Parallel programming with MPI

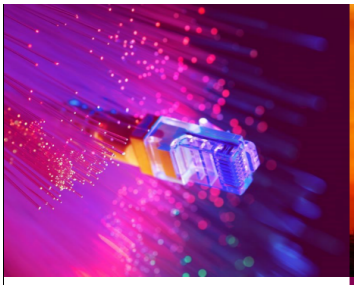
September 2 – 3, 2021

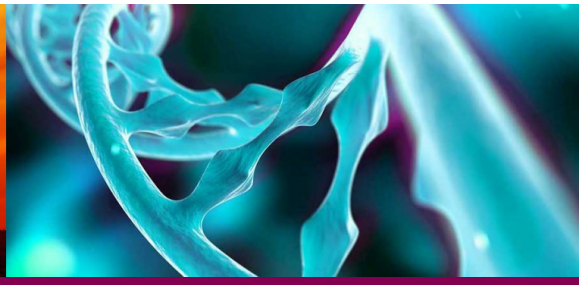
CSC – IT Center for Science Ltd., Espoo

Jussi Enkovaara
Cristian Achim

CSC
Training for Brilliant Minds

PRACE





Overview of High Performance Computing

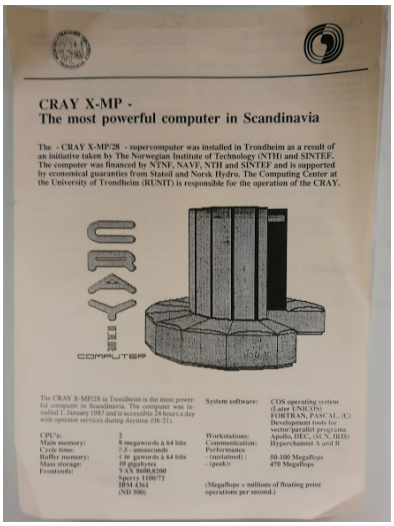
CSC Training, 2021


CSC – Finnish expertise in ICT for research, education and public administration

What is high-performance computing?

- Utilising computer power that is much larger than available in typical desktop computer
- Performance of HPC system (i.e. supercomputer) is often measured in floating point operations per second (flop/s)
  - For software, other measures can be more meaningful
- Currently, the most powerful system reaches ~500 x 10^{15} flop/s (500 Pflop / s)

What is high-performance computing?



CRAY X-MP -
The most powerful computer in Scandinavia

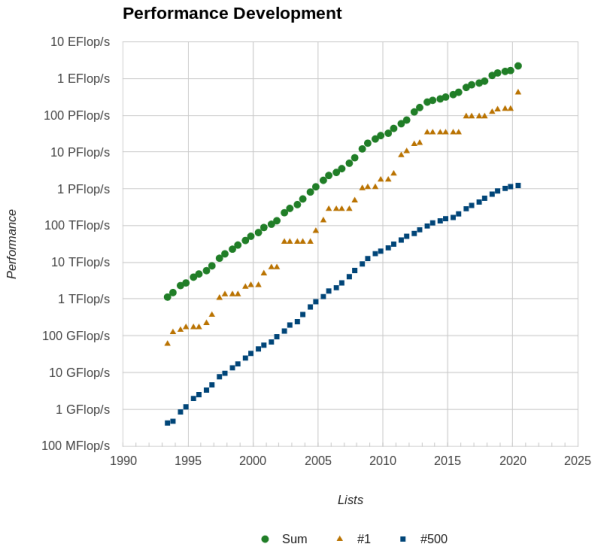
The - CRAY X-MP/28 - supercomputer was installed in Trondheim as a result of an initiative taken by The Norwegian Institute of Technology (NTH) and SINTEF. The computer was financed by NTNF, NAVF, NTH and SINTEF and is supported by economical guaranties from Statoil and Norsk Hydro. The Computing Center at the University of Trondheim (RUNIT) is responsible for the operation of the CRAY.

CRAY
COMPUTER

The CRAY X-MP/28 in Trondheim is the most powerful computer in Scandinavia. The computer was installed 1. January 1987 and is accessible 24 hours a day with operator services during daytime (08-21).

CPU's:	2	System software:	COS operating system (Later UNICOS). FORTRAN, PASCAL, (C) Development tools for vector/parallel programs
Main memory:	8 megawords à 64 bits	Communication:	Ethernet, HSC (SUN, IRIS)
Cycle time:	2,5 - nanoseconds	Performance	Hyperchannel A and B
Buffer memory:	4 @ gawords à 64 bits	- (sustained) :	50-100 Megaflops
Main storage:	10 gigabyte	- (peak) :	470 Megaflops
Frontends:	VAX 8600,8200 Sperry 1100/72 IBM 4361 (ND 500)	(Megaflops = millions of floating point operations per second.)	

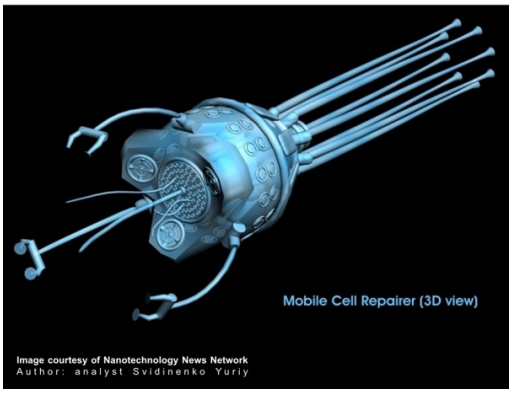
Top 500 list



What are supercomputers used for?

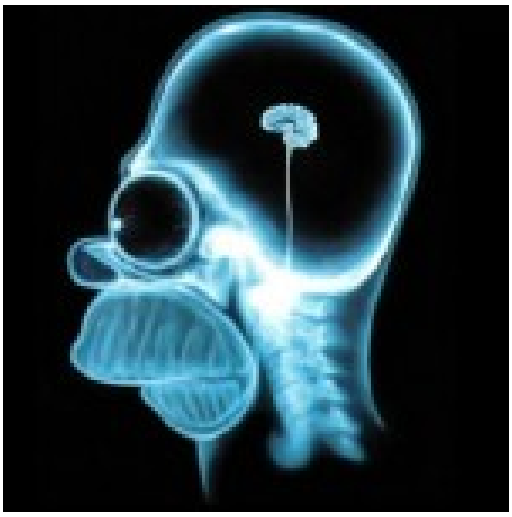
Materials science

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



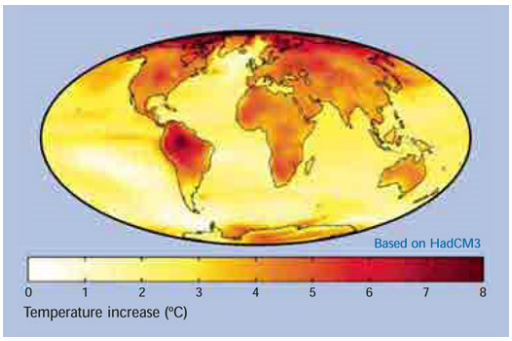
Life sciences

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



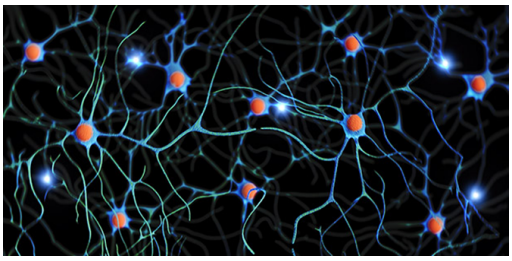
Earth sciences

- 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
  - Ensembles
- Whole-Earth seismological models



Artificial intelligence

- Machine learning
  - deep neural networks
- Large scale data analysis
- Interpreting experimental data
- Prediction of material properties

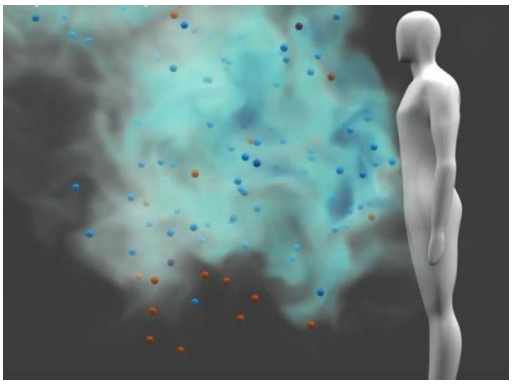


Puhti vs. the Coronavirus. COVID-19 fast track

- 2/3 of the Puhti's cores allocated
- 15 research project
- 3 areas:
  - airborne transmission
  - looking medication
  - identifying new variants

Airborne transmission of coronavirus

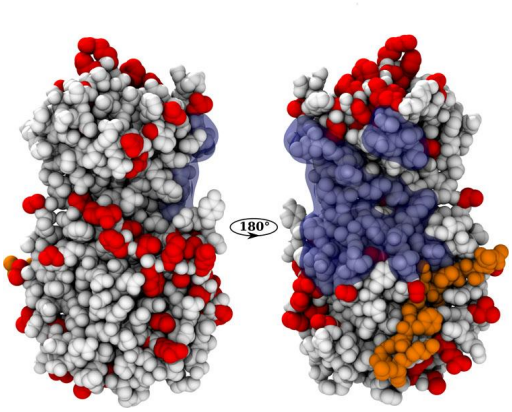
- consortium of fluid dynamics physicists, virologists and biomedical engineering specialists
- to model how the extremely small droplets are transported in air currents
- to asses the implications for coronavirus infections



- to be able to make scientific based recommendations for policy makers

The functional mechanism of the main protease

- atomistic molecular dynamics simulations and machine learning techniques
- to unveil the mechanism of action of the main protease of the SARS-COV-2 virus.
- figure out how the two proteins dimerize and how their binding could be blocked.

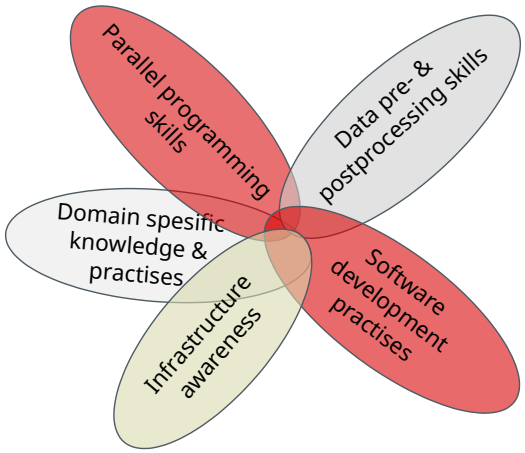


- investigate how drug candidates interfere with this this binding or blocks the enzyme activation

Monitoring of the virus variants

- identifying virus variants requires significant computational power
- increase the sequencing capacity
- Puhti supercomputer and the in-built virtual analysis workflow

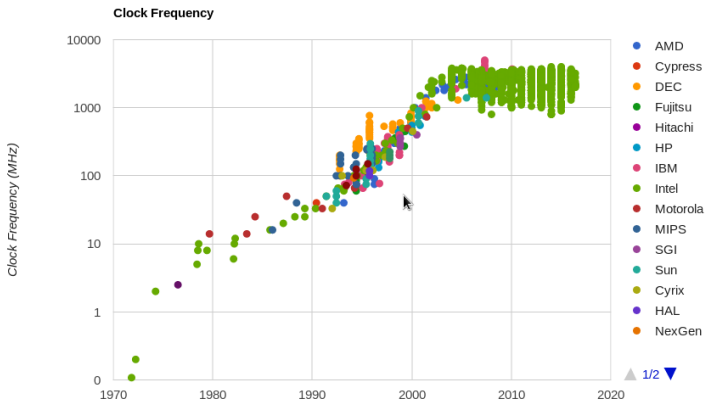
Utilizing HPC in scientific research



What are supercomputers made of?

