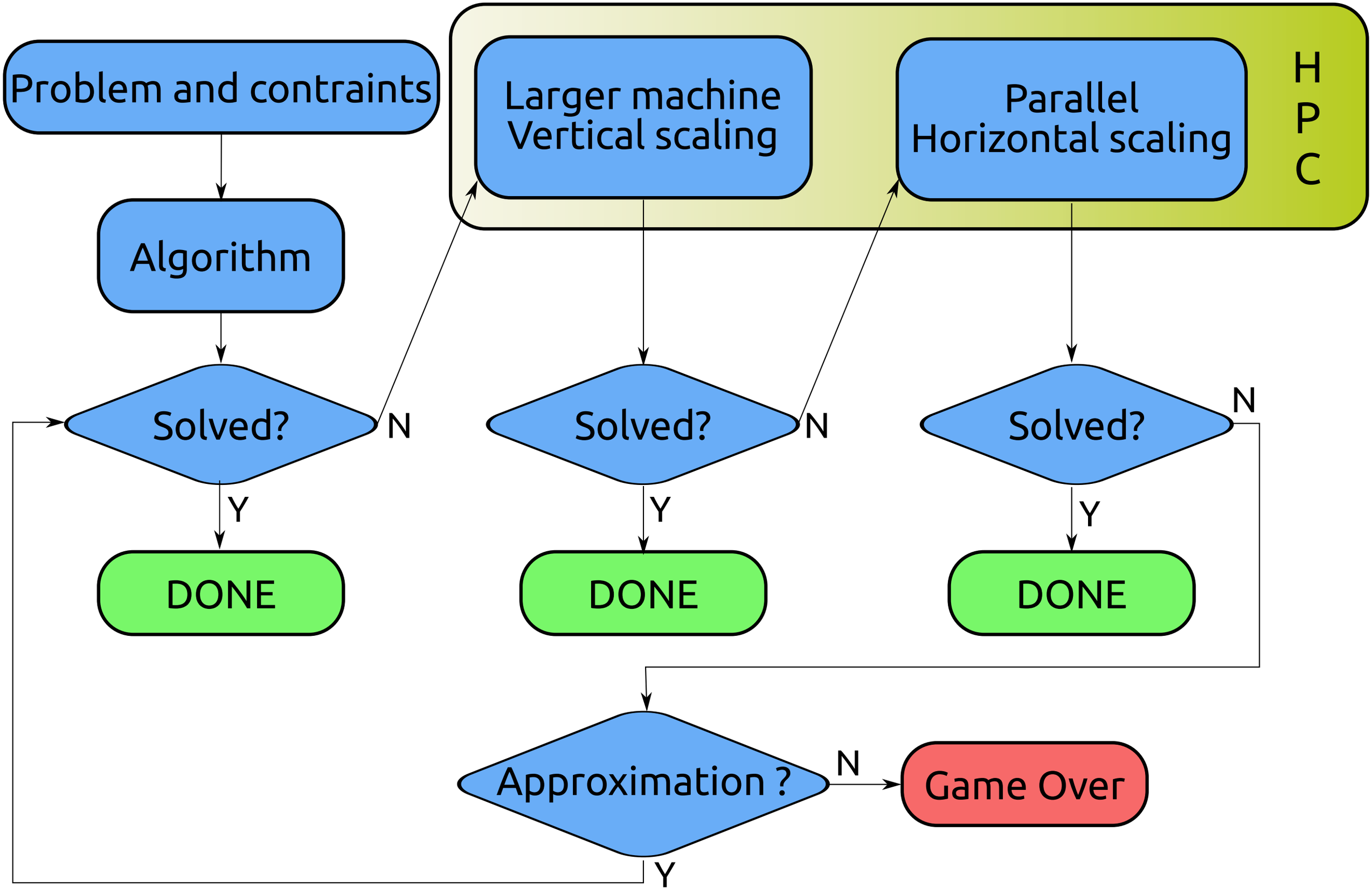
High Performance Computing

Source of majority of materials

- CSC (IT Center for science)
- Summer school on HPC (2014)
- Creative commons by-nc-sa

How To Solve any problem



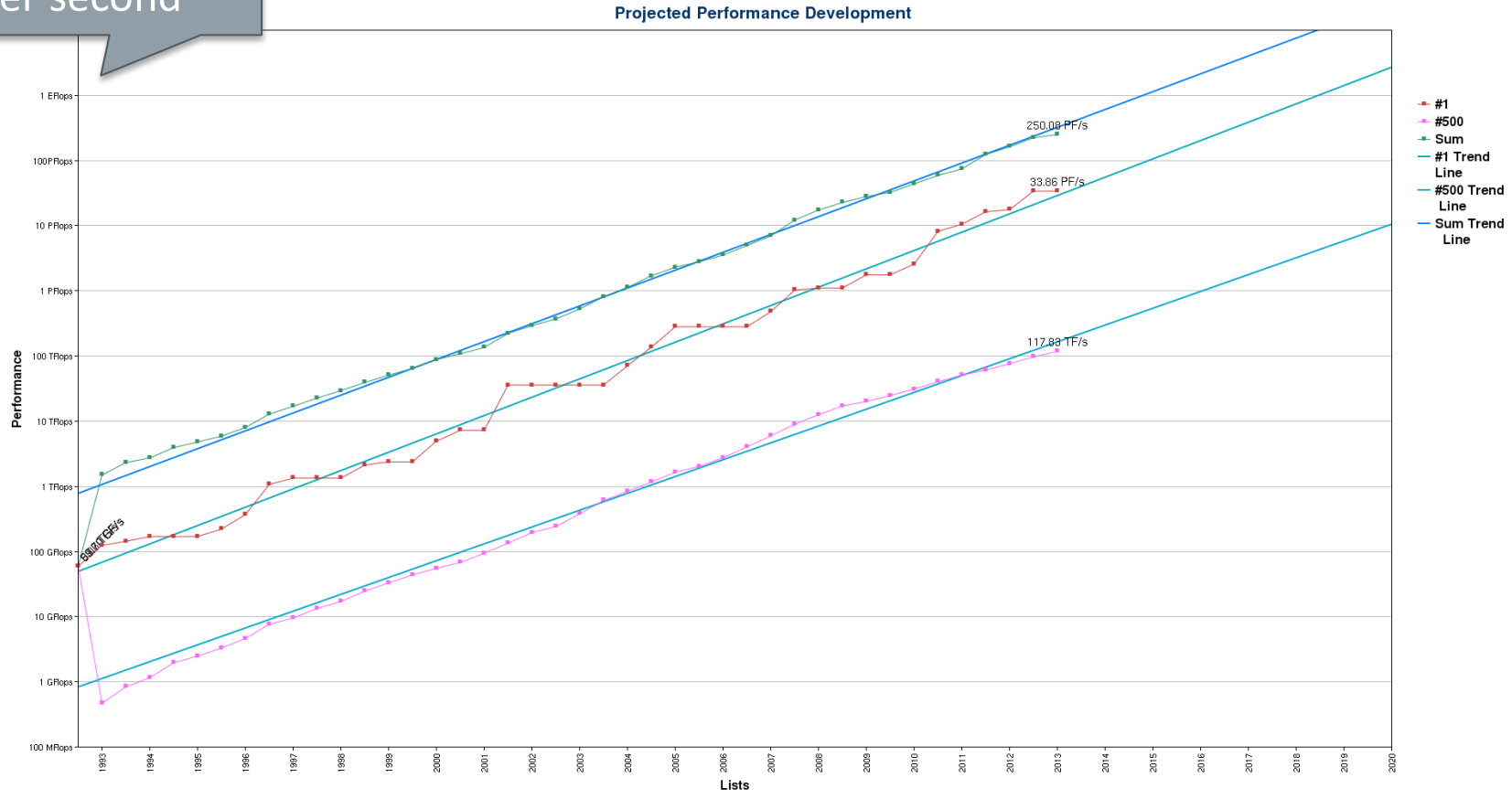
WHAT IS HIGH-PERFORMANCE COMPUTING?

