology which is connecting the different machines. Note that not only the bandwidth might be influenced by the network, also the latency depends on the network, as a different number of "hops" might be required, depending on which two machines are communicating. Therefore, two measurements are important:

Bisection bandwidth: Given an interconnection network connecting N machines represented as a graph, the bisection bandwidth is defined as the sum of the bandwidths corresponding to the minimum number of edges (i.e. communication

Bus topology

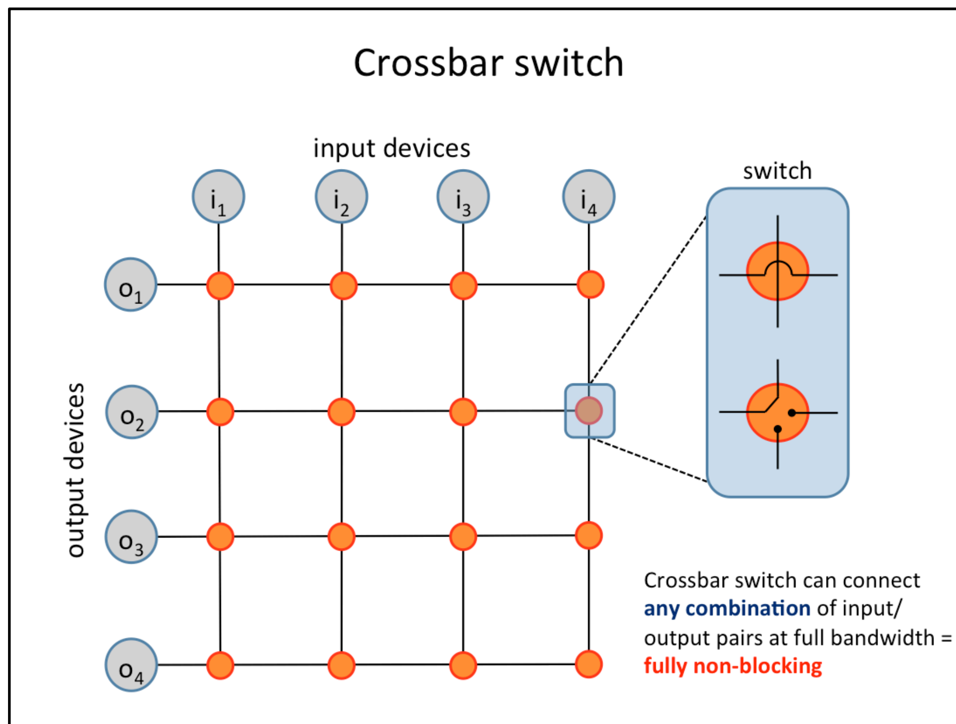
Bus = communication channel that is **shared** by all connected devices

- 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.



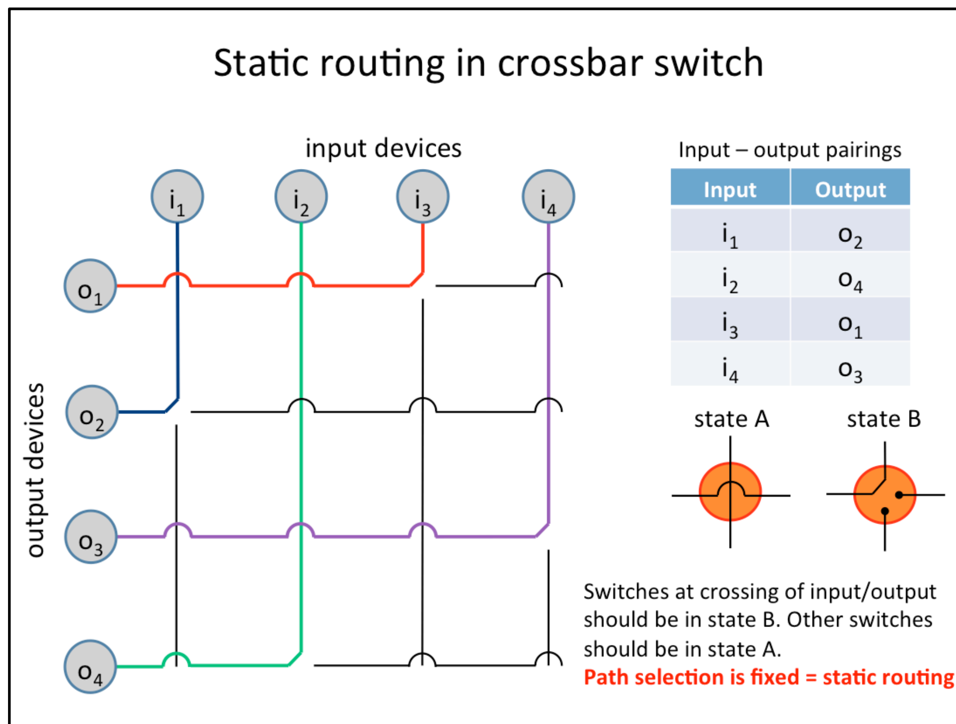
Bus network properties

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



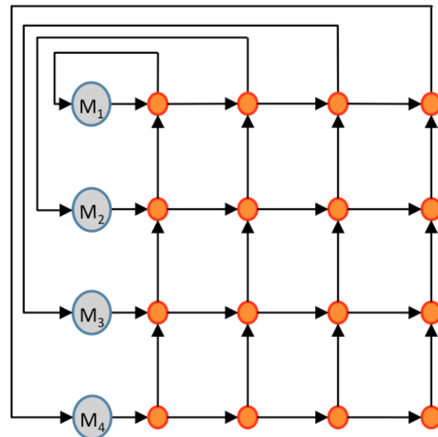
A crossbar switch divides the communicating devices into two groups: “input devices” and “output devices”, even though in the general case, the links are bidirectional and communication can flow in both directions between any “input” / “output” pair. For example, suppose that the input devices are processors, and the output devices are memory modules. Each processor can then have bidirectional communication with any memory module. Note that using this topology, processors cannot mutually communicate. The same is true for the memory modules.