CPU frequency development

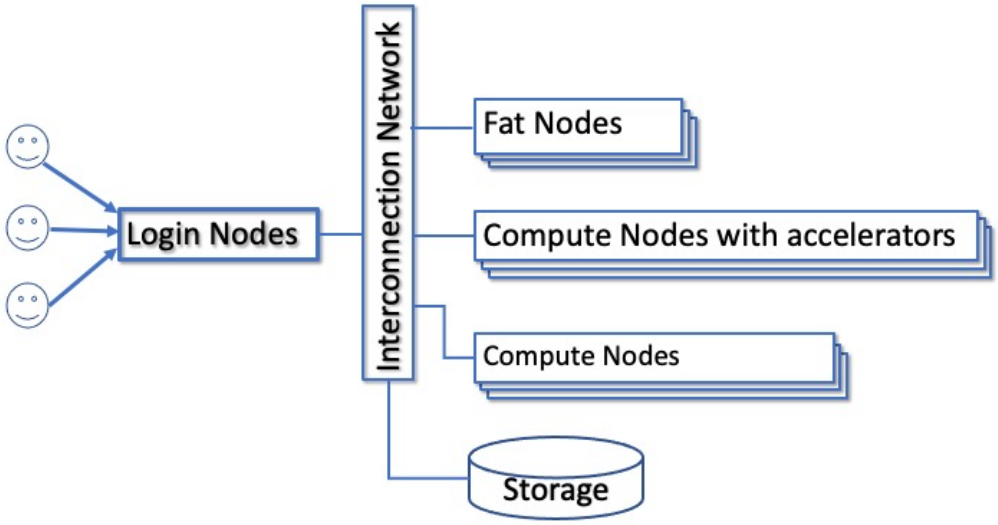
- Power consumption of CPU: ~f^3



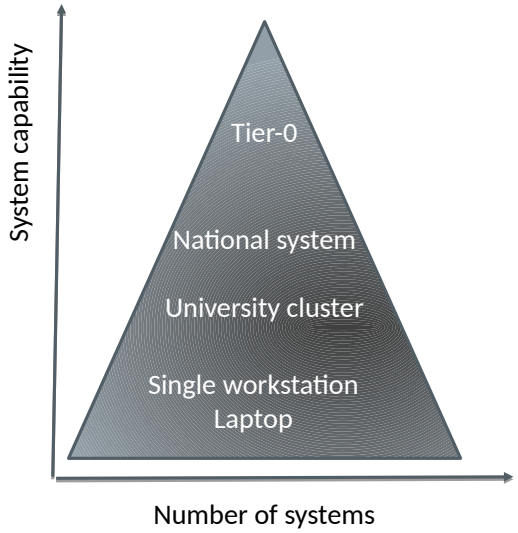
Parallel processing

- Modern (super)computers rely on parallel processing
- Nodes **multiple** CPU cores
  - Latency ~1 µs, bandwidth ~200 Gb / s
  - #1 system has ~10 000 000 cores
- Vectorization
  - Single instruction can process multiple data (SIMD)
- Pipelining
  - Core executes different parts of instructions in parallel
- Accelerators
  - GPUs

Anatomy of supercomputer



From laptop to Tier-0



- The most fundamental difference between a small university cluster and Tier-0 supercomputer is the number of nodes
  - The interconnect in high end systems is often also more capable

Cloud computing

- Cloud infrastructure is run on top of normal HPC system:
  - Shared memory nodes connected by network
- User obtains **virtual** machines
- Infrastructure as a service (IaaS)
  - User has full freedom (and responsibility) of operating system and the whole software environment
- Platform as a service (PaaS)
  - User develops and runs software within the provided environment

Cloud computing and HPC

- Suitability of cloud computing for HPC depends heavily on application
  - Single node performance is often ok
- Virtualization adds overhead especially for the networking
  - Some providers offer high-speed interconnects with a higher price
- Moving data out from the cloud can be expensive
- Currently, cloud computing is not very cost-effective solution for most large scale HPC simulations

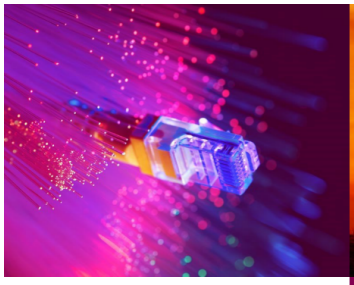
Future of High-performance computing

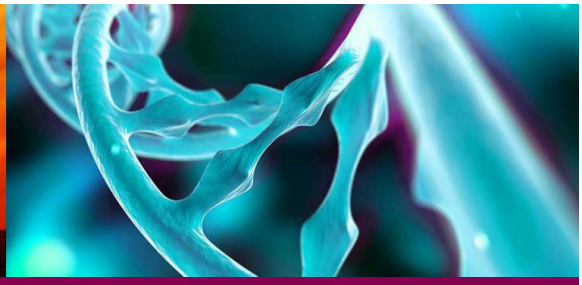
Exascale challenge

- Performance of supercomputers has increased exponentially for a long time
- However, there are unprecedented challenges in reaching exascale (1×10^{18} flop/s)
  - Power consumption: scaling current #1 energy efficient system would still require ~60 MW
  - Fault tolerance: with current technology exascale system experiences hardware failure every few hours
  - Application scalability: how to program for 100 000 000 cores?

Quantum computing

- Quantum computers can solve certain types of problems exponentially faster than classical computers
- General purpose quantum computer is still far away
- For optimisation problems, D-Wave computer based on quantum annealing is already commercially available





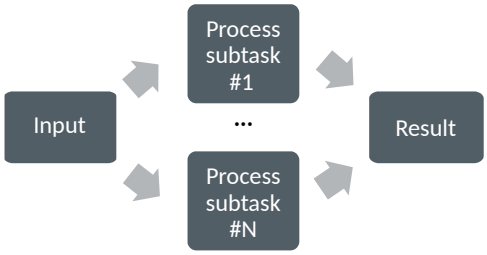
Parallel computing concepts

CSC Training, 2021


CSC – Finnish expertise in ICT for research, education and public administration

Computing in parallel

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

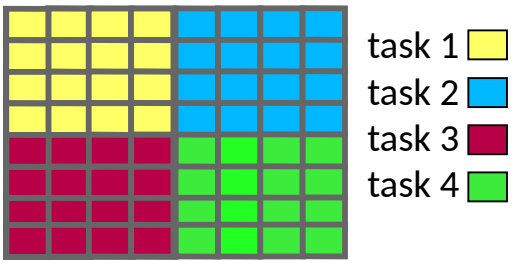


Types of parallel problems

- Tightly coupled
  - Lots of interaction between subtasks
  - Weather simulation
  - Low latency, high speed interconnect is essential
- Embarrassingly parallel
  - Very little (or no) interaction between subtasks
  - Sequence alignment queries for multiple independent sequences in bioinformatics

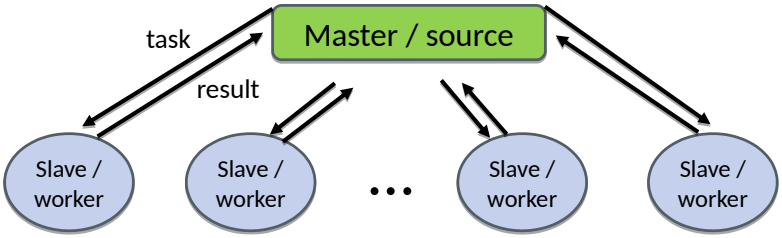
Exposing parallelism

- Data parallelism
  - Data is distributed across cores
  - Each core performs simultaneously (nearly) identical operations with different data
  - Cores may need to interact with each other, e.g. exchange information about data on domain boundaries



Exposing parallelism

- Task farm (master / worker)



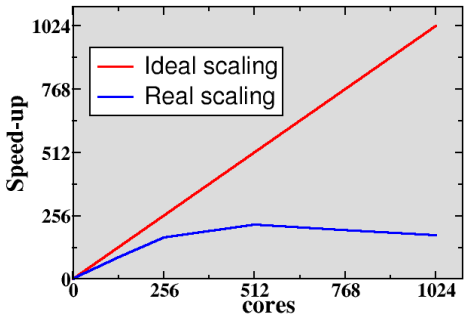
- Master sends tasks to workers and receives results
- There are normally more tasks than workers, and tasks are assigned dynamically

Exposing parallelism

- MapReduce is special task farm like approach suitable especially for large scale data analysis
- Two types of workers
  - Map: given an input data, emit list of (key, value) pairs
  - Reduce: combine the values for a key
- User works only with serial code for Map and Reduce operations
- MapReduce framework takes care of parallelization and data distribution

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



What limits parallel scaling

- Load imbalance
  - Distribution of workload to different cores varies
- Parallel overheads
  - Additional operations which are not present in serial calculation
  - Synchronization, redundant computations, communications
- Amdahl's law: the fraction of non-parallelizable parts establishes the limit on how many cores can be harnessed

$$S(N) = \frac{1}{(1-s)/N + s}, \quad \text{s=non-parallel fraction}$$

Parallel programming

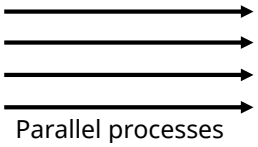
Programming languages

- The de-facto standard programming languages in HPC are (still!) C/C++ and Fortran
- Higher level languages like Python and Julia are gaining popularity
  - Often computationally intensive parts are still written in C/C++ or Fortran
- For some applications there are high-level frameworks with interfaces to multiple languages
  - Petsc, Trilinos, Kokkos
  - TensorFlow for deep learning
  - Spark for MapReduce

Parallel programming models

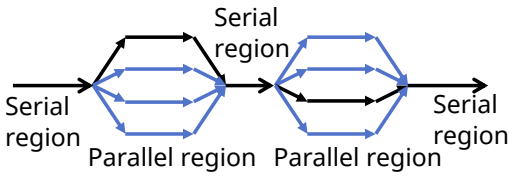
- Parallel execution is based on threads or processes (or both) which run at the same time on different CPU cores
- Processes
  - Interaction is based on exchanging messages between processes
  - MPI (Message passing interface)
- Threads
  - Interaction is based on shared memory, i.e. each thread can access directly other threads data
  - OpenMP, pthreads

Parallel programming models



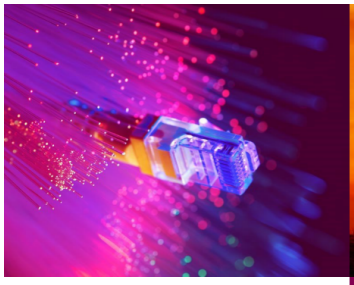
MPI: Processes

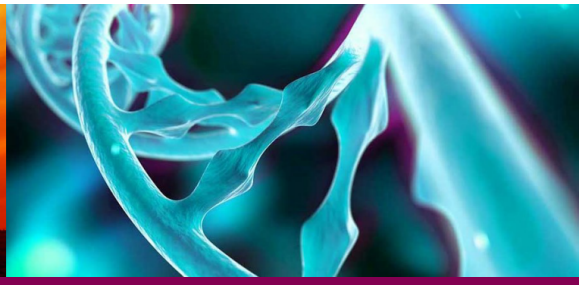
- Independent execution units
- MPI launches N processes at application startup
- Works over multiple nodes



OpenMP: Threads

- Threads share memory space
- Threads are created and destroyed (parallel regions)
- Limited to a single node





Heat equation

CSC Training, 2021

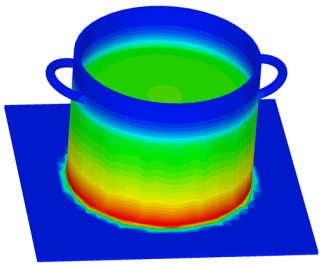

CSC – Finnish expertise in ICT for research, education and public administration

Heat equation

- Partial differential equation that describes the variation of temperature in a given region over time