High-performance computing

- A special branch of scientific computing – high-performance computing (HPC) or *supercomputing* - that refers to computing with supercomputer systems, is the scientific instrument of the future
- It offers a promise of breakthroughs in many major challenges that humankind faces today
- Useful through various disciplines

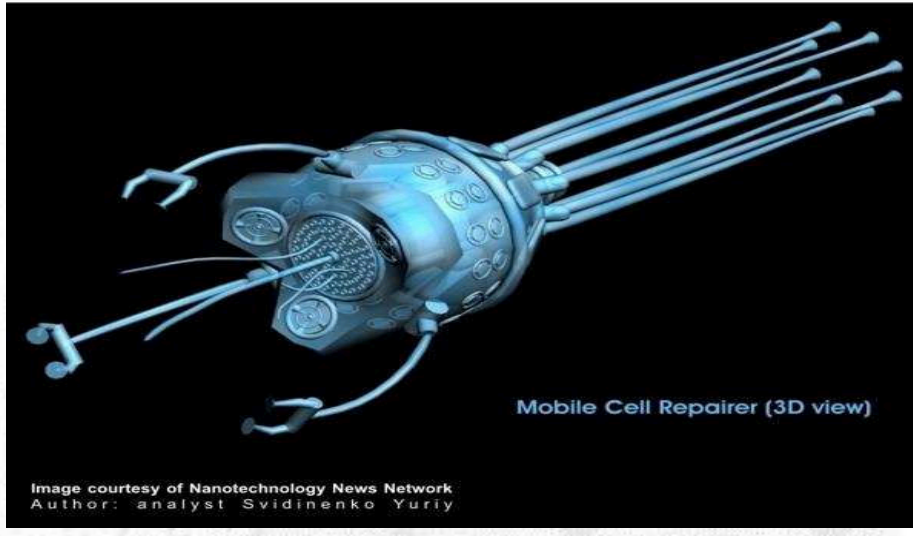
Flops: floating-point operations per second

HPC through the ages



Materials science

- New materials
  - Design of meta-materials
  - Hydrogen storage
- New methods for catalysis
  - Industrial processes
  - Air and water purification
- Design of devices from first principles

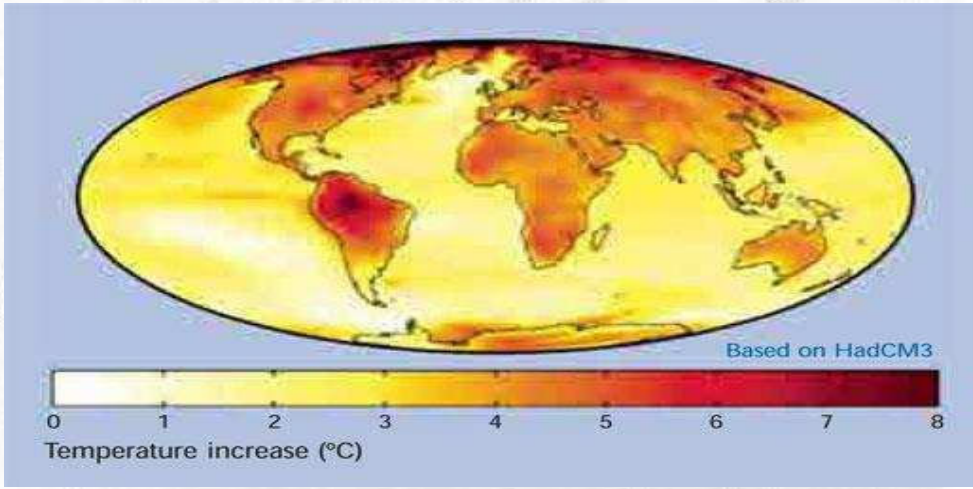


Life sciences

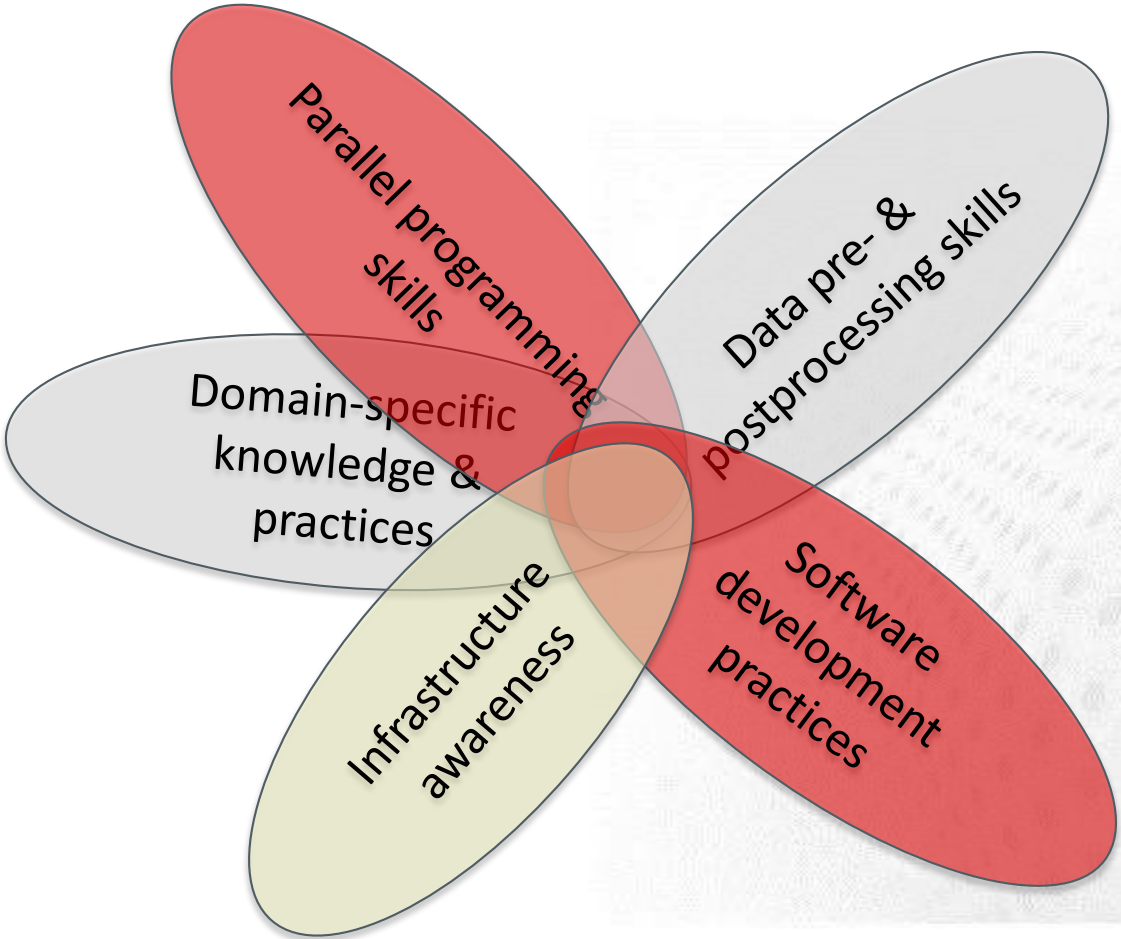
- Next-generation sequencing techniques
- Identifying genomic variants associated with common complex diseases
- Understanding the natural development of diseases
- Simulated surgeries
- Predicting protein folding

Earth sciences

- Long term climate modeling
  - Coupling atmospheric, ocean and land models
  - Understanding and predicting the climate change
- High-resolution weather prediction
  - Predicting extreme weather conditions
  - District-scale forecasts
- Whole-Earth seismological models