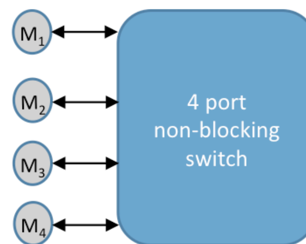
An important property of the crossbar switch is that, as long as an output device is not accessed by two processors simultaneously, any input device can communicate at full speed with any output device, regardless of other ongoing communication. This is a very beneficial property of a network. The network is said to be **fully non-blocking**. The bisection bandwidth is then BP/2, where B is the point-to-point saturation bandwidth and P is the number of connected devices.



In a crossbar switch, the routing process is static, that is, a given input / output communication is always routed over the same physical wires, regardless of the other communicating processes.

Crossbar switch for distributed memory systems

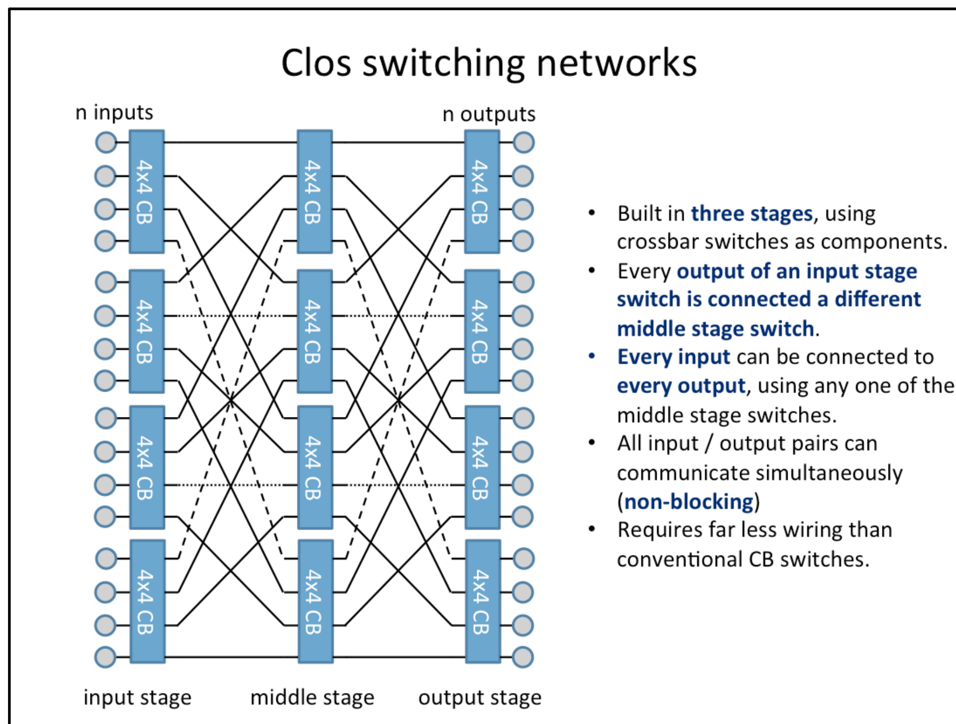




- Crossbar implementation of a switch with P = 4 machines.
  - **Fully non-blocking**
  - **Expensive: P^2 switches and $O(P^2)$ wires**

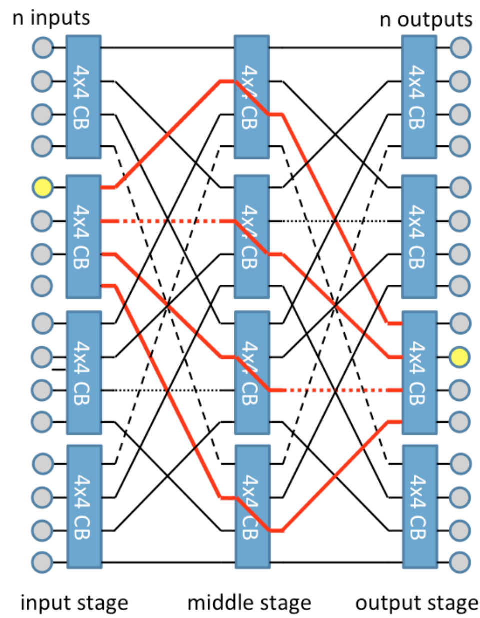
In a distributed memory environment, the separation of devices into "input" and "output" devices is not desirable. A crossbar switching network however, can easily be modified. Note that the directed arrows now denote unidirectional communication links. The network is again fully non-blocking. Note that in reality, the switching network is usually contained in a so-called switch. The switch then has a number of ports (typically 24 to 96) to which machines can be connected by means of a cable interconnect (copper or optical cable).

Note that even though a crossbar switch has attractive properties, because P^2 switches are required (and the same order of magnitude of cables connecting these switches) crossbar switches are not directly used when dealing with a high number of devices. They do however serve as building blocks for other, fully non-blocking switching networks (see next slide).