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

- Temperature variation: $u(x, y, z, t)$
- Thermal diffusivity constant: α

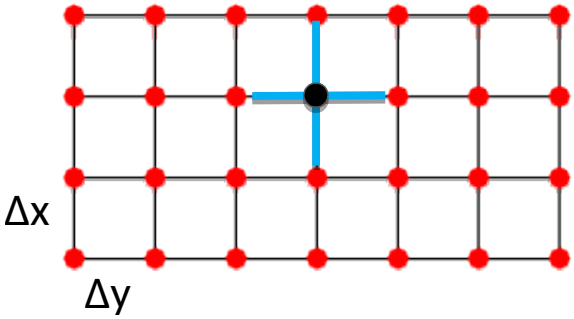


Numerical solution

- Discretize: Finite difference Laplacian in two dimensions

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

Temperature field $u(i,j)$



Time evolution

- Explicit time evolution with time step Δt

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t \alpha \nabla^2 u^m(i,j)$$

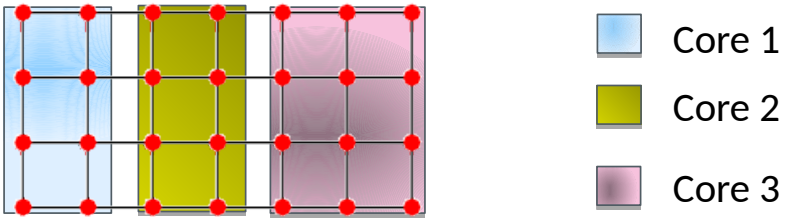
- Note: algorithm is stable only when

$$\Delta t < \frac{1}{2\alpha}\frac{(\Delta x \Delta y)^2}{(\Delta x)^2(\Delta y)^2}$$

- Given the initial condition ($u(t=0)=u^0$) one can follow the time evolution of the temperature field

Solving heat equation in parallel

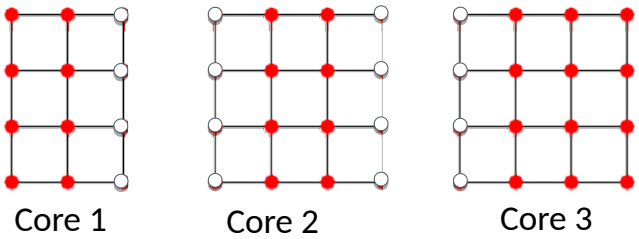
- Temperature at each grid point can be updated independently
- Domain decomposition



- Straightforward in shared memory computer

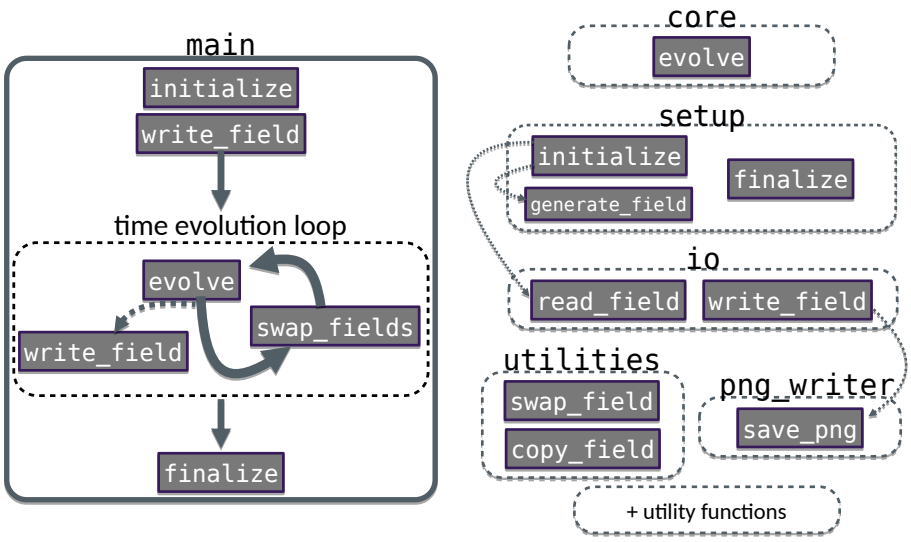
Solving heat equation in parallel

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

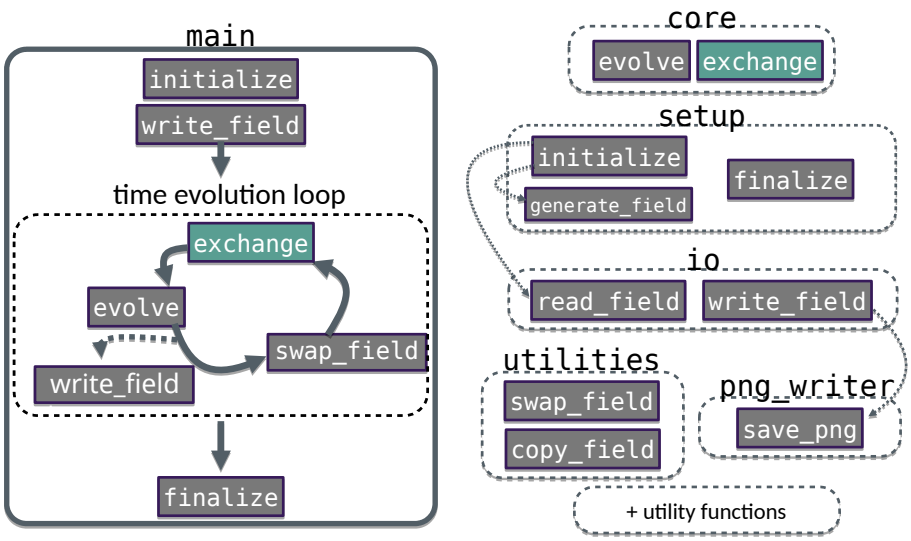


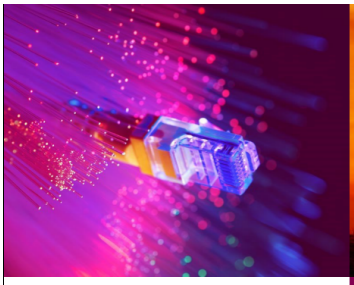
- Before each update cycle, CPU cores communicate boundary data: halo exchange

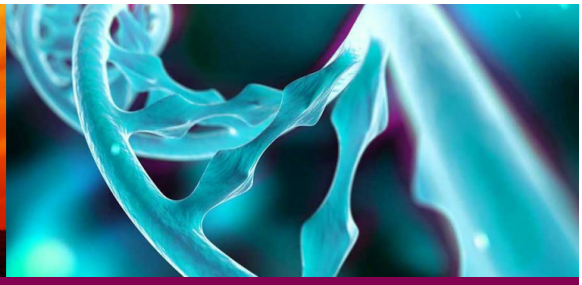
Serial code structure



Parallel code structure







Message-Passing Interface (MPI)

CSC Training, 2021

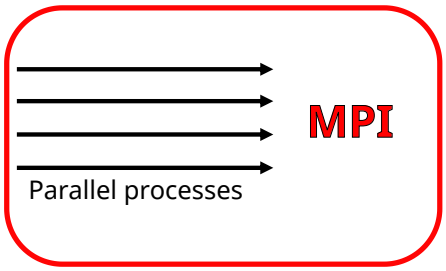
CSC – Finnish expertise in ICT for research, education and public administration

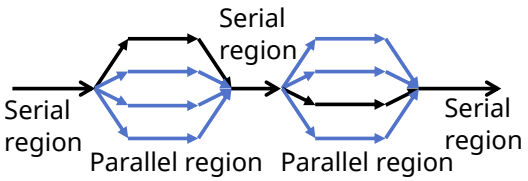
Basic concepts in MPI

Message-passing interface

- MPI is an application programming interface (API) for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from laptops to supercomputers
- MPI is flexible and comprehensive
  - large (hundreds of procedures)
  - concise (only 10-20 procedures are typically needed)

Processes and threads





Process

- Independent execution units
- Have their own state information and *own memory* address space

Thread

- A single process may contain multiple threads
- Have their own state information, but *share* the *same memory* address space

Execution model in MPI

- Normally, parallel program is launched as set of *independent*, *identical processes*
  - execute the *same program code* and instructions
  - processes can reside in different nodes (or even in different computers)
- The way to launch parallel program depends on the computing system
  - **`mpiexec`**, **`mpirun`**, **`srun`**, **`aprun`**, …
  - **`srun`** on puhti.csc.fi and mahti.csc.fi
- MPI supports also dynamic spawning of processes and launching *different* programs communicating with each other
  - rarely used in HPC systems

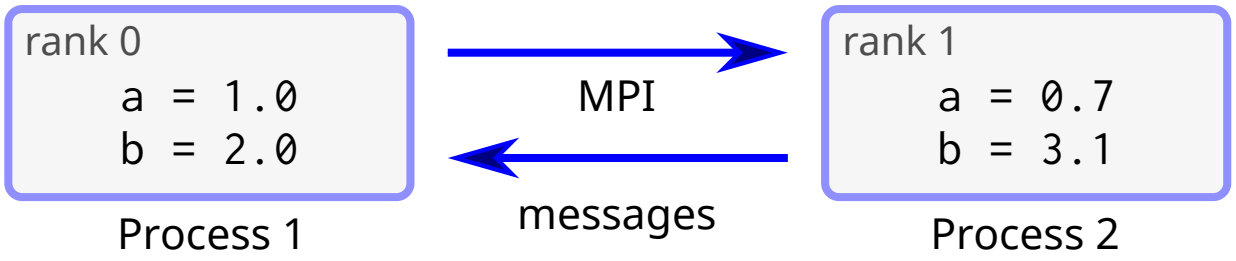
MPI ranks

- MPI runtime assigns each process a unique rank
  - identification of the processes
  - ranks start from 0 and extend to N-1
- Processes can perform different tasks and handle different data based 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



The MPI library

- Information about the communication framework
  - 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
  - Communicator manipulation, user defined datatypes, one-sided communication, ...

MPI communicator

- Communicator is an object connecting a group of processes, i.e. the communication framework
- Most MPI functions require communicator as an argument
- Initially, there is always a communicator **MPI_COMM_WORLD** which contains all the processes
- Users can define custom communicators

Programming MPI

- The MPI standard defines interfaces to C and Fortran programming languages
  - No C++ bindings in the standard, C++ programs use the C interface
  - There are unofficial bindings to eg. Python, Perl and Java
- C call convention (*case sensitive*)

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

  - some arguments have to be passed as pointers
- Fortran call convention (*case insensitive*)

```
call mpi_xxxx(parameter,..., rc)
```

  - return code in the last argument

Writing an MPI program

- C: include the MPI header file

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

- Fortran: use MPI module

```
use mpi_f08
```

(older Fortran codes might have `use mpi` or `include 'mpif.h'`)

- Start by calling the routine **MPI_Init**
- Write the program
- Call **MPI_Finalize** before exiting from the main program

Compiling an MPI program

- MPI is a library (+ runtime system)
- In principle, MPI programs can be build with standard compilers (*i.e.* `gcc` / `g++` / `gfortran`) with the appropriate `-I` / `-L` / `-l` options
- Most MPI implementations provide convenience wrappers, typically `mpicc` / `mpicxx` / `mpif90`, for easier building
  - no need for MPI related options

```
mpicc -o my_mpi_prog my_mpi_code.c
mpicxx -o my_mpi_prog my_mpi_code.cpp
mpif90 -o my_mpi_prog my_mpi_code.F90
```

Presenting syntax

- MPI calls are presented as pseudocode
  - actual C and Fortran interfaces are given in reference section
  - Fortran error code argument not included

`MPI_Function(arg1, arg2)`

`arg1`

input arguments in red

`arg2`

output arguments in blue. Note that in C the output arguments are always pointers

First five MPI commands: Initialization and finalization

`MPI_Init`

(in C `argc` and `argv` pointer arguments are needed)

`MPI_Finalize`

First five MPI commands: Information about the communicator

```
MPI_Comm_size(comm, size)
```

comm
communicator

size
number of processes in the communicator

```
MPI_Comm_rank(comm, rank)
```

comm
communicator

rank
rank of this process

First five MPI commands: Synchronization

- Wait until everybody within the communicator reaches the call

```
MPI_Barrier(comm)
```