Utilizing HPC in scientific research



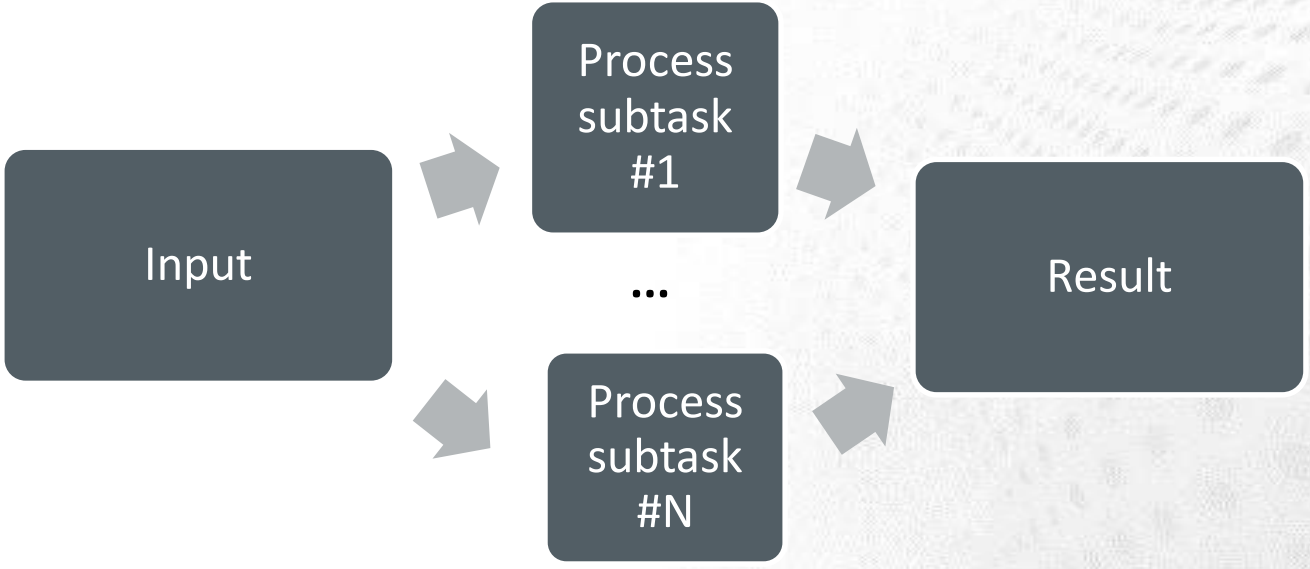
PARALLEL COMPUTING CONCEPTS

Computing in parallel

- Serial computing
  - single processing unit (“core”) is used for solving a problem

Computing in parallel

- Parallel computing
  - A problem is split into smaller subtasks
  - multiple subtasks are processed *simultaneously* using multiple cores

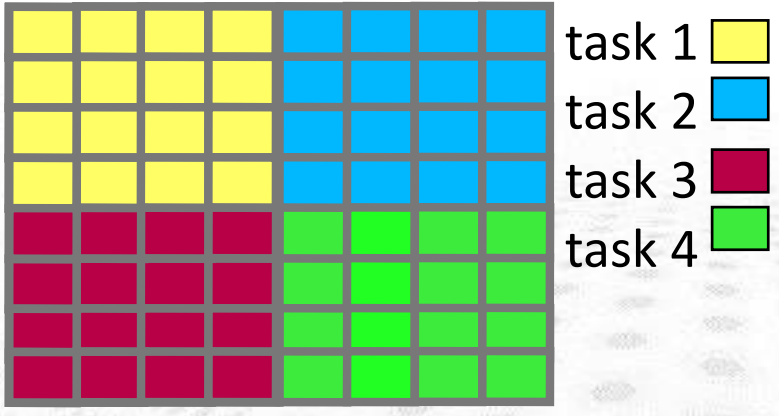


5 types of Parallel computing (personal opinion)

- Vector operations
- Multiple pipelines
- Hyper threading
- Multiple cores
- Multiple nodes

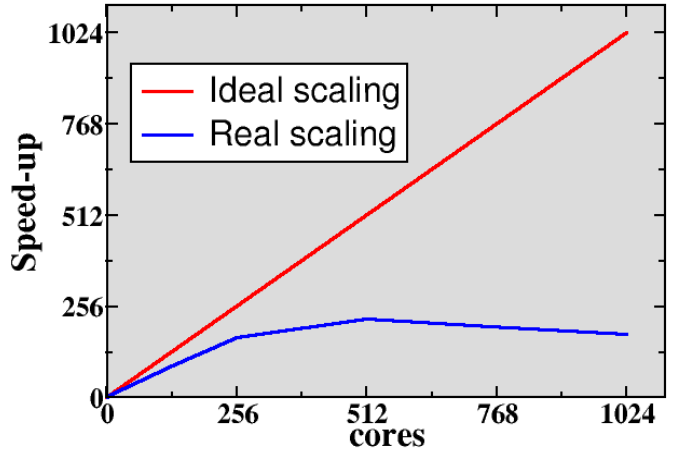
Exposing parallelism

- Data parallelism
  - Data is distributed to processor cores
  - Each core performs simultaneouously (nearly) identical operations with different data
- Task parallelism
  - Different cores perform different operations with (the same or) different data
- These can be combined



Parallel scaling

- Strong parallel scaling
  - constant problem size
  - execution time decreases in proportion to the increase in the number of cores
- Weak parallel scaling
  - increasing problem size
  - execution time remains constant when number of cores increases in proportion to the problem size

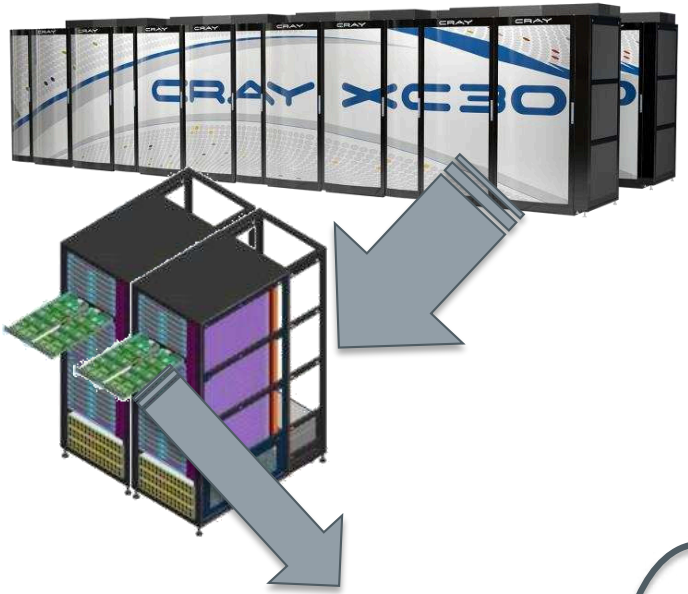


Parallel computing concepts

- Load balance
  - distribution of workload to different cores
- Parallel overhead
  - additional operations which are not present in serial calculation
  - synchronization, redundant computations, communications