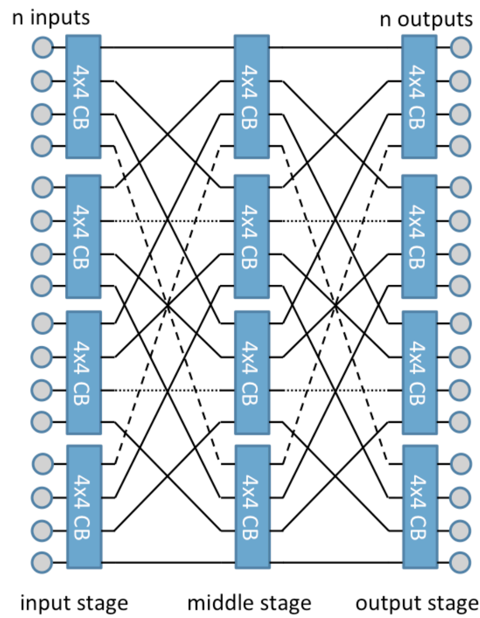
Clos networks have three stages: the input stage, middle stage, and the output stage. Each stage is made up of a number of crossbar switches (see diagram below), often just called *crossbars*. Each call entering an ingress crossbar switch can be routed through any of the available middle stage crossbar switches, to the relevant egress crossbar switch. A middle stage crossbar is available for a particular new call if both the link connecting the ingress switch to the middle stage switch, and the link connecting the middle stage switch to the egress switch, are free.

Clos switching networks



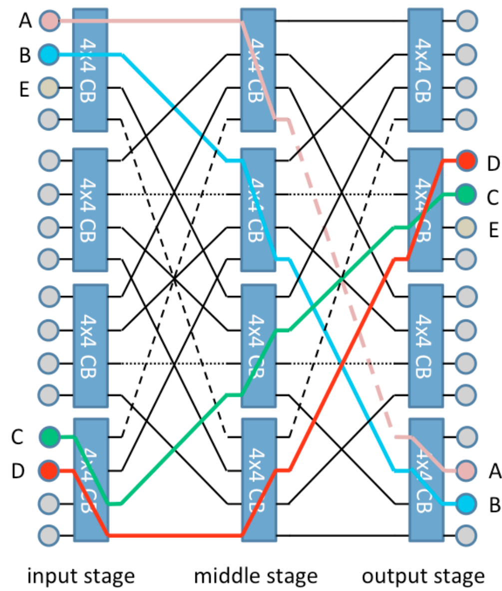
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate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

Clos switching networks



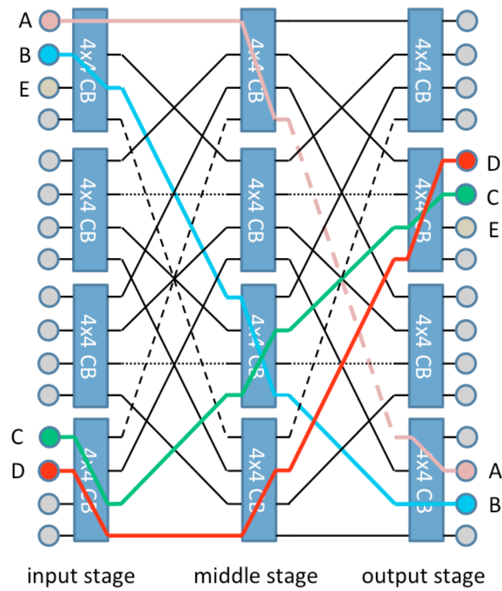
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate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

Clos switching networks



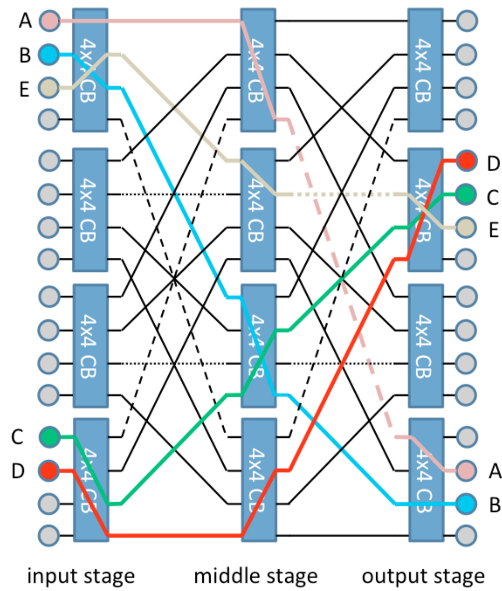
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate 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



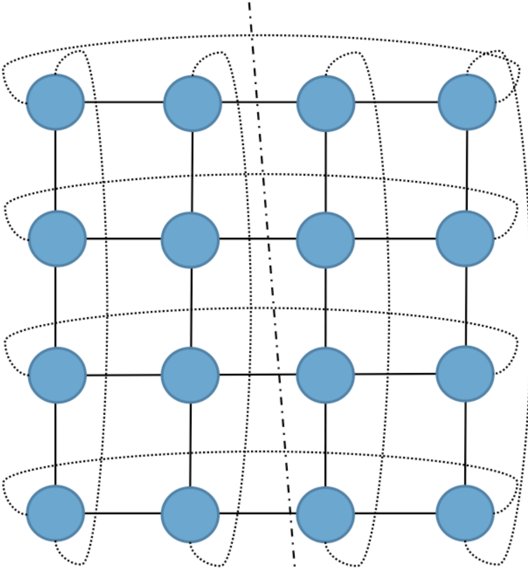
- Built in **three stages**, using crossbar switches as components.
- Every **output of an input stage switch is connected a different middle stage switch**.
- **Every input** can be connected to **every output**, using any one of the middle stage switches.
- All input / output pairs can communicate 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