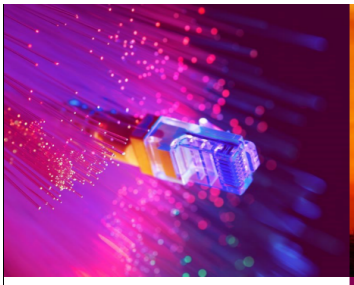
comm
communicator

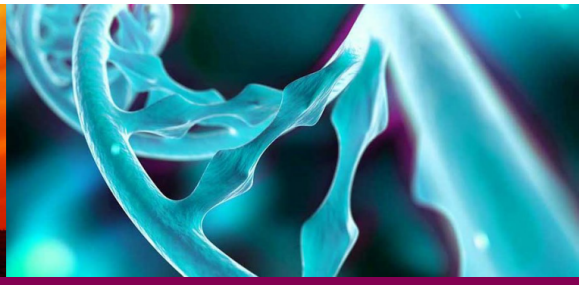
Summary

- In parallel programming with MPI, the key concept is a set of independent processes
- Data is always local to the process
- Processes can exchange data by sending and receiving messages
- The MPI library contains functions for communication and synchronization between processes

Web resources

- List of MPI functions with detailed descriptions
  http://mpi.deino.net/mpi_functions/index.htm
- Good online MPI tutorials
  - https://computing.llnl.gov/tutorials/mpi
  - http://mpitutorial.com/tutorials/
  - https://www.youtube.com/watch?v=BPSgXQ9aUXY
- MPI 4.0 standard http://www.mpi-forum.org/docs/
- MPI implementations
  - OpenMPI http://www.open-mpi.org/
  - MPICH https://www.mpich.org/





Point-to-point communication

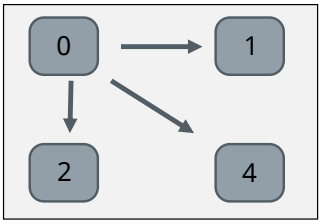
CSC Training, 2021

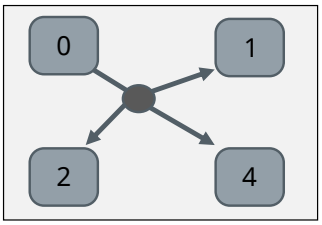
CSC – Finnish expertise in ICT for research, education and public administration

Point-to-point communication

Communication

- Data is local to the MPI processes
  - They need to *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

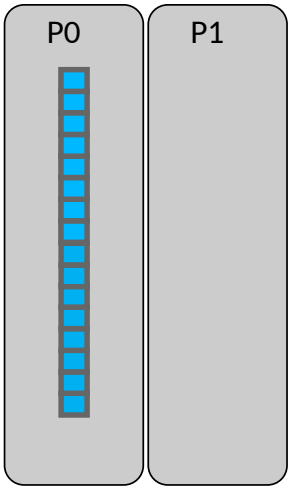




MPI point-to-point operations

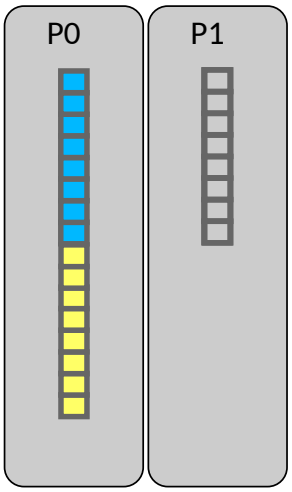
- One process *sends* a message to another process that *receives* it with `MPI_Send` and `MPI_Recv` routines
- Sends and receives in a program should match – one receive per send
- Each message (envelope) contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

Case study: parallel sum

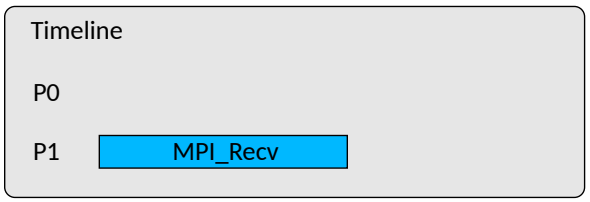


- Array initially on process #0 (P0)
- Parallel algorithm
  - **Scatter**
    Half of the array is sent to process 1
  - **Compute**
    P0 & P1 sum independently their segments
  - **Reduction**
    Partial sum on P1 sent to P0 P0 sums the partial sums

Case study: parallel sum

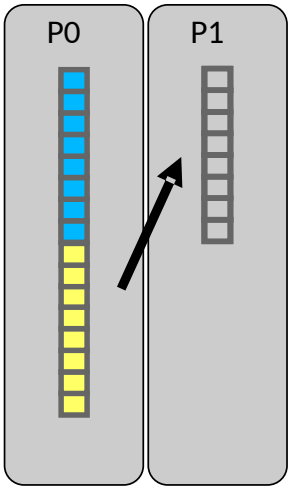


Step 1.1: Receive call in scatter

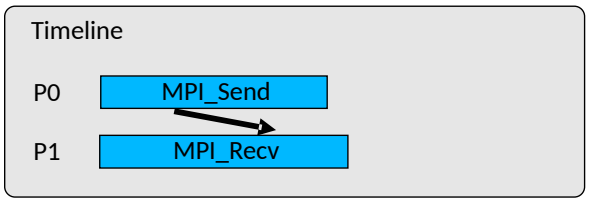


P1 issues MPI_Recv to receive half of the array from P0

Case study: parallel sum

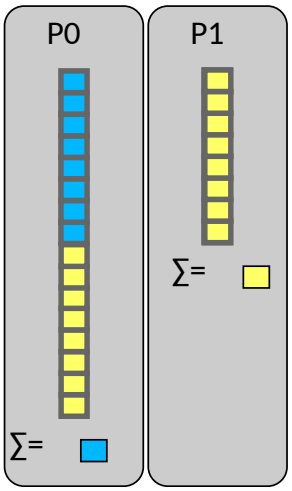


Step 1.2: Send call in scatter

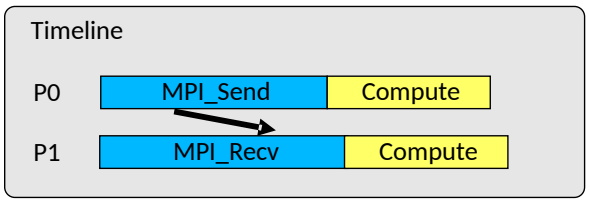


Po issues an MPI_Send to send the lower part of the array to P1

Case study: parallel sum

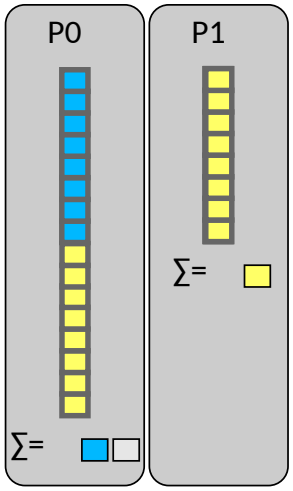


Step 2: Compute the sum in parallel

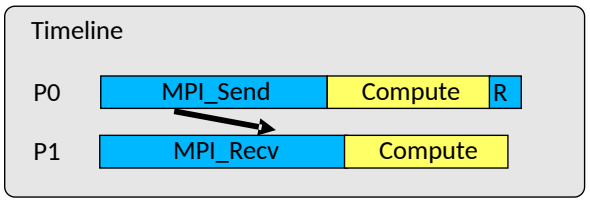


Both Po & P1 compute their partial sums and store them locally

Case study: parallel sum

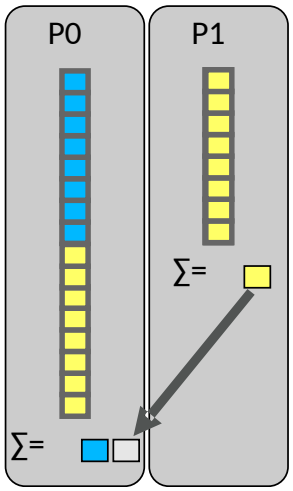


Step 3.1: Receive call in reduction

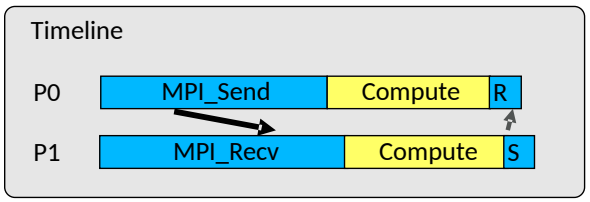


Po issues an MPI_Recv operation for receiving P1's partial sum

Case study: parallel sum

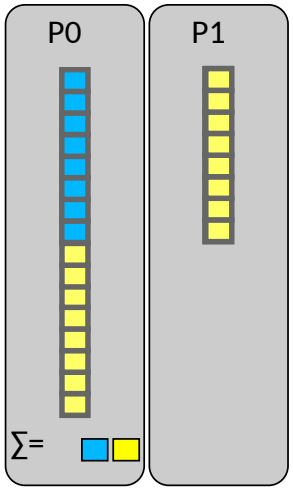


Step 3.2: Send call in reduction

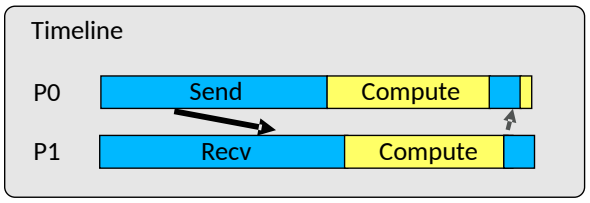


P1 sends the partial sum to Po

Case study: parallel sum



Step 3.3: compute the final answer



Po computes the total sum

Send operation

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

buffer
The data to be sent

count
Number of elements in buffer

datatype
Type of elements in buffer (see later slides)

dest
The rank of the receiver

tag
An integer identifying the message

comm
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(buffer, count, datatype, source, tag, comm, status)`

`buffer`
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 buffer

`source`
Sender of the message

`tag`
Number identifying the message

`comm`
Communicator

`status`
Information on the received message

`error`
As for send operation

"Buffers" in MPI

- The "buffer" arguments are memory addresses
- MPI assumes contiguous chunk of memory
  - `count` elements are send starting from the address
  - received elements are stored starting from the address
- In Fortran, arguments are passed by reference, *i.e.* variables can be passed as such
  - Note: be careful if passing non-contiguous array segmens such as `a(1, 1:N)`
- In C/C++ "buffer" is pointer
  - `data()` method of C++ `<array>` and `<vector>` containers can be used

MPI datatypes

- On low level, MPI sends and receives stream of bytes
- MPI datatypes specify how the bytes should be interpreted
  - Allows data conversios in heterogenous environments (*e.g.* little endian to big endian)
- MPI has a number of predefined basic datatypes corresponding to C or Fortran datatypes
  - C examples: `MPI_INT` for `int` and `MPI_DOUBLE` for `double`
  - Fortran examples: `MPI_INTEGER` for `integer`, `MPI_DOUBLE_PRECISION` for `real64`
- One can also define custom datatypes for communicating more complex data

Blocking routines & deadlocks

- MPI_Send and MPI_Recv are blocking routines
  - `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
- Completion depends on other processes => risk for *deadlocks*
  - For example, all processes are in `MPI_Recv`
  - If deadlocked, the program is stuck forever

Status parameter

- The status parameter in `MPI_Recv` contains information about the received data after the call has completed, e.g.
  - Number of received elements
  - Tag of the received message
  - Rank of the sender
- In C the status parameter is a struct
- In Fortran the status parameter is of type `mpi_status`
  - Old interface: integer array of size `MPI_STATUS_SIZE`