ON SUPERCOMPUTER ARCHITECTURES

Supercomputer autopsy

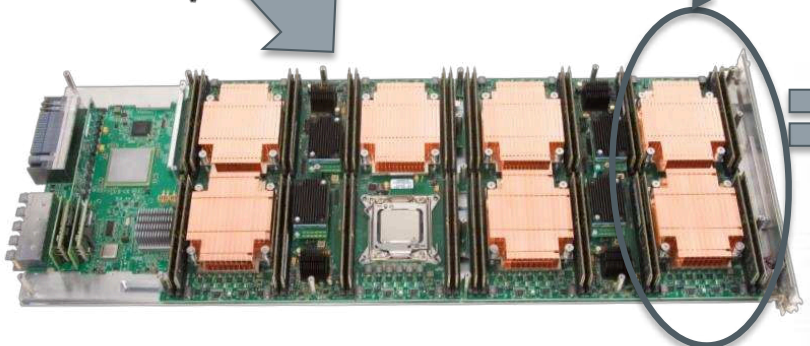


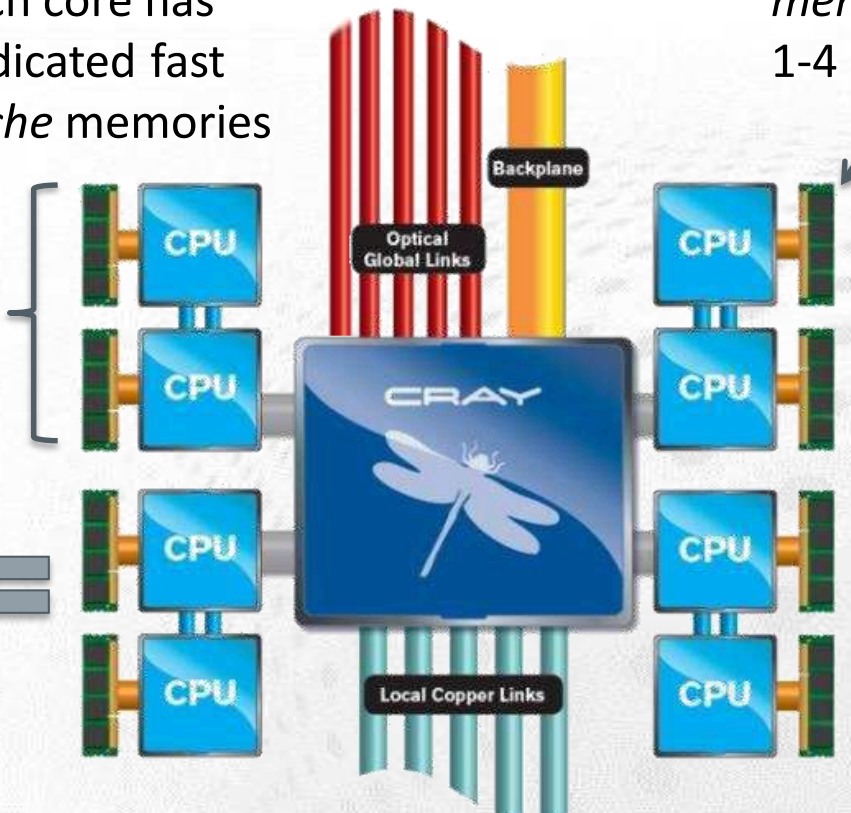
1 *CPU* = 8 *cores*
Each core has dedicated fast *cache* memories

Interconnect

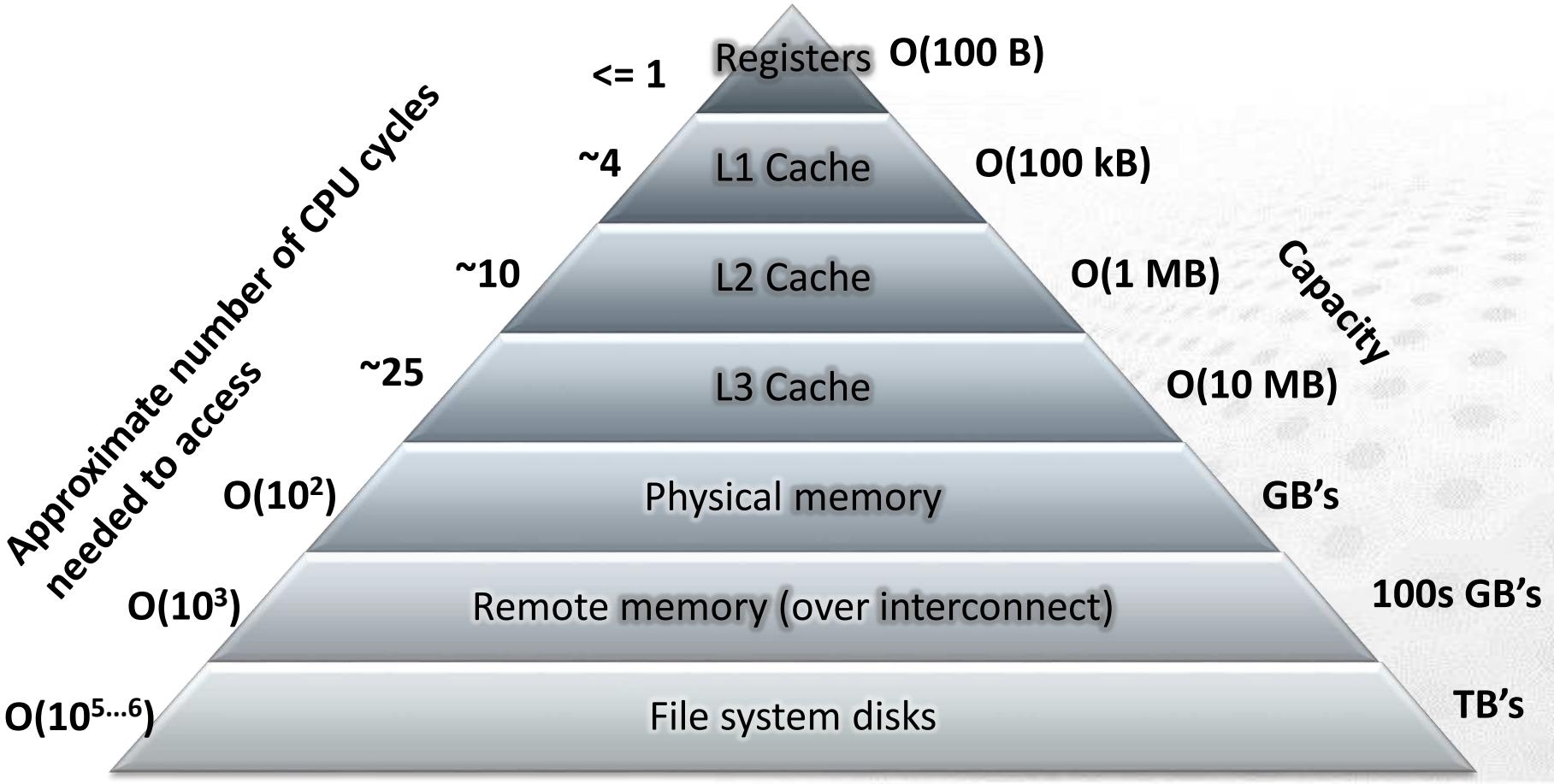
Local *memory*, 1-4 GB/core

1 *node*





Memory hierarchy



1. Productivity: Choosing a programming language

- Most common are C, C++ and Fortran 9X
  - mostly a question of taste
  - C++ more full featured with object oriented features and many more data structures (maps, etc.)
  - Fortran has really good array syntax
- One should also consider Python
  - much faster coding cycle (and less error prone)
  - parts of the code can be written in C, Fortran
  - tradeoff in speed; e.g. 10% overhead (with C extensions)

Accelerators

- Specialized parallel HW for floating point operations
  - General purpose graphics processing units (GPGPU) have been the most common accelerators during the last few years
  - New technology emerging: Intel Xeon Phi
- Co-processors for traditional CPUs
- Refactoring of programs required

Parallel programming models

- Message passing
  - Can be used both in distributed and shared memory computers
  - Programming model allows for good parallel scalability
  - Programming is quite explicit
- Threading (pthreads, OpenMP)
  - Can be used only in shared memory computers
  - Limited parallel scalability
  - “Simpler”/less explicit programming

Message-passing interface

- MPI is an application programming interface (API) for communication between separate processes
  - The most widely used approach for *distributed* parallel computing
- MPI programs are portable and scalable
- MPI is flexible and comprehensive
  - Large (over 120 procedures)
  - Concise (often only 6 procedures are needed)
- MPI standardization by MPI Forum

Execution model

- Parallel program is launched as set of independent, identical processes
- The same program code and instructions
- Can reside in different nodes
  - or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, srun, aprun, poe, ...