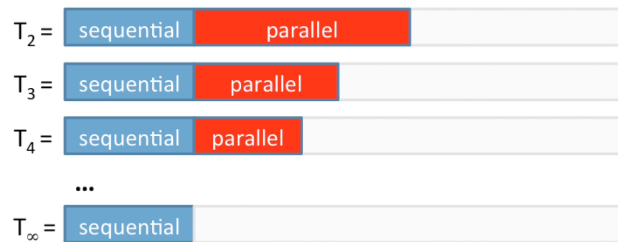
Basic performance terminology

- **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**
  - T_P = runtime using P processes
- **Speedup** ("*How much faster does my program run in parallel*")
  - $S_P = T_1 / T_P$
  - 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*")
  - $\eta_P = S_P / P$ $\quad (0 \leq \eta_P \leq 1)$
  - In the ideal case, $\eta_P = 1 = 100\%$
  - Depends on the application what is considered acceptable efficiency

Strong scaling: Amdahl's law

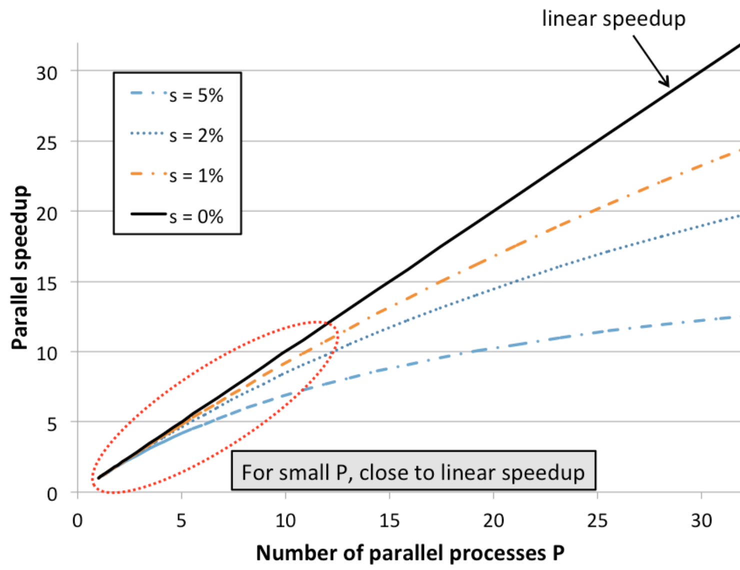
- **Strong Scaling** = increasing the number of parallel processes for a fixed-size problem
- Simple model: partition sequential runtime T_1 in a parallelizable fraction (1-s) and inherently sequential fraction (s).
  - $T_1 = sT_1 + (1\text{-}s)T_1$ $(0 \leq s \leq 1)$

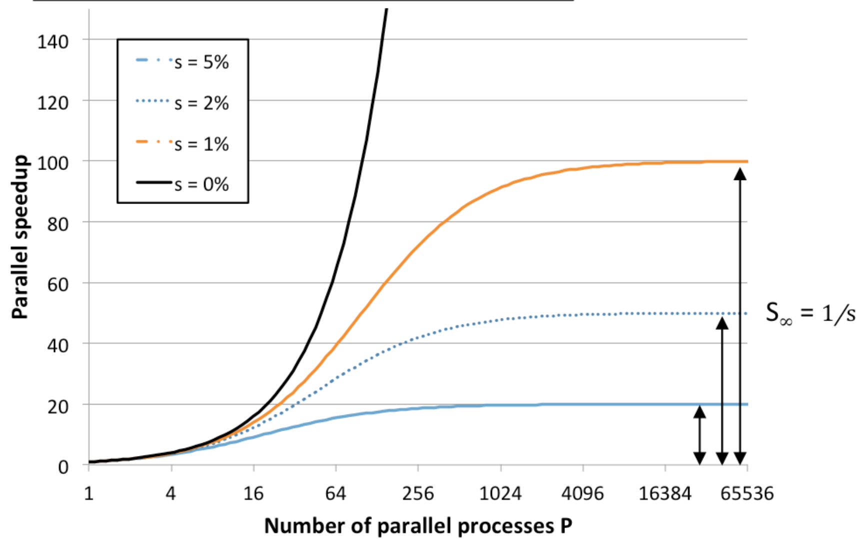
  - Therefore, $T_P = sT_1 + (1-s)T_1/P$