Status parameter

- Received elements

  Use the function

  `MPI_Get_count(status, datatype, count)`
- Tag of the received message

  C: `status.MPI_TAG`

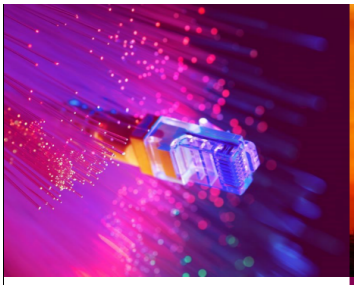
  Fortran: `status%mpi_tag` (old version `status(MPI_TAG)`)
- Rank of the sender

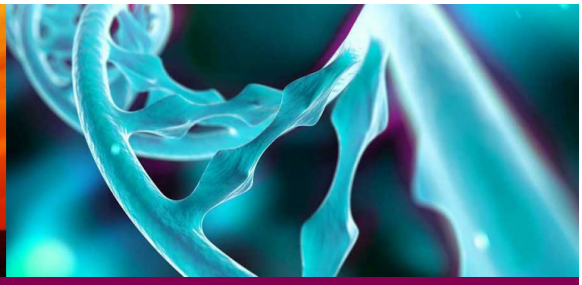
  C: `status.MPI_SOURCE`

  Fortran: `status%mpi_source` (old version `status(MPI_SOURCE)`)

Summary

- Point-to-point communication = messages are sent between two MPI processes
- Point-to-point operations enable any parallel communication pattern (in principle)
  - `MPI_Send` and `MPI_Recv`
- Status parameter of `MPI_Recv` contains information about the message after the receive is completed





Parallel debugging

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Parallel debugging

Debugging

- Bugs are evident in any non-trivial program
- Crashes (Segmentation fault) or incorrect results
- Parallel programs can have also deadlocks and race conditions

Finding bugs

- Print statements in the code
  - Typically cumbersome, especially with compiled languages
  - Might result in lots of clutter in parallel programs
  - Order of printouts from different processes is arbitrary
- "Standard" debuggers
  - gdb: common command line debugger
  - Debuggers within IDEs, e.g. VS Code
  - No proper support for parallel debugging
- Parallel debuggers
  - **Allinea DDT**, Totalview (commercial products)

Common features in debuggers

- Setting breakpoints and watchpoints
- Executing code line by line
- Stepping into / out from functions
- Investigating values of variables
- Investigating program stack
- Parallel debuggers allow all of the above on per process/thread basis

Web resources

- Defensive programming and debugging online course https://www.futurelearn.com/courses/defensive-programming-and-debugging
- Using gdb for parallel debugging https://www.open-mpi.org/faq/?category=debugging
- Memory debugging with Valgrind https://valgrind.org/docs/manual/mc-manual.html#mc-manual.mpiwrap

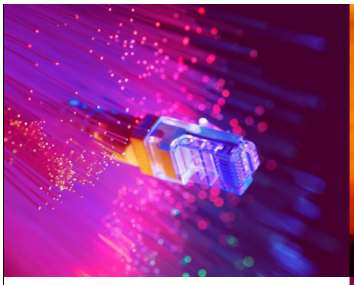
Demo: using Allinea DDT

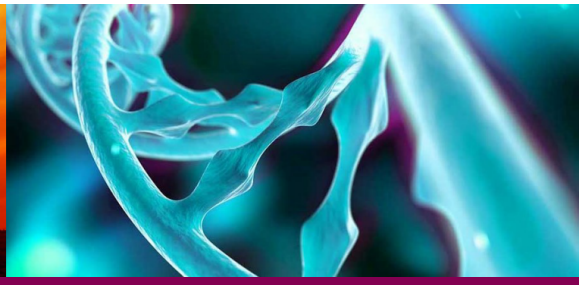
Using Allinea DDT

- Code needs to be compiled with debugging option `-g`
- Compiler optimizations might complicate debugging (dead code elimination, loop transformations, *etc.*), recommended to compile without optimizations with `-O0`
  - Sometimes bugs show up only with optimizations
- In CSC environment DDT is available via `module load ddt`
- Debugger needs to be started in an interactive session

```
module load ddt
export SLURM_OVERLAP=1
salloc --nodes=1 --ntasks-per-node=2 --account=project_xxx -p small
ddt srun ./buggy
```

- NoMachine remote desktop is recommended for smoother GUI performance





Special MPI parameters

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

"Special" parameters

MPI programming practices

- For the sake of illustration, we have so far hard-coded the `source` and `destination` arguments in the MPI calls, and placed the MPI calls within `if` constructs
- This produces typically code which is difficult to read and to generalize to arbitrary number of processes
- Store source and destination in variables and place MPI calls outside **if**s when possible.

```
 if ( myid == 0 ) then
    call mpi_send(message, msgsize, MPI_INTEGER, 1, &
         1, MPI_COMM_WORLD, rc)
    call mpi_recv(receiveBuffer, arraysize, MPI_INTEGER, 1,  &
         1, MPI_COMM_WORLD, status, rc)
 else if (myid == 1) then
    call mpi_send(message, msgsize, MPI_INTEGER, 0, &
         1, MPI_COMM_WORLD, rc)
    call mpi_recv(receiveBuffer, arraysize, MPI_INTEGER, 0,  &
         1, MPI_COMM_WORLD, status, rc)
```

```
! Modulo operation can be used for wrapping around
dst = mod(myid + 1, ntasks)
src = mod(myid - 1 + ntasks, ntasks)

call mpi_send(message, msgsize, MPI_INTEGER, dst, &
        1, MPI_COMM_WORLD, rc)
call mpi_recv(receiveBuffer, arraysize, MPI_INTEGER, src,  &
        1, MPI_COMM_WORLD, status, rc)
```


MPI programming practices

- As rank 0 is always present even in the serial case, it is normally chosen as the special task in scatter and gather type operations

```
if (0 == myid) {
  for (int i=1; i < ntasks; i++) {
     MPI_Send(&data, 1, MPI_INT, i, 22, MPI_COMM_WORLD);
  }
} else {
     MPI_Recv(&data, 1, MPI_INT, 0, 22, MPI_COMM_WORLD, &status);
}
```

Coping with boundaries

- In some communication patterns there are boundary processes that do not send or receive while all the other processes do
- A special constant `MPI_PROC_NULL` can be used for turning `MPI_Send` / `MPI_Recv` into a dummy call
  - No matching `receive` / send is needed

```
if (myid == 0) then
    src = MPI_PROC_NULL
end if
if (myid == ntasks - 1) then
    dst = MPI_PROC_NULL
end if

call mpi_send(message, msgsize, MPI_INTEGER, dst, ...
call mpi_recv(message, msgsize, MPI_INTEGER, src, ...
```


Arbitrary receives

- In some communication patterns one might want to receive from arbitrary sender or a message with arbitrary tag
- `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
  - The actual sender and tag can be queried from **status** if needed

```
if (0 == myid) {
  for (int i=1; i < ntasks; i++) {
     MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 22, MPI_COMM_WORL
     process(data)
  }
} else {
     MPI_Send(&data, 1, MPI_INT, 0, 22, MPI_COMM_WORLD, &status);
}
```

- There needs to be still `receive` for each send
- `MPI_ANY_SOURCE` and `MPI_ANY_TAG` introduce often performance overhead
- Use them only when there is clear benefit *e.g.* in load balancing

Ignoring status

- When `source`, `tag`, and number of received elements are known, there is no need to examine `status`
- A special constant `MPI_STATUS_IGNORE` can be used for the `status` parameter
- Saves memory in the user program and allows optimizations in the MPI library

Special parameter values in sending

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

Parameter	Special value	Implication
`dest`	`MPI_PROC_NULL`	Null destination, no operation takes place

Special parameter values in receiving

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

Parameter	Special value	Implication
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
status	MPI_STATUS_IGNORE	Do not store any status data

Summary

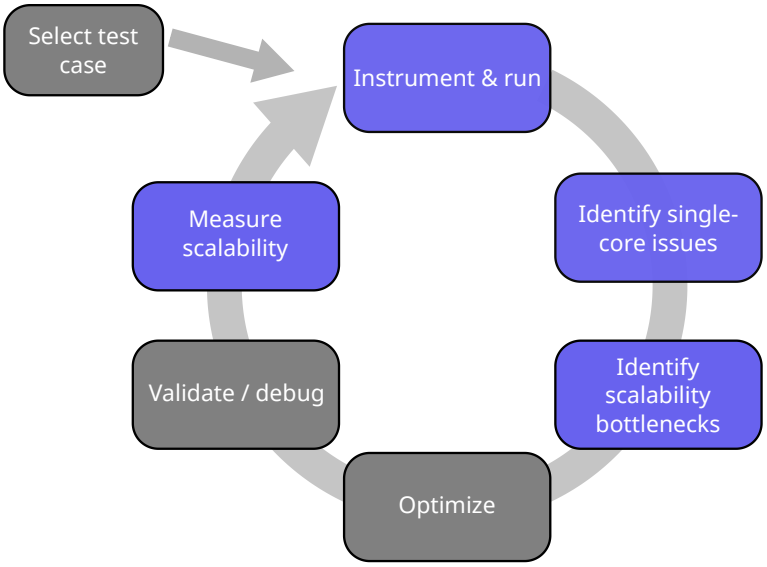
- Generally, it is advisable to make MPI programs to work with arbitrary number of processes
- When possible, MPI calls should be placed outside `if` constructs
- Employing special parameter values may simplify the implementations of certain communication patterns

Performance analysis

CSC Training, 2021

Performance analysis

Performance analysis



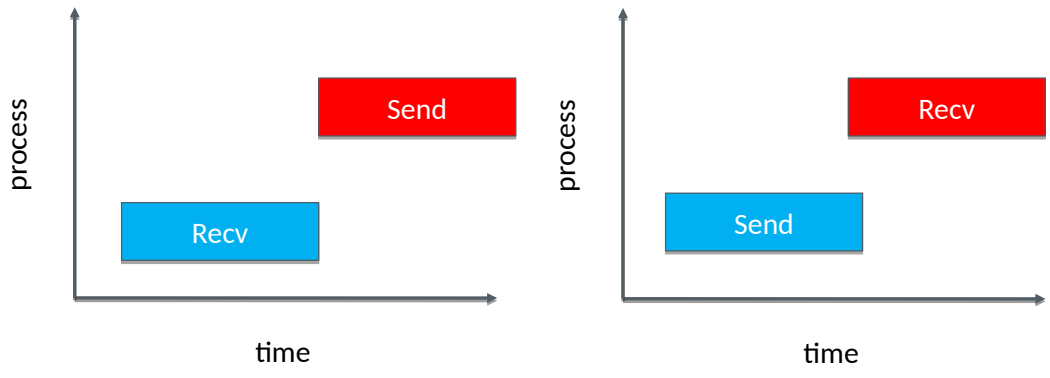
Introduction

- Finding out the scalability bottlenecks of parallel application is non-trivial
- Bottlenecks can be very different with few CPUs than with thousands of CPUs
- Performance analysis needs to be carried in scalability limit with large enough test case
- Efficient tools are needed for the analysis of massively parallel applications

Potential scalability bottlenecks

- Bad computation to communication ratio
  - In 2D heat equation with one dimensional parallelization $\frac{T_{comp}}{T_{comm}} \sim \frac{N}{p}$
- Load imbalance
- Synchronization overhead
- Non-optimal communication patterns

Common inefficient communication patterns



Late sender

Late receiver

Measuring performance

- Timing routines within the application
  - MPI includes high resolution timer `MPI_Wtime`
  - Can be useful for big picture

```
double t0 = MPI_Wtime()
do_something()
double t1 = MPI_Wtime()
printf("Elapsed time %f\n", t1 - t0);
```

- Dedicated parallel performance analysis tools
- ScoreP / Scalasca, Extrae / Paraver, Intel Trace Analyzer, TAU, CrayPAT, Vampir

MPI performance analysis tools

- Many tools can automatically identify common problematic communication patterns
- Flat profile
  - Time spent in MPI calls during the whole program execution
- Trace
  - Also the temporal profile of MPI calls
  - Potentially huge data
    - Filtering only the most relevant calls may be needed