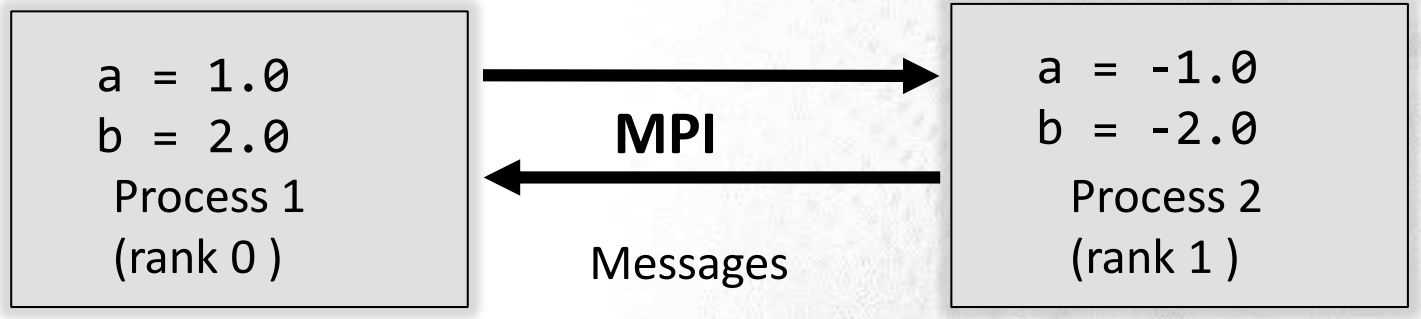
MPI ranks

- MPI runtime assigns each process a rank
  - identification of the processes
  - ranks start from 0 and extent to N-1
- Processes can perform different tasks and handle different data basing on their rank

```
...
if ( rank == 0 ) {
   ...
   }
if ( rank == 1) {
   ...
   }
...
```

Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



MPI communicator

- Communicator is an object connecting a group of processes
- Initially, there is always a communicator MPI_COMM_WORLD which contains all the processes
- Most MPI functions require communicator as an argument
- Users can define own communicators

Routines of the MPI library

- Information about the communicator
  - number of processes
  - rank of the process
- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Advanced features

Programming MPI

- MPI standard defines interfaces to C and Fortran programming languages
  - There are unofficial bindings to Python, Perl and Java
- C call convention

```
rc = MPI_Xxxx(parameter,...)
```

  - some arguments have to passed as pointers
- Fortran call convention

```
CALL MPI_XXXX(parameter,..., rc)
```

  - return code in the last argument

First five MPI commands

- Set up the MPI environment

  ```
  MPI_Init()
  ```

- Information about the communicator

  ```
  MPI_Comm_size(comm, size)
  MPI_Comm_rank(comm, rank)
  ```

  – Parameters

  comm communicator

  size number of processes in the communicator

  rank rank of this process

First five MPI commands

- Synchronize processes

  `MPI_Barrier(comm)`

- Finalize MPI environment

  `MPI_Finalize()`

Writing an MPI program

- Include MPI header files
  - C: `#include <mpi.h>`
  - Fortran: `INCLUDE 'mpif.h'`
- Call MPI_Init
- Write the actual program
- Call MPI_Finalize before exiting from the main program

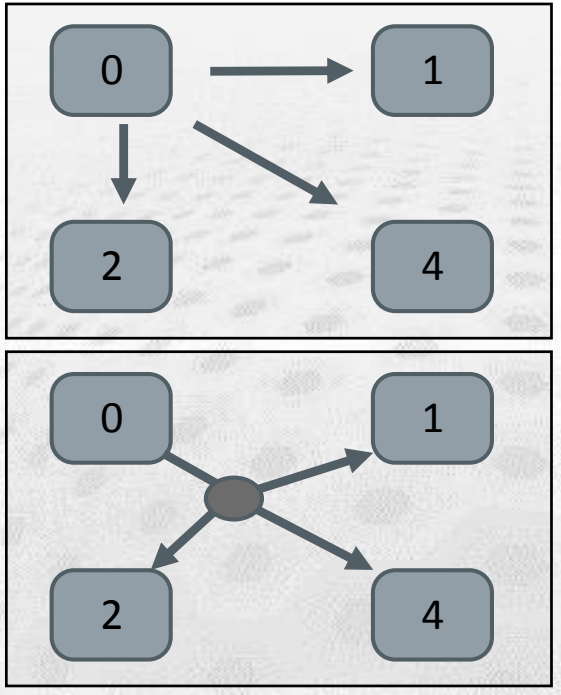
Summary

- In MPI, a set of *independent processes* is launched
  - Processes are identified by *ranks*
  - Data is always *local* to the process
- Processes can exchange data by sending and receiving *messages*
- MPI library contains functions for
  - Communication and synchronization between processes
  - Communicator manipulation

POINT-TO-POINT COMMUNICATION

Introduction

- MPI processes are *independent*, they *communicate* to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time



MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match – one receive per send

MPI point-to-point operations

- Each message (envelope) contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data.
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

Presenting syntax

Send operation

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

- buf — The data that is sent
- count — Number of elements in buffer
- datatype — Type of each element in buf (see later slides)
- dest — The rank of the receiver
- tag — An integer identifying the message
- comm — A communicator
- error — Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Operations presented in pseudocode, C and Fortran bindings presented in extra material slides.

INPUT arguments in red

OUTPUT arguments in blue

Note! Extra error parameter for Fortran

Slide with extra material included in handouts

Send operation

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

- The return value of the function is the error value

Fortran binding

```
MPI_SEND(buffer, count, datatype,
dest, tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror
```

- **ierror:** the error value

Send operation

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

buf	The data that is sent
count	Number of elements in buffer
datatype	Type of each element in buf (see later slides)
dest	The rank of the receiver
tag	An integer identifying the message
comm	A communicator
error	Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Receive operation

```
MPI_Recv(buf, count, datatype, source, tag, comm,
   status)
```

buf	Buffer for storing received data
count	Number of elements in buffer, not the number of element that are actually received
datatype	Type of each element in buf
source	Sender of the message
tag	Number identifying the message
comm	Communicator
status	Information on the received message
error	As for send operation

MPI datatypes

- MPI has a number of predefined datatypes to represent data
- Each C or Fortran datatype has a corresponding MPI datatype
  - C examples: MPI_INT for int and MPI_DOUBLE for double
  - Fortran example: MPI_INTEGER for integer
- One can also define custom datatypes

Special parameter values

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

dest	MPI_PROC_NULL	Null destination, no operation takes place
comm	MPI_COMM_WORLD	Includes all processes
error	MPI_SUCCESS	Operation successful

Special parameter values

```
MPI_Recv(buf, count, datatype, source, tag,
   comm, status)
```

source	MPI_PROC_NULL	No sender, no operation takes place
	MPI_ANY_SOURCE	Receive from any sender
tag	MPI_ANY_TAG	Receive messages with any tag
comm	MPI_COMM_WORLD	Includes all processes
status	MPI_STATUS_IGNORE	Do not store any status data
error	MPI_SUCCESS	Operation successful

Status parameter

- The status *parameter* in MPI_Recv contains information on how the receive succeeded
  - Number and datatype of received elements
  - Tag of the received message
  - Rank of the sender
- In C the status parameter is a struct, in Fortran it is an integer array

Status parameter

- Received elements

  Use the function

  ```
  MPI_Get_count(status, datatype, count)
  ```

- Tag of the received message

  C: **`status.MPI_TAG`**

  Fortran: **`status(MPI_TAG)`**

- Rank of the sender

  C: **`status.MPI_SOURCE`**

  Fortran: **`status(MPI_SOURCE)`**

Blocking routines & deadlocks

- Blocking routines
  - Completion depends on other processes
  - Risk for deadlocks – the program is stuck forever
- MPI_Send exits once the send buffer can be safely read and written to
- MPI_Recv exits once it has received the message in the receive buffer