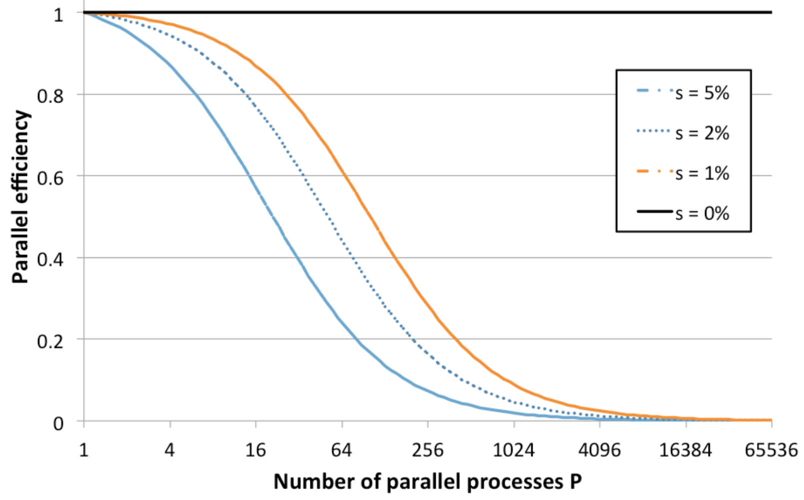


Strong scaling: Amdahl's law

- Consequently, $S_P = T1/TP = T1/sT1+(1-s)T1/P = 1/s+1-s/P$ (**Amdahl's law**)
- Speedup is **bounded** $S_\infty = 1/s$
  - 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

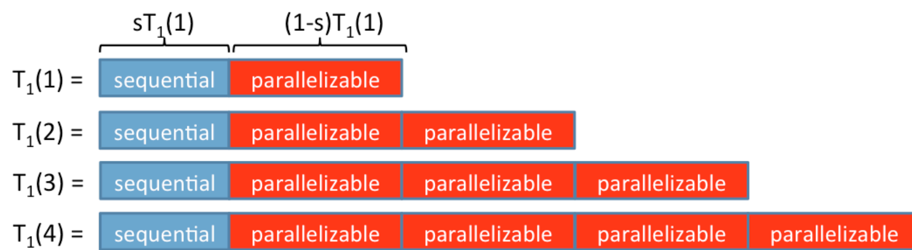

Amdahl's law
linear speedup
s = 5%
s = 2%
s = 1%
s = 0%
Parallel speedup
30
25
20
15
10
5
0
0
5
10
15
20
25
30
Number of parallel processes P
For small P, close to linear speedup


Amdahl's law
Same graph as on previous slide, but logarithmic x-as
s = 5%
s = 2%
s = 1%
s = 0%
Parallel speedup
0
20
40
60
80
100
120
140
1
4
16
64
256
1024
4096
16384
65536
Number of parallel processes P
S∞ = 1/s


Amdahl's law
Same graph as on previous slide, but expressed as parallel efficiency
Parallel efficiency
1
0.8
0.6
0.4
0.2
0
1
4
16
64
256
1024
4096
16384
65536
Number of parallel processes P
s = 5%
s = 2%
s = 1%
s = 0%

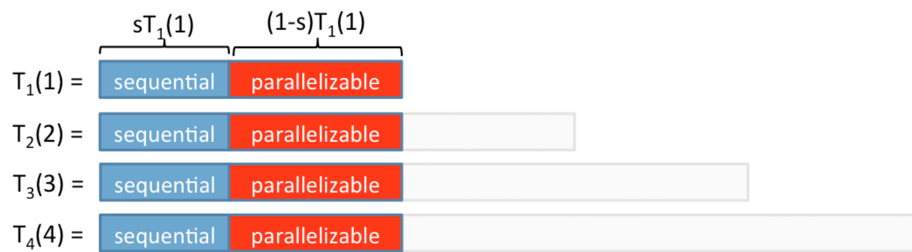
Weak scaling: Gustafson's law

- **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.
  - $T_1(N) = T_1(1)[\, s + (1-s)N \,]$ $(0 \le s \le 1)$