Demo: Trace analysis

Summary

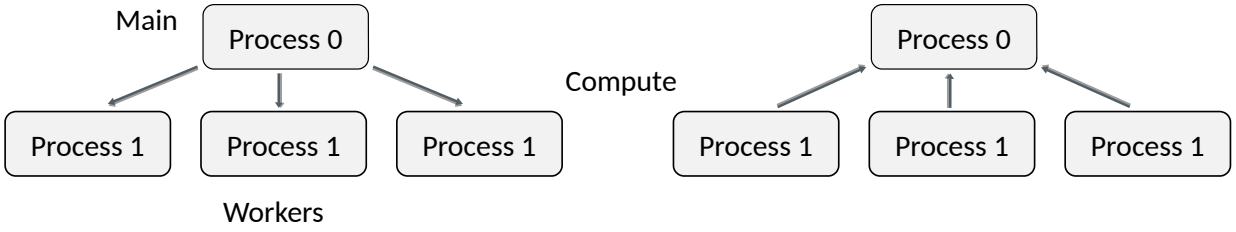
- Many tools can provide information about MPI performance problems
  - Manual investigation impossible in massively parallel scale
- Problems often caused by load imbalance or by badly designed communication pattern

Web resources

- https://pop-coe.eu/further-information/learning-material
- http://www.prace-ri.eu/best-practice-guides/

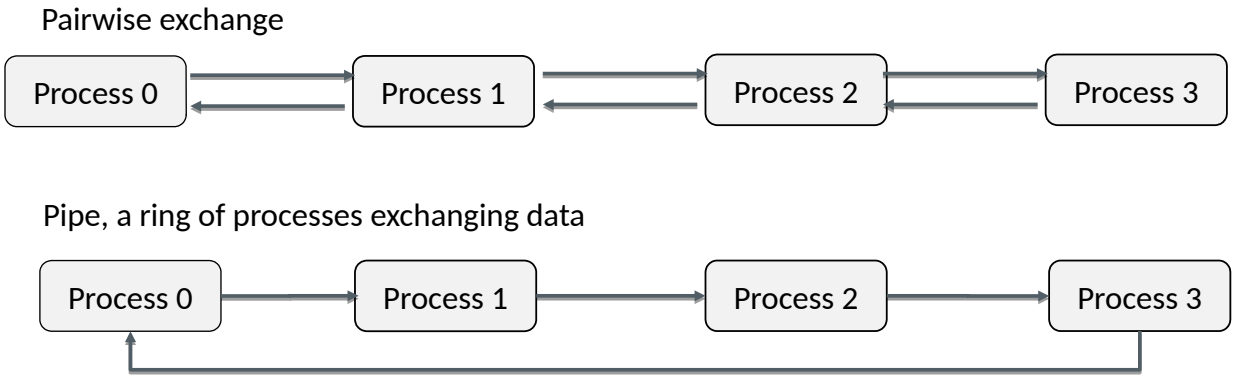
Common communication patterns

Main - worker



- Each process is only sending or receiving at the time

Pairwise neighbour communication



- Incorrect ordering of sends/receives may give a rise to a deadlock (or unnecessary idle time)
- Can be generalized to multiple dimensions

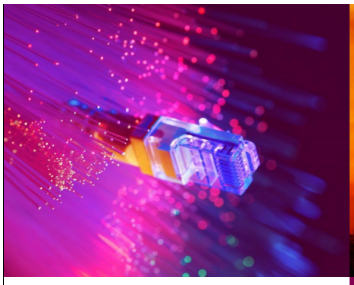
Combined send & receive

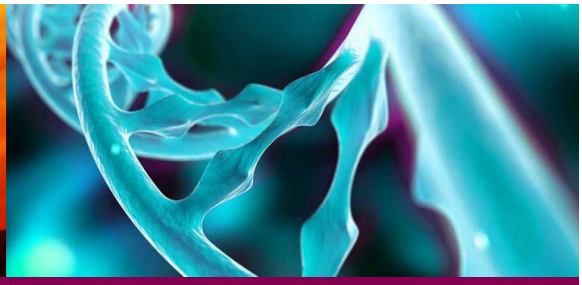
```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
recvcount, recvtype, source, recvtag, comm, status)
```

- Sends one message and receives another one, with a single command
  - Reduces risk for deadlocks and improves performance
- Parameters as in `MPI_Send` and `MPI_Recv`
- Destination rank and source rank can be same or different
- `MPI_PROC_NULL` can be used for coping with the boundaries

Summary

- Individual `MPI_Send` and `MPI_Recv` are suitable for irregular communication
- When there is always both sending and receiving, `MPI_Sendrecv` can prevent deadlocks and serialization of communication





Collective communication

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Collective communication

Introduction

- Collective communication transmits data among all processes in a process group (communicator)
- Collective communication includes
  - data movement
  - collective computation
  - synchronization

Introduction

- Collective communication typically outperforms 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)
```

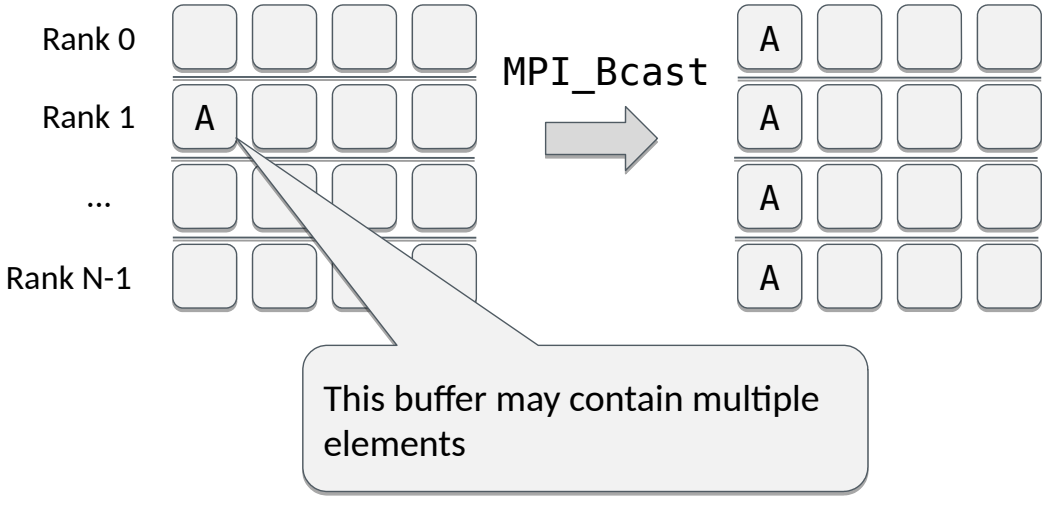
Communicating a vector **a** consisting of 1M float elements from the task 0 to all other tasks

Introduction

- These routines *must be called by all the processes* in the communicator
- Amount of sent and received data must match
- No tag arguments
  - Order of execution must coincide across processes

Broadcasting

- Replicate data from one process to all others



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

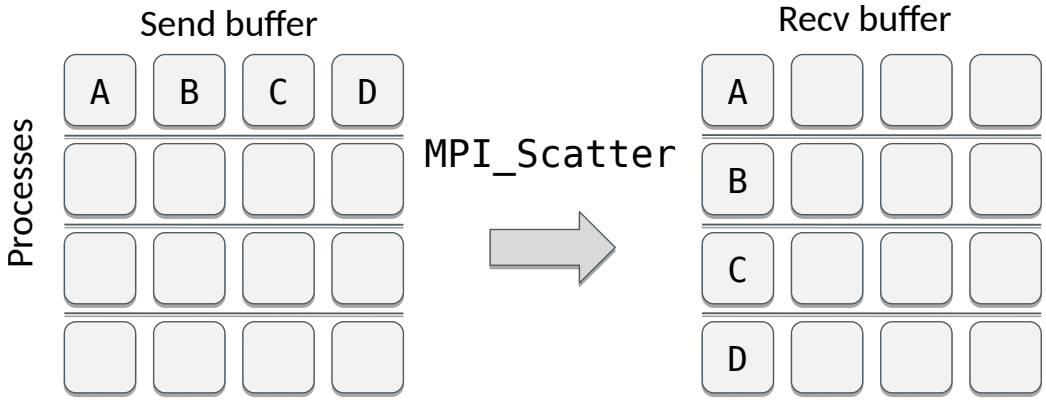
datatype
data type of buffer

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

- Task root sends an equal share of data 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 to receive at each process

recvtype
data type of receive buffer elements

root
rank of sending process

comm
communicator

Examples

Assume 4 MPI tasks. What would the (full) program print?

```
if (my_id==0) then
    do i = 1, 16
        a(i) = i
    end do
end if
call mpi_bcast(a, 16, MPI_INTEGER, 0, &
        MPI_COMM_WORLD, rc)
if (my_id==3) print *, a(:)
```

A) 1 2 3 4

B) 13 14 15 16

C) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

```
if (my_id==0) then
    do i = 1, 16
        a(i) = i
    end do
end if
call mpi_scatter(a, 4, MPI_INTEGER, aloc, 4 &
    MPI_INTEGER, 0, MPI_COMM_WORLD, rc)
if (my_id==3) print *, aloc(:)
```

A) 1 2 3 4

B) 13 14 15 16

C) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Vector version of MPI_Scatter

`MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)`

sendbuf
send buffer

sendcounts
array (of length ntasks) specifying the number of elements to send to each processor

displs
array (of length ntasks). Entry i specifies the displacement(relative to sendbuf)

sendtype
data type of send buffer elements

recvbuf
receive buffer

recvcount
number of elements to receive

recvtype
data type of receive buffer elements

root
rank of sending process

comm
communicator

Scatterv example

```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
end if

scounts(0:3) = [ 1, 2, 3, 4 ]
displs(0:3) = [ 0, 1, 3, 6 ]