Point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data

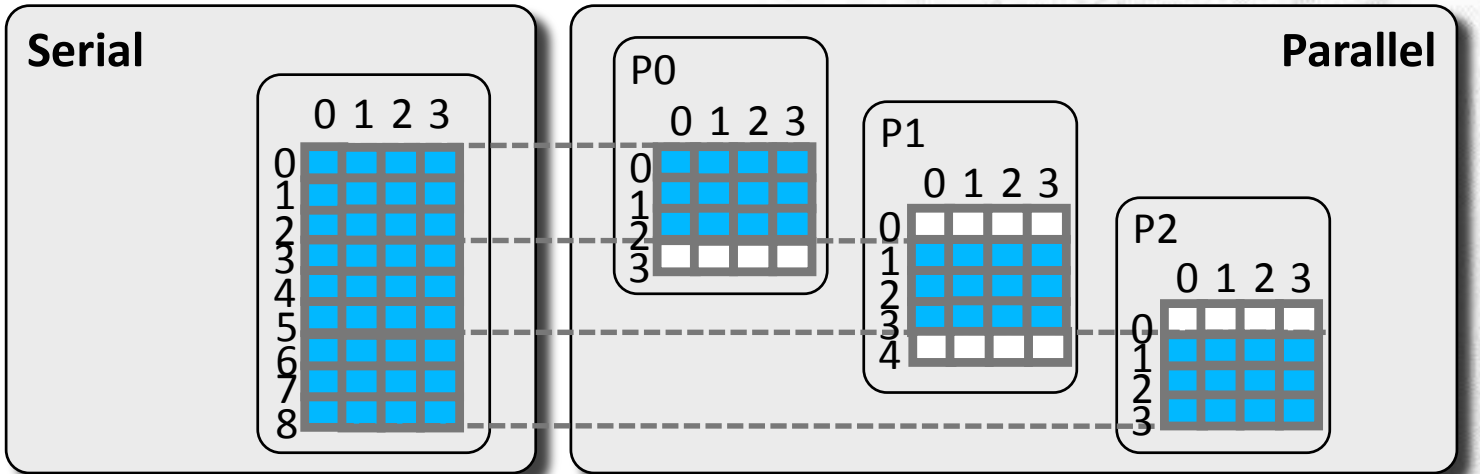
Combined send & receive

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
   recvbuf, recvcount, recvtype, source, recvtag, comm,
   status)
```

  - Parameters as for MPI_Send and MPI_Recv combined
- Sends one message and receives another one, with one single command
  - Reduces risk for deadlocks
- Destination rank and source rank can be same or different

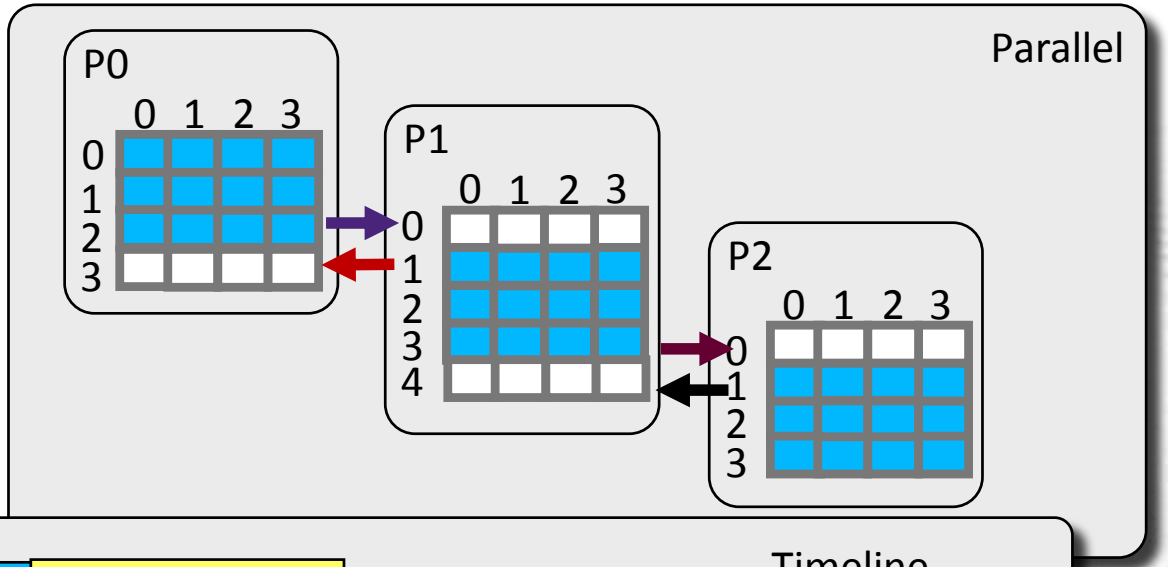
Case study 2: Domain decomposition

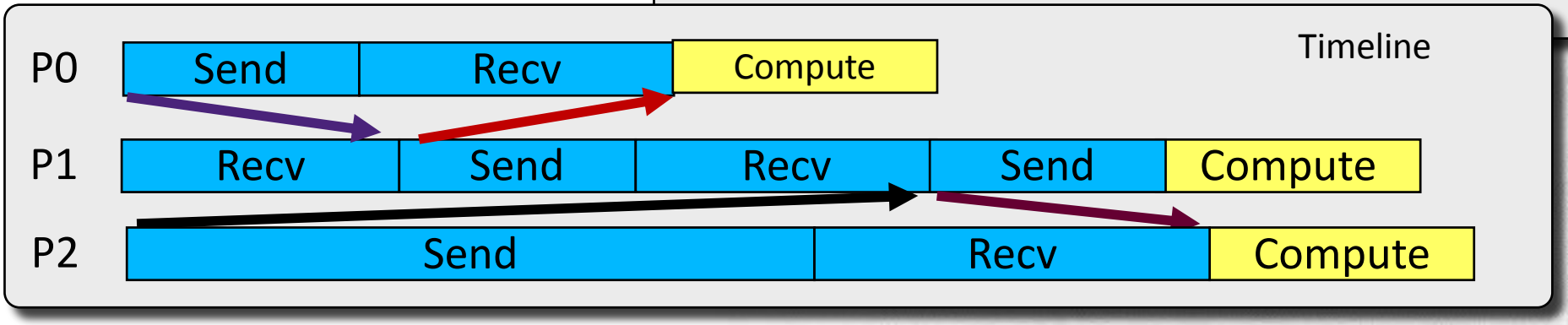
- Computation inside each domain can be carried out independently; hence in parallel
- *Ghost layer* at boundary represent the value of the elements of the other process



CS2: One iteration step

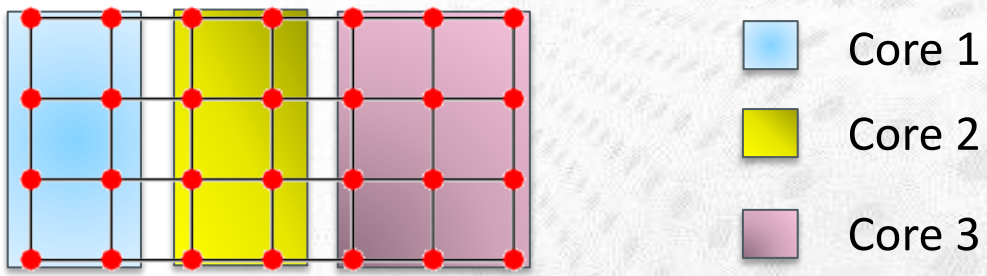
- Have to carefully schedule the order of sends and receives in order to avoid deadlocks





Solving heat equation in parallel

- Temperature at each grid point can be updated independently
- Domain decomposition



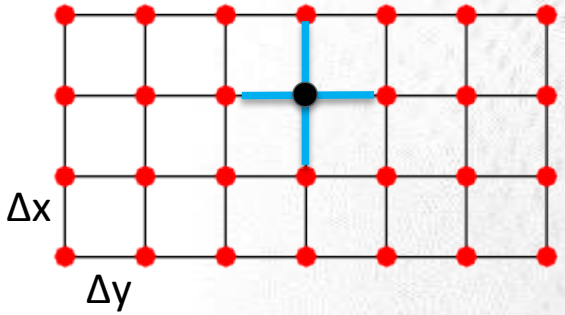
- Straightforward in shared memory computer

Numerical solution

- Finite difference Laplacian in two dimensions

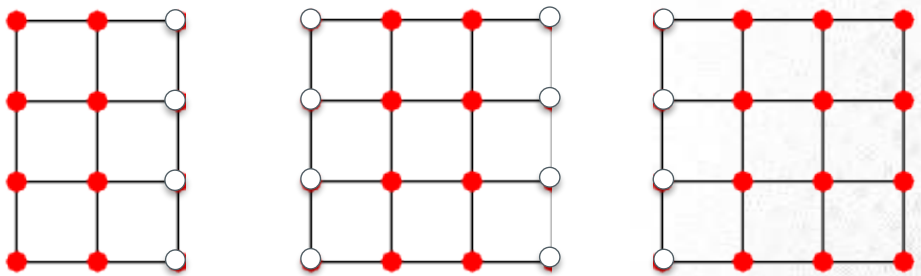
$$\nabla^2 u(i,j) = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2}$$

Temperature field u(i,j)



Solving heat equation in parallel

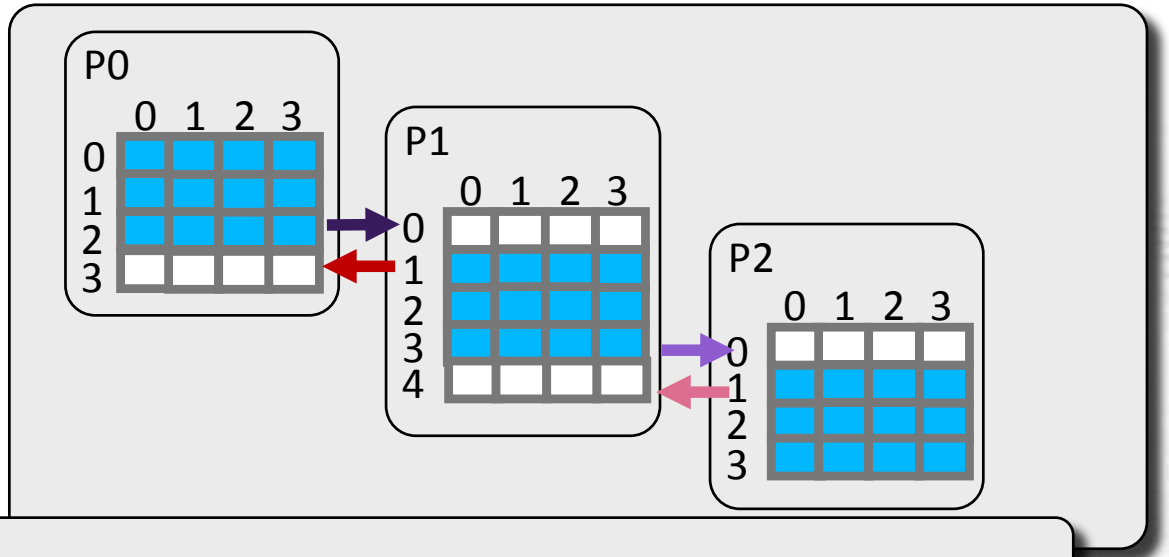
- In distributed memory computers, each core can access only its own memory
- Information about neighbouring domains is stored in ”ghost layers”