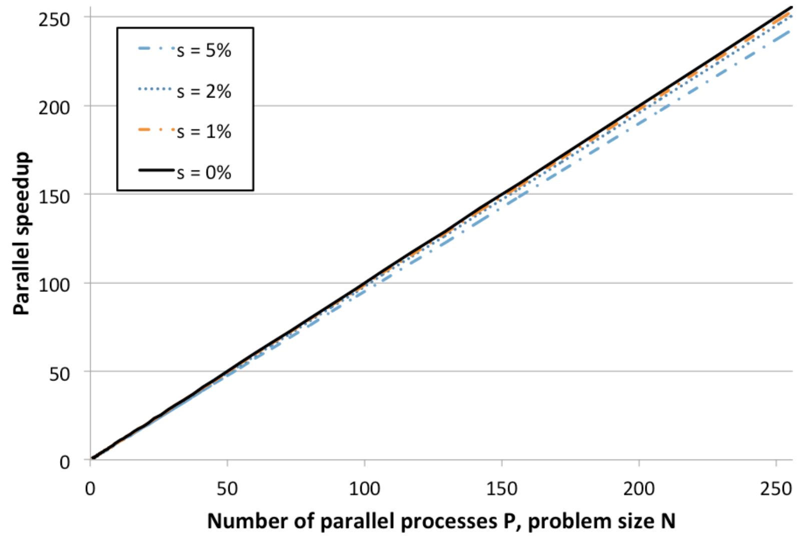
Gustafson's law

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



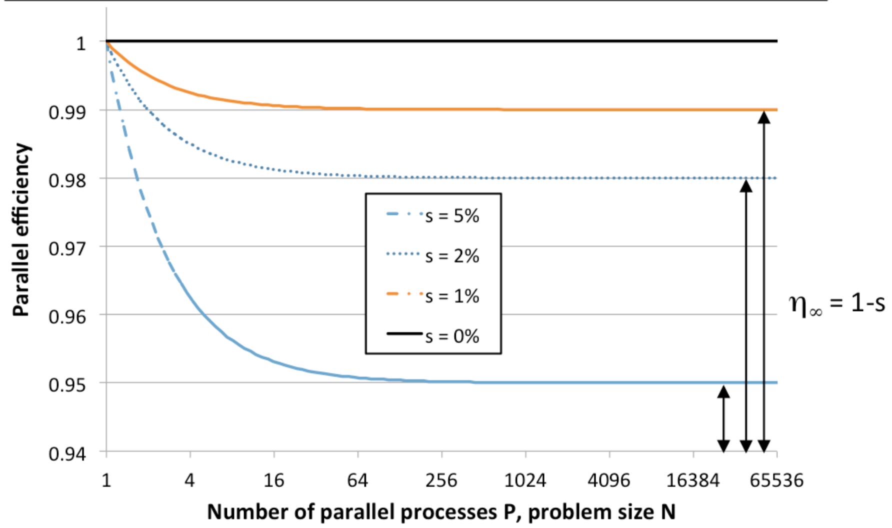
Therefore, $T_P(N) = T_1(1)[\, s + (1-s)N/P \,]$ and hence $T_P(P) = T_1(1)$

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


Gustafson's law
Parallel speedup
Number of parallel processes P, problem size N
s = 5%
s = 2%
s = 1%
s = 0%
0
50
100
150
200
250

Gustafson's law

Same graph as on previous slide, but expressed as parallel efficiency and log scale



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

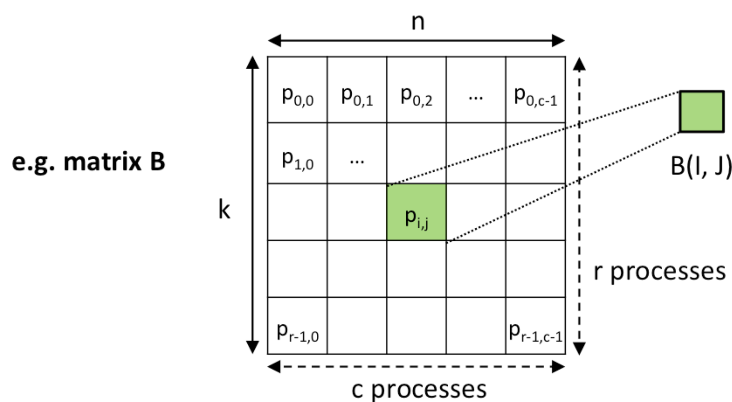
Case study reproduced from J. Demmel

Do not confuse the alpha and beta from the matrix-matrix multiplication formula with the alpha and beta from the communication cost model (latency and time per element).

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):

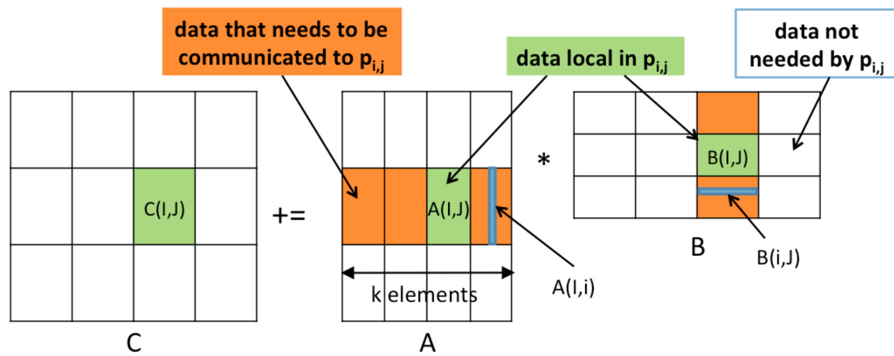
$$C(I,J) = C(I,J) + \sum_{i=0}^{k-1} A(I,i) * B(i,J)$$

do this in parallel

$\forall\, I = 0 \ldots r$

$\forall\, J = 0 \ldots c$

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



Note that when r is not equal to c (as in this case: r = 3; c = 4), the summation has to be performed over individual elements, rather than over blocks (blocks of A and B are incompatible in general).

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

We again ignore the contribution of the log factor in the isoefficiency discussion.