call mpi_scatterv(a, scounts, &
    displs, MPI_INTEGER, &
    aloc, scounts(my_id), &
    MPI_INTEGER, 0, &
    MPI_COMM_WORLD, rc)
```

Assume 4 MPI tasks. What are the values in aloc in the last task (#3)?

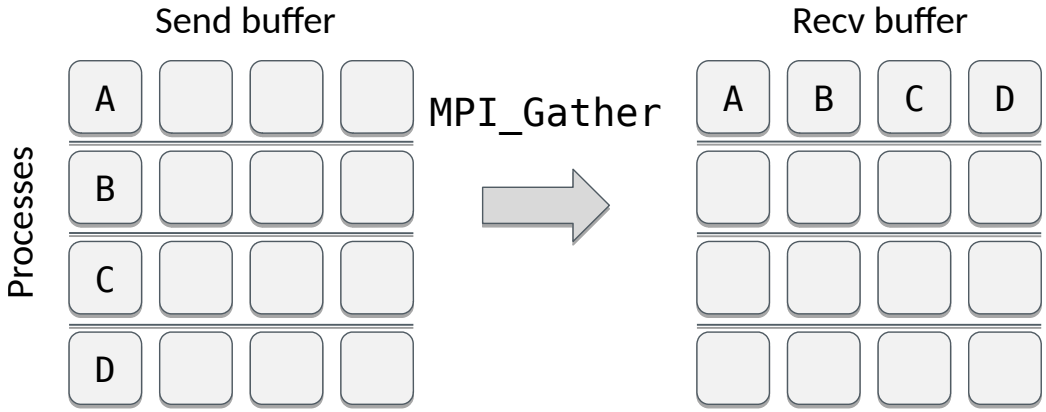
A) 1 2 3

B) 7 8 9 10

C) 1 2 3 4 5 6 7 8 9 10

Gathering data

- Collect data from all the process to one process



- Segments A, B, ... may contain multiple elements

Gathering data

- `MPI_Gather`: Collect an equal share of data from all processes to root

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

sendbuf
send buffer (data to be gathered)

sendcount
number of elements pulled from each process

sendtype
data type of send buffer elements

recvbuf
receive buffer

recvcount
number of elements in any single receive

recvtype
data type of receive buffer elements

root
rank of receiving process

comm
communicator

Vector version of MPI_Gather

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

sendbuf
send buffer

sendcount
the number of elements to send

sendtype
data type of send buffer elements

recvbuf
receive buffer

recvcounts
array (of length ntasks). Entry *i* specifies how many to receive from that process

displs
array (of length ntasks). Entry *i* specifies the displacement (relative to recvbuf)

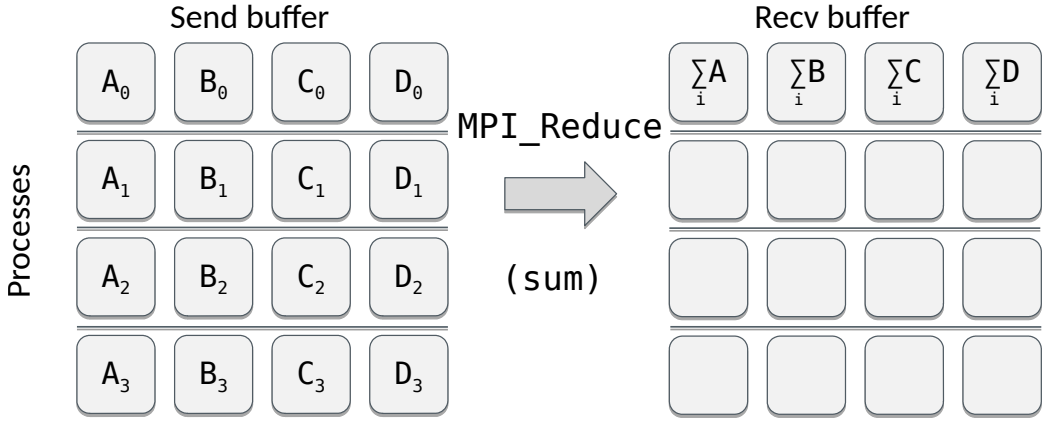
recvtype
data type of receive buffer elements

root
rank of receiving process

comm
communicator

Reduce operation

- Applies an operation over set of processes and places result in single process



Available reduction operations

Operation	Meaning
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location

Operation	Meaning
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

Reduce operation

- Applies a reduction operation op to sendbuf over the set of tasks and places the result in recvbuf on root

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

sendbuf
send buffer

recvbuf
receive buffer

count
number of elements in send buffer

datatype
data type of elements in send buffer

op
operation

root
rank of root process

comm
communicator

Global reduction

- MPI_Allreduce combines values from all processes and distributes the result back to all processes
  - Compare: MPI_Reduce + MPI_Bcast

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)
```

sendbuf
starting address of send buffer

recvbuf
starting address of receive buffer

count
number of elements in send buffer

datatype
data type of elements in send buffer

op
operation

comm
communicator

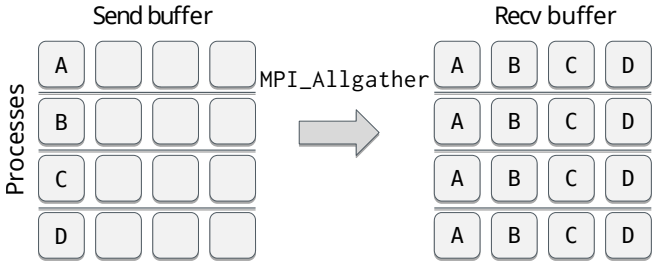
Allreduce example: parallel dot product

```
real :: a(1024), aloc(128)
...
if (my_id==0) then
    call random_number(a)
end if
call mpi_scatter(a, 128, MPI_INTEGER, &
                 aloc, 128, MPI_INTEGER, &
                 0, MPI_COMM_WORLD, rc)
rloc = dot_product(aloc,aloc)
call mpi_allreduce(rloc, r, 1, MPI_REAL,&
                   MPI_SUM, MPI_COMM_WORLD,
                   rc)
```

```
> aprun -n 8 ./mpi_pdot
 id= 6 local= 39.68326  global= 338.8004
 id= 7 local= 39.34439  global= 338.8004
 id= 1 local= 42.86630  global= 338.8004
 id= 3 local= 44.16300  global= 338.8004
 id= 5 local= 39.76367  global= 338.8004
 id= 0 local= 42.85532  global= 338.8004
 id= 2 local= 40.67361  global= 338.8004
 id= 4 local= 49.45086  global= 338.8004
```

All gather

- `MPI_Allgather` gathers data from each task and distributes the resulting data to each task
  - Compare: `MPI_Gather` + `MPI_Bcast`



All gather

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

sendbuf
send buffer

sendcount
number of elements in send buffer

sendtype
data type of send buffer elements

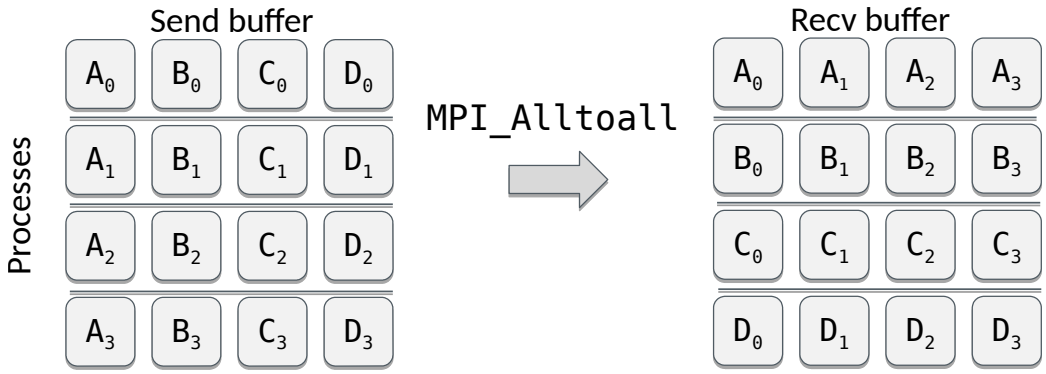
recvbuf
receive buffer

recvcount
number of elements received from any process

recvtype
data type of receive buffer

All to all

- Send a distinct message from each task to every task



- "Transpose" like operation

All to all

- Sends a distinct message from each task to every task
  - Compare: "All scatter"

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

sendbuf
send buffer

sendcount
number of elements to send

sendtype
data type of send buffer elements

recvbuf
receive buffer

recvcount
number of elements received

recvtype
data type of receive buffer elements

comm
communicator

All-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_bcast(a, 16, MPI_INTEGER, 0, &
    MPI_COMM_WORLD, rc)

call mpi_alltoall(a, 4, MPI_INTEGER, &
                  aloc, 4, MPI_INTEGER, &
                  MPI_COMM_WORLD, rc)
```

A) 1, 2, 3, 4

B) 1, ..., 16

C) 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4

Assume 4 MPI tasks. What will be the values of **aloc in the process #0?**

Common mistakes with collectives

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

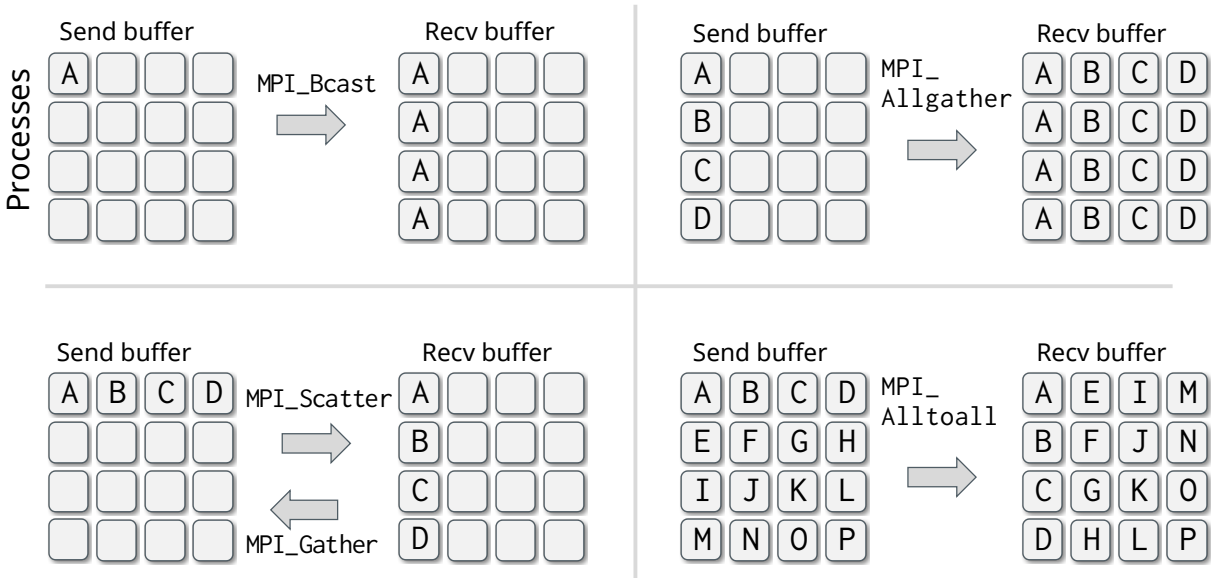
  `if (my_id == 0) call mpi_bcast(...`
  - All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), must call the collective routine!
- Assuming that all processes making a collective call would complete at the same time
- Using the input buffer as the output buffer

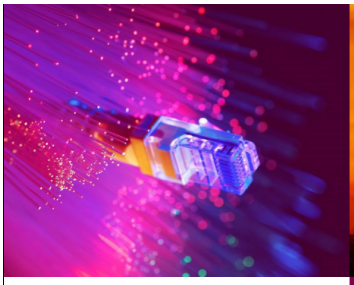
  `call mpi_allreduce(a, a, n, mpi_real,...`
  - One should employ MPI_IN_PLACE for this purpose

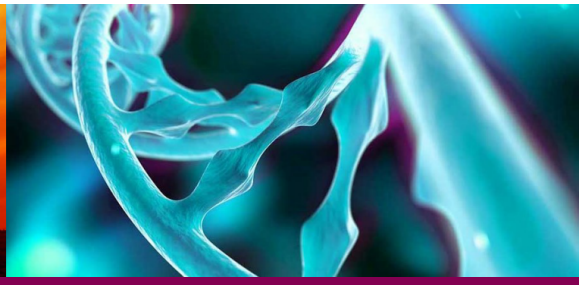
Summary

- 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

Summary







Non-blocking communication

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

Non-blocking communication

Non-blocking communication

- Non-blocking communication operations return immediately and perform sending/receiving in the background
  - `MPI_Isend` & `MPI_Irecv`
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations
- Collective operations are also available as non-blocking versions

Non-blocking send

Parameters similar to **`MPI_Send`** but has an additional request parameter.

```
MPI_Isend(buffer, count, datatype, dest, tag, comm, request)
```

`buffer`

send buffer that must not be written to until one has checked that the operation is over

`request`

a handle that is used when checking if the operation has finished (`type(mpi_request)` in Fortran, `MPI_Request` in C)

Non-blocking receive

Parameters similar to **MPI_Recv** but has no status parameter.