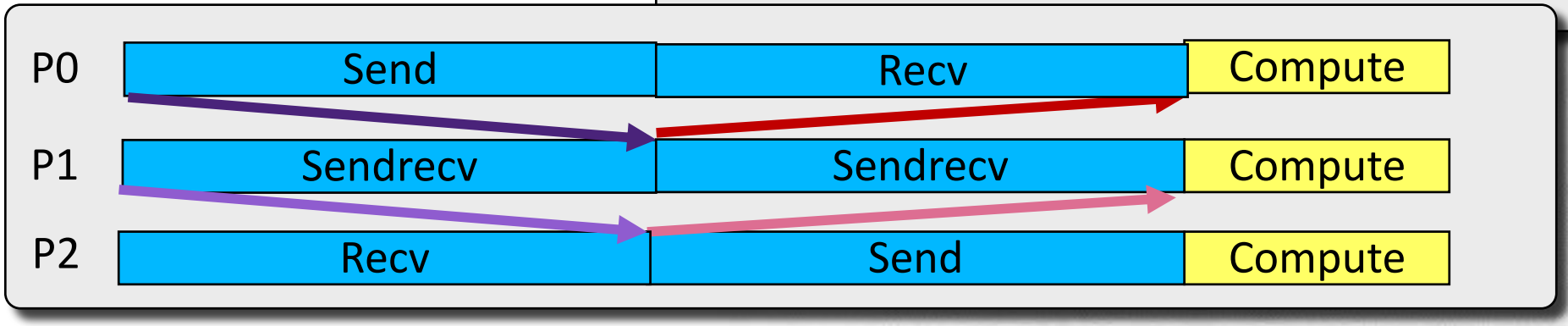


- Before each update cycle, CPU cores communicate boundary data: halo exchange

CS2: MPI_Sendrecv

- MPI_Sendrecv
  - Sends and receives with one command
  - No risk of deadlocks





Summary

- Point-to-point communication
  - Messages are sent between two processes
- We discussed send and receive operations enabling any parallel application
  - `MPI_Send` & `MPI_Recv`
  - `MPI_Sendrecv`
- Status parameter
- Special argument values

Web resources

- List of MPI functions with detailed descriptions

  http://mpi.deino.net/mpi_functions/index.htm

- Good online MPI tutorial:

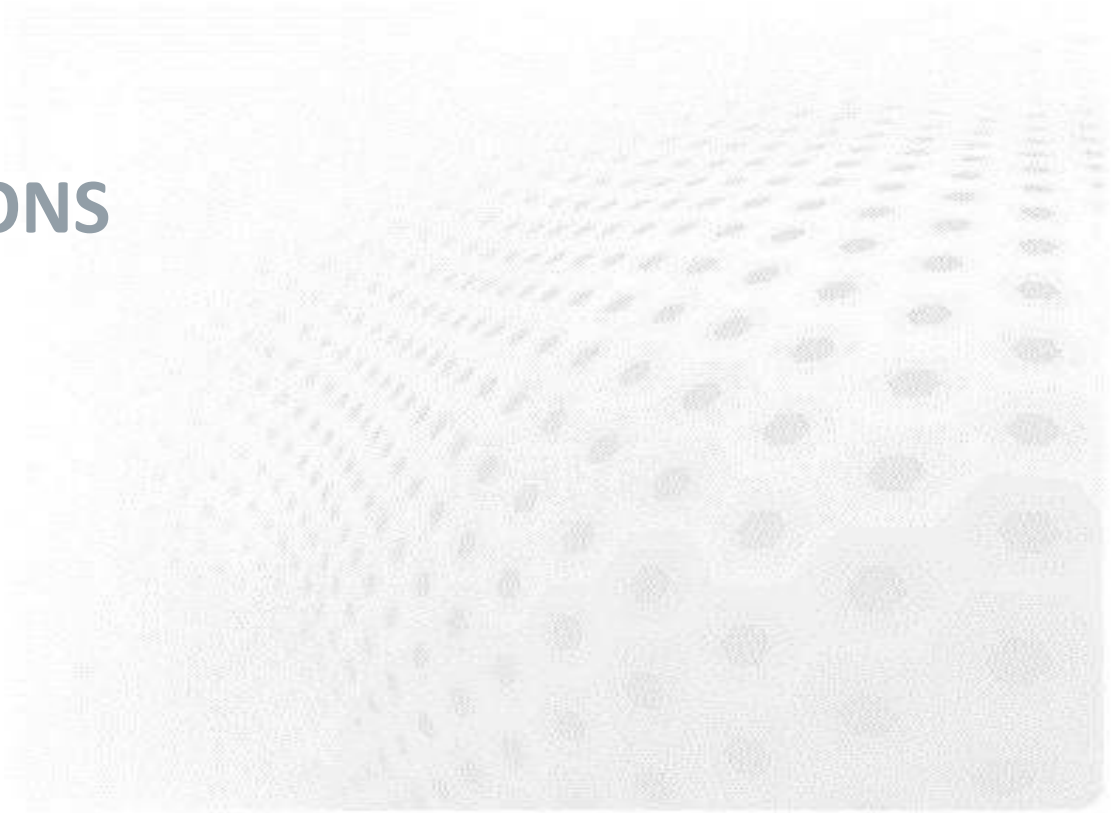
  https://computing.llnl.gov/tutorials/mpi

- MPI 3.0 standard

  http://www.mpi-forum.org/docs/

- MPI Implementations
  - MPICH2 http://www.mcs.anl.gov/research/projects/mpich2/
  - OpenMPI http://www.open-mpi.org/

COLLECTIVE OPERATIONS



Outline

- Introduction to collective communication
- One-to-many collective operations
- Many-to-one collective operations
- Many-to-many collective operations
- Non-blocking collective operations
- User-defined communicators

Introduction

- Collective communication transmits data among all processes in a process group
  - These routines must be called by all the processes in the group
- Collective communication includes
  - data movement
  - collective computation
  - synchronization

Example

MPI_Barrier

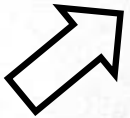
makes each task hold until all tasks have called it