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

ScaLAPACK SOFTWARE HIERARCHY

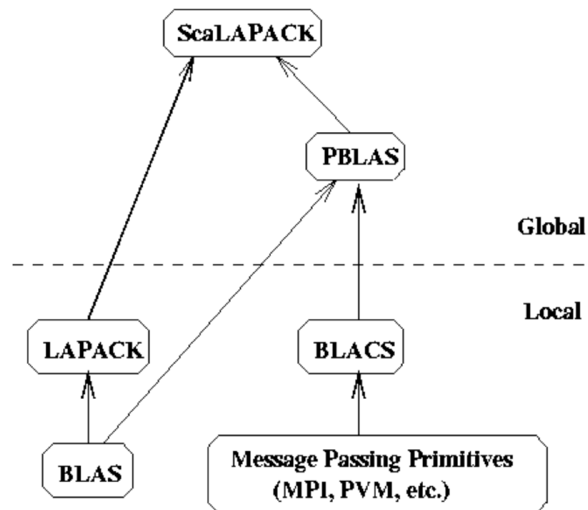


Image take from J. Demmel

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

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;
                }
        }
}
```

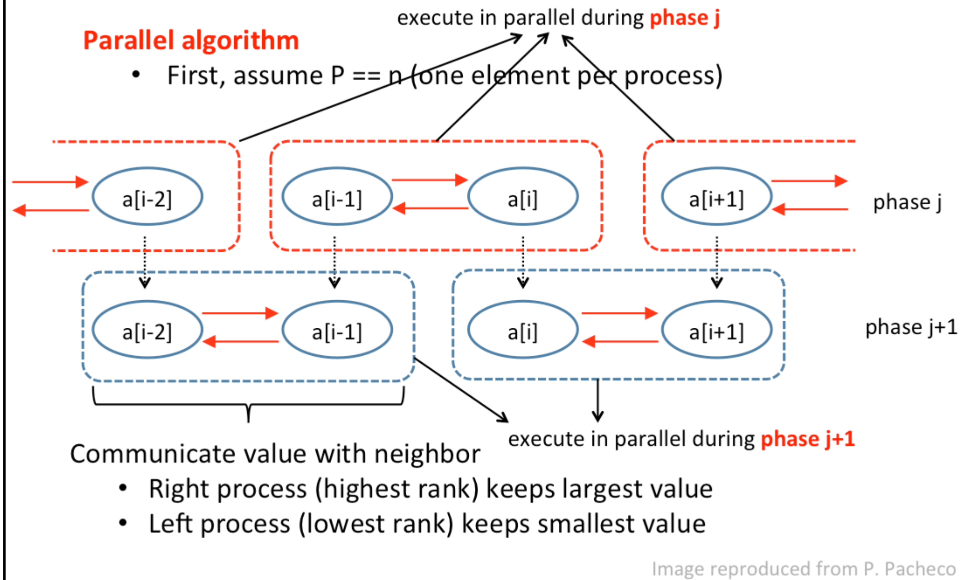

Algorithm reproduced from P. Pacheco

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])



After P == n phases, all elements are sorted

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

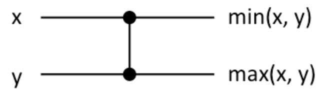
Parallel odd-even transposition sort algorithm **analysis**

- **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!)

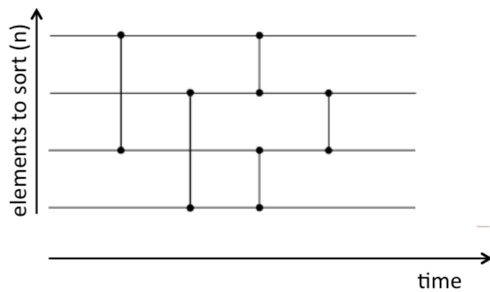
Algorithm reproduced from P. Pacheco

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)



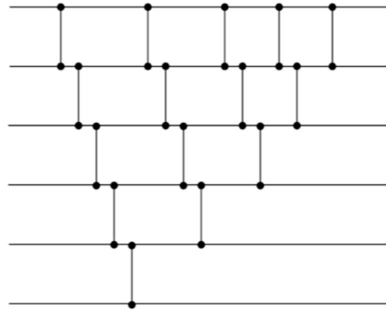
Example (4 elements to sort)



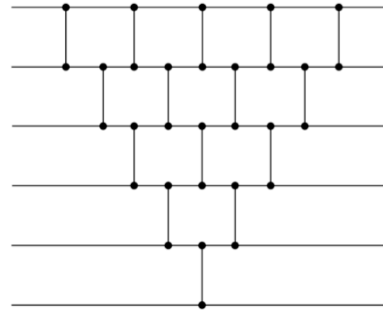
Note that sorting networks can only be used for sorting algorithms for which the order of comparisons is fixed and set in advance.

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

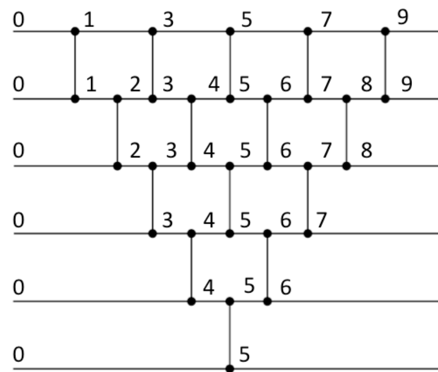
Parallel runtime
(assume P == n processes)
= "**depth** of sorting network"
= 2n – 3

For the parallel runtime, we assume that comparators can be indepently