```
MPI_Irecv(buffer, count, datatype, source, tag, comm, request)
```

buffer

receive buffer guaranteed to contain the data only after one has checked that the operation is over

request

a handle that is used when checking if the operation has finished

Non-blocking communication

- Important: Send/receive operations have to be finalized
  - `MPI_Wait`, `MPI_Waitall`, ...
    - Waits for the communication started with `MPI_Isend` or `MPI_Irecv` to finish (blocking)
  - `MPI_Test`, ...
    - Tests if the communication has finished (non-blocking)
  - Remember: successfully finished send does not mean successful receive!
- You can mix non-blocking and blocking routines
  - e.g., receive a message sent by `MPI_Isend` with `MPI_Recv`

Wait for non-blocking operation

```
MPI_Wait(request, status)
```

request
handle of the non-blocking communication

status
status of the completed communication, see `MPI_Recv`

A call to `MPI_Wait` returns when the operation identified by request is complete

Wait for non-blocking operations

```
MPI_Waitall(count, requests, status)
```

count
number of requests

requests
array of requests

status
array of statuses for the operations that are waited for

A call to `MPI_Waitall` returns when all operations identified by the array of requests are complete.

Non-blocking test for non-blocking operations

```
MPI_Test(request, flag, status)
```

request
request

flag
True if the operation has completed

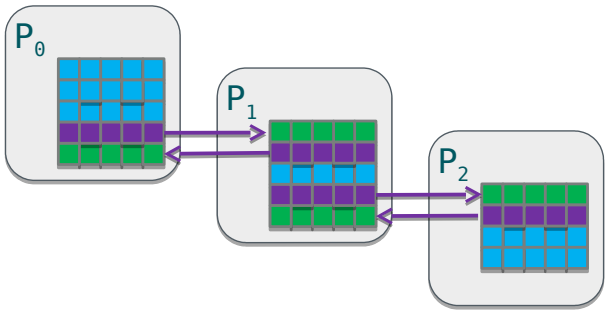
status
status for the completed operation

A call to `MPI_Test` is non-blocking. It allows one to schedule alternative activities while periodically checking for completion.

`MPI_Probe` is a similar kind of operation (see later slides).

Typical usage pattern

```
MPI_Irecv(ghost_data)
MPI_Isend(border_data)
compute(ghost_independent_data)
MPI_Waitall
compute(border_data)
```



Additional completion operations

Routine	Meaning
MPI_Waitany	Waits until any one operation has completed
MPI_Waitsome	Waits until at least one operation has completed
MPI_Test	Tests if an operation has completed (non-blocking)
MPI_Testall	Tests whether a list of operations have completed
MPI_Testany	Like Waitany but non-blocking
MPI_Testsome	Like Waitsome but non-blocking
MPI_Probe	Check for incoming messages without receiving them

Wait for non-blocking operations

```
MPI_Waitany(count, requests, index, status)
```

count
number of requests

requests
array of requests

index
index of request that completed

status
status for the completed operations

A call to `MPI_Waitany` returns when one operation identified by the array of requests is complete.

Wait for non-blocking operations

```
MPI_Waitsome(count, requests, done, index, status)
```

count
number of requests

requests
array of requests

done
number of completed requests

index
array of indexes of completed requests

status
array of statuses of completed requests

Returns when one or more operations is/are complete.

Message Probing

```
MPI_Iprobe(source, tag, comm, flag, status)
```

source
rank of sender (or `MPI_ANY_SOURCE`)

tag
message of the tag (or `MPI_ANY_TAG`)

comm
communicator

flag
true if there is a message that matches the pattern and can be received

status
status object

Allows incoming messages to be checked, without actually receiving them.

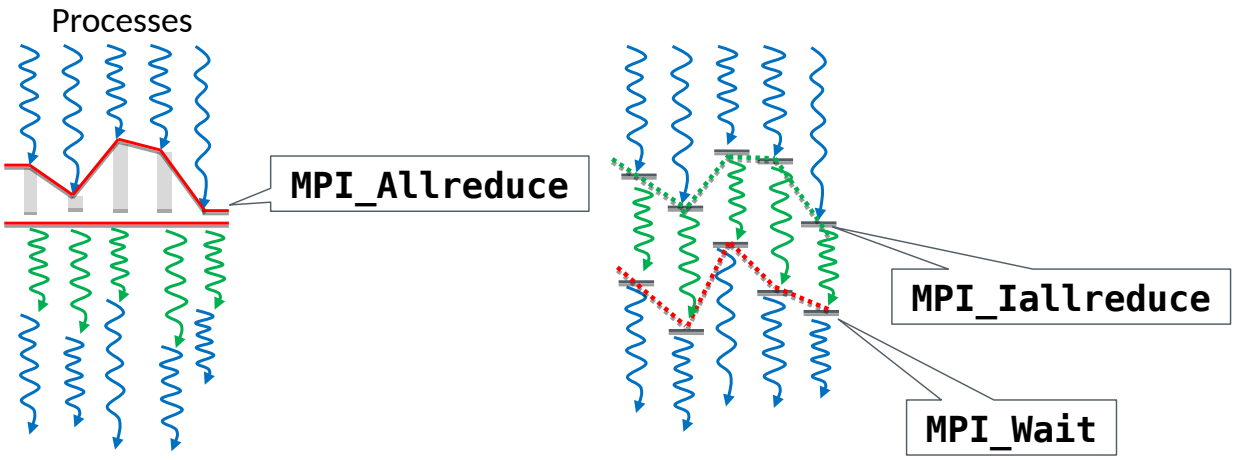
Non-blocking collectives

- Non-blocking collectives ("`I`-collectives") enable the overlapping of communication and computation together with the benefits of collective communication.
- Same syntax as for blocking collectives, besides
  - "`I`" at the front of the name (`MPI_Alltoall` -> `MPI_Ialltoall`)
  - Request parameter at the end of the list of arguments
  - Completion needs to be waited

Non-blocking collectives

- Restrictions
  - Have to be called in same order by all ranks in a communicator
  - Mixing of blocking and non-blocking collectives is not allowed

Non-blocking collectives



(Computation) work 1

(Computation) work 2, not involving data in the `Allreduce` operation

Example: Non-blocking broadcasting

```
MPI_Ibcast(buffer, count, datatype, root, comm, request)
```

buffer
data to be distributed

count
number of entries in buffer

datatype
data type of buffer

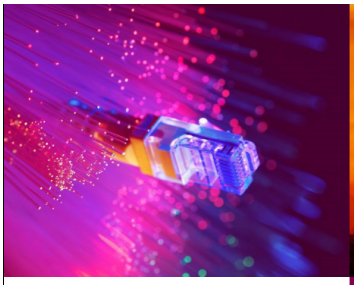
root
rank of broadcast root

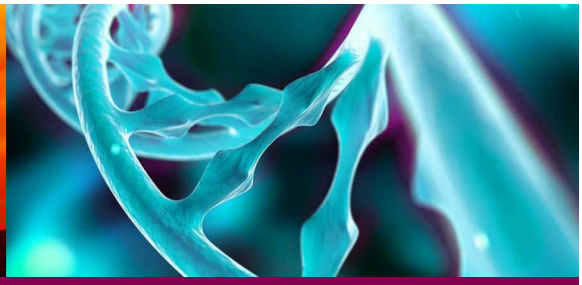
comm
communicator

request
a handle that is used when checking if the operation has finished

Summary

- Non-blocking communication is often useful way to do point-to-point communication in MPI.
- Non-blocking communication core features
  - Open receives with `MPI_Irecv`
  - Start sending with `MPI_Isend`
  - Possibly do something else while the communication takes place
  - Complete the communication with `MPI_Wait` or a variant
- MPI-3 contains also non-blocking collectives





MPI reference

CSC Training, 2021

CSC – Finnish expertise in ICT for research, education and public administration

C interfaces

C interfaces for the "first six" MPI operations

```c
int MPI_Init(int *argc, char **argv)

int MPI_Init_thread(int *argc, char **argv, int required, int *provided)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

int MPI_Barrier(MPI_Comm comm)

int MPI_Finalize()
```


C interfaces for the basic point-to-point operations

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

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

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)

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

MPI datatypes for C

MPI type	C type
MPI_CHAR	signed char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	

C interfaces for collective operations

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

int MPI_Scatter(void* sendbuf, int sendcount, MPI_datatype sendtype,
                void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_Comm comm)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_datatype sendtype,
                 void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_Comm comm)

int MPI_Gather(void* sendbuf, int sendcount, MPI_datatype sendtype,
               void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_Comm comm)

int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

C interfaces for collective operations

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

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

int MPI_Allgather(void* sendbuf, int sendcount, MPI_datatype sendtype,
                  void* recvbuf, int recvcount, MPI_datatype recvtype, MPI_Comm comm)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int* recvcounts, MPI_Datatype datatype,
                       MPI_Op op, MPI_Comm comm)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_datatype sendtype,
                 void* recvbuf, int recvcount, MPI_datatype recvtype, MPI_Comm comm)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype,
                  void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype,
                  MPI_Comm comm)
```


Available reduction operations

Operation	Meaning
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location

Operation	Meaning
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

C interfaces for user-defined communicators

```
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm newcomm)

int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int result)

int MPI_Comm_dup ( MPI_Comm comm, MPI_Comm newcomm )

int MPI_Comm_free ( MPI_Comm comm )
```


C interfaces for non-blocking operations

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request )

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request )

int MPI_Wait(MPI_Request *request, MPI_Status *status)

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

C interfaces for Cartesian process topologies

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims, int *dims, int *periods, int reorder,
                    MPI_Comm *comm_cart)

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdim, int *coords)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int rank)

int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int *low, int *high )
```


C interfaces for datatype routines

```
int MPI_Type_commit(MPI_Datatype *type)

int MPI_Type_free(MPI_Datatype *type)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_vector(int count, int block, int stride, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)

int MPI_Type_indexed(int count, int blocks[], int displs[], MPI_Datatype oldtype,
                     MPI_Datatype *newtype)

int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[],
                             int array_of_starts[], int order, MPI_Datatype oldtype,
                             MPI_Datatype *newtype )

int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

C interfaces for one-sided routines

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
                   MPI_Comm comm, MPI_Win *win)

int MPI_Win_fence(int assert, MPI_Win win)

int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
                  int target_rank, MPI_Aint target_disp, int target_count,
                  MPI_Datatype target_datatype, MPI_Win win)

int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)

int MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
                   int target_rank, MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```


C interfaces to MPI I/O routines

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

int MPI_File_close(MPI_File *fh)

int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)

int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
                     MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
                      MPI_Datatype datatype, MPI_Status *status)
```

C interfaces to MPI I/O routines

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
                      MPI_Datatype filetype, char *datarep, MPI_Info info)

int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
                      MPI_Status *status)

int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count,
                         MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
                       MPI_Status *status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count,
                          MPI_Datatype datatype, MPI_Status *status)
```


C interfaces for environmental inquiries

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

Fortran interfaces

Fortran interfaces for the "first six" MPI operations

```
mpi_init(ierror)
  integer :: ierror

mpi_init_thread(required, provided, ierror)
  integer :: required, provided, ierror

mpi_comm_size(comm, size, ierror)
mpi_comm_rank(comm, rank, ierror)
  type(mpi_comm) :: comm
  integer :: size, rank, ierror

mpi_barrier(comm, ierror)
  type(mpi_comm) :: comm
  integer :: ierror

mpi_finalize(ierror)
  integer :: ierror
```

Fortran interfaces for the basic point-to-point operations

```
mpi_send(buffer, count, datatype, dest, tag, comm, ierror)
  <type> :: buf(*)
  integer :: count, dest, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm

mpi_recv(buf, count, datatype, source, tag, comm, status, ierror)
  <type> :: buf(*)
  integer :: count, source, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm
  type(mpi_status) :: status
```


Fortran interfaces for the basic point-to-point operations

```
mpi_sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, &
             recvtype, source, recvtag, comm, status, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, dest, source, sendtag, recvtag, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
  type(mpi_status) :: status

mpi_get_count(status, datatype, count, ierror)
  integer :: count, ierror
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

MPI datatypes for Fortran

MPI type	Fortran type
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	real32
MPI_DOUBLE_PRECISION	real64
MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double complex
MPI_LOGICAL	logical
MPI_BYTE	

Fortran interfaces for collective operations

```
mpi_bcast(buffer, count, datatype, root, comm, ierror)
  <type> :: buffer(*)
  integer :: count, root, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm

mpi_scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount,  root, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, &
             root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), displs(*), recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
```

Fortran interfaces for collective operations

```
mpi_gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount,  root, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, &
            root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcounts(*), displs(*), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: count, root, ierror
  type(mpi_datatype) :: datatype
  type(mpi_op) :: op
  type(mpi_comm) :: comm
```


Fortran interfaces for collective operations

```
mpi_