```
int MPI_Barrier(comm)
MPI_BARRIER(comm, rc)
```

Introduction

- Collective communication outperforms normally point-to-point communication
- Code becomes more compact and easier to read:

```
if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_send(a, 1048576, &
        MPI_REAL, i, tag, &
        MPI_COMM_WORLD, rc)
  end do
else
  call mpi_recv(a, 1048576, &
      MPI_REAL, 0, tag, &
      MPI_COMM_WORLD, status, rc)
end if
```



```
call mpi_bcast(a, 1048576, &
    MPI_REAL, 0, &
    MPI_COMM_WORLD, rc)
```

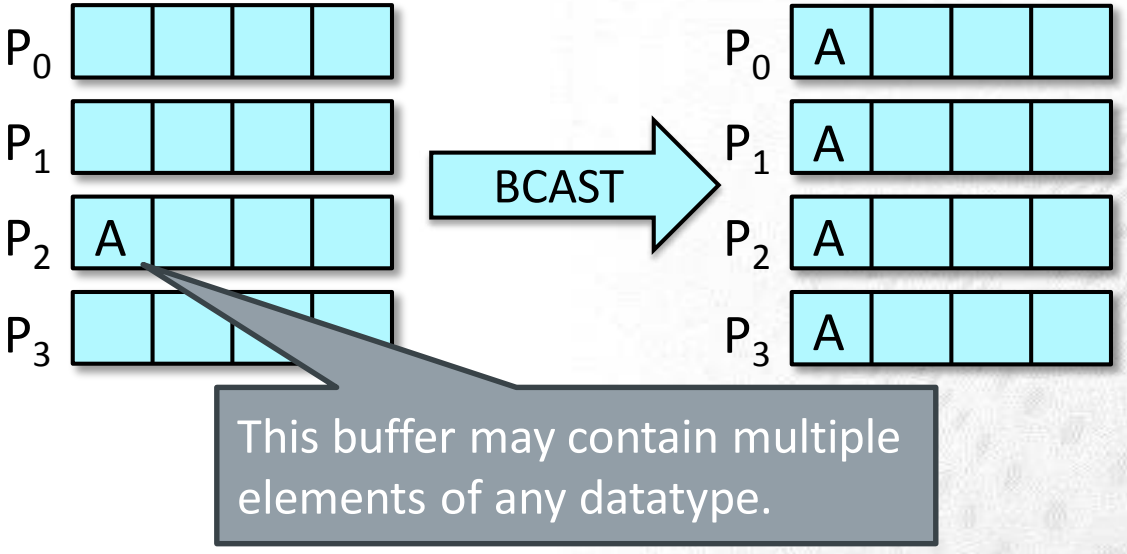
Communicating a vector a consisting of 1M float elements from the task 0 to all other tasks

Introduction

- Amount of sent and received data must match
- Non-blocking routines are available in the MPI 3 standard
  - Older libraries do not support this feature
- No tag arguments
  - Order of execution must coincide across processes

Broadcasting

- Send the same data from one process to all the other



Broadcasting

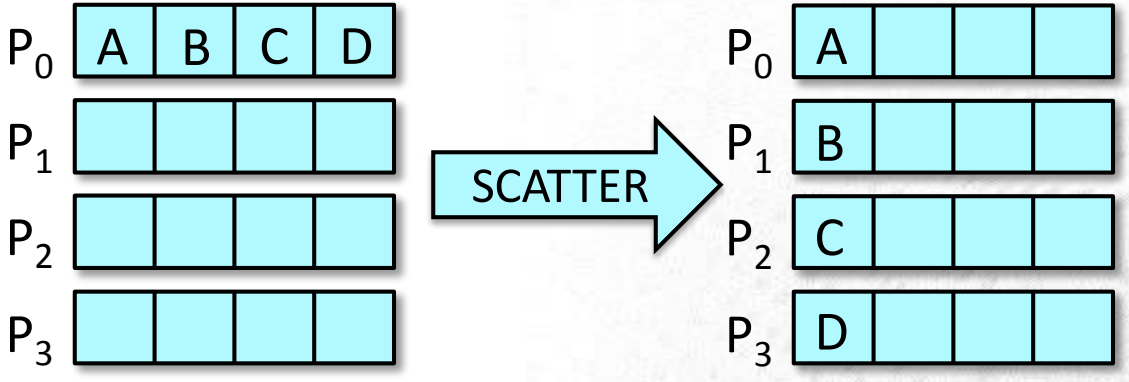
- With MPI_Bcast, the task *root* sends a *buffer* of data to all other tasks

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

buffer	data to be distributed
count	number of entries in buffer
datatype	data type of buffer
root	rank of broadcast root
comm	communicator

Scattering

- Send equal amount of data from one process to others



- Segments A, B, … may contain multiple elements

Scattering

- MPI_Scatter: Task *root* sends an equal share of data (*sendbuf*) to all other processes

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

sendbuf	send buffer (data to be scattered)
sendcount	number of elements sent to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements in receive buffer
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

Common mistakes with collectives

- ✘ Using a collective operation within one branch of an if-test of the rank

```
IF (my_id == 0) CALL MPI_BCAST(...
```

  - All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), *must* call the collective routine!
- ✘ Assuming that all processes making a collective call would complete at the same time
- ✘ Using the input buffer as the output buffer

```
CALL MPI_ALLREDUCE(a, a, n, MPI_REAL, MPI_SUM, ...
```

Summary

- Collective communications involve all the processes within a communicator
  - All processes must call them
- Collective operations make code more transparent and compact
- Collective routines allow optimizations by MPI library
- Performance consideration:
  - Alltoall is expensive operation, avoid it when possible

USER-DEFINED COMMUNICATORS

Communicators

- The communicator determines the "communication universe"
  - The source and destination of a message is identified by process rank within the communicator
- So far: MPI_COMM_WORLD
- Processes can be divided into subcommunicators
  - Task level parallelism with process groups performing separate tasks
  - Parallel I/O

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - In each of them it has a unique ID, however
  - Communication is normally within the communicator

Grouping processes in communicators