Case study: parallel sorting algorithm

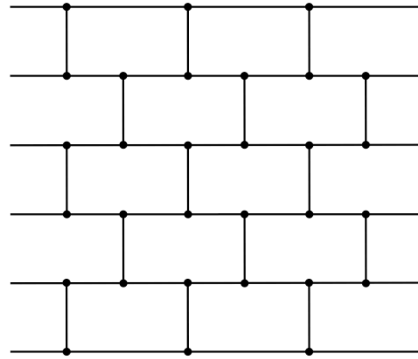
Definition: **depth of a sorting network** (= parallel runtime)

- 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)

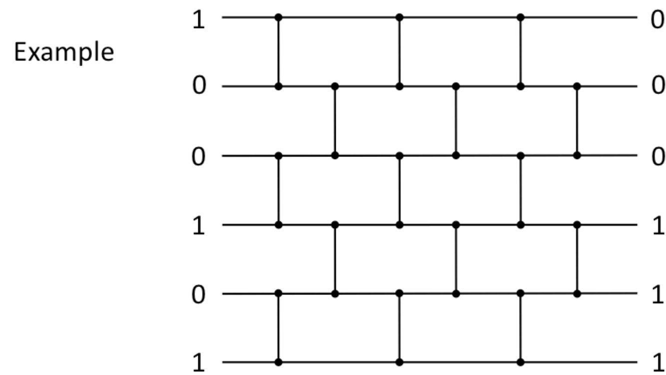
Case study: parallel sorting algorithm

- **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

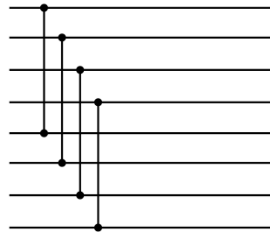
Case study: parallel sorting algorithm

- **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

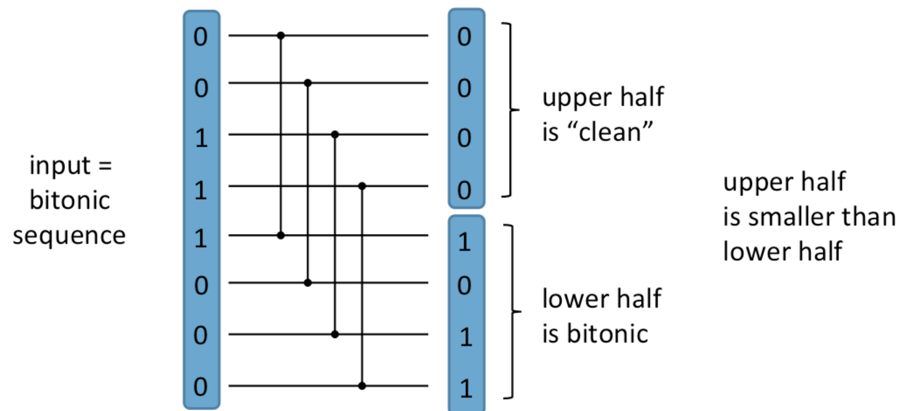
Half-cleaner
example (n=8)



- 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.

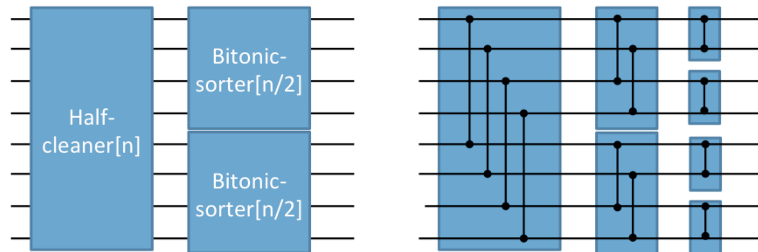
Case study: parallel sorting algorithm

- 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

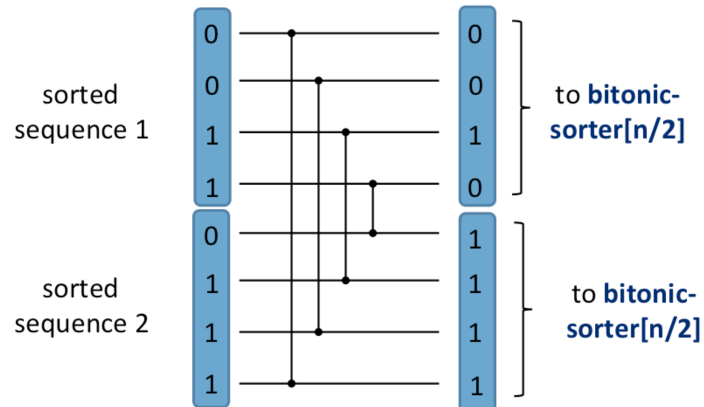


A bitonic sorter sorts a bitonic sequence of length $n = 2^k$ using

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

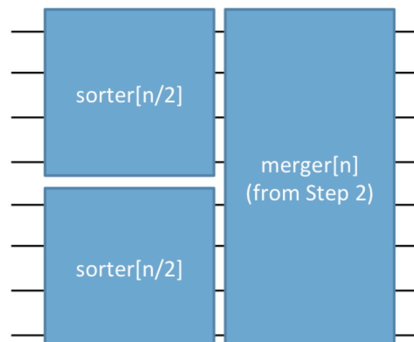
Case study: parallel sorting algorithm

- **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



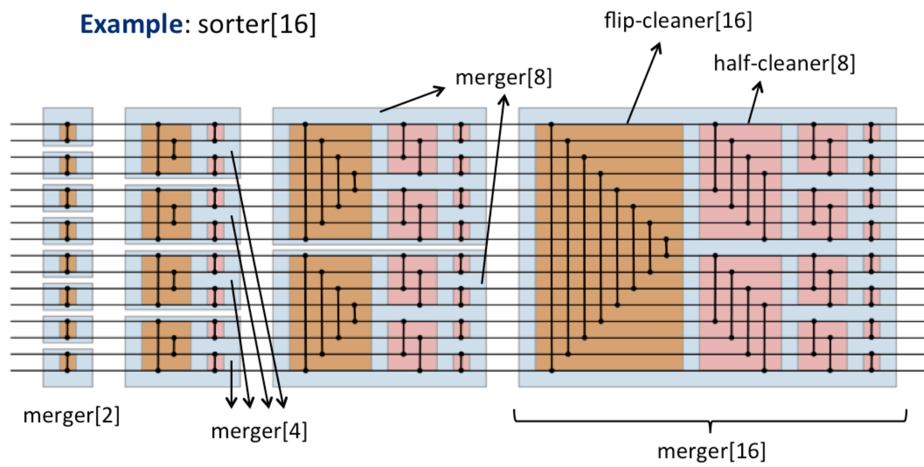
Case study: parallel sorting algorithm

- **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

Example: sorter[16]



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

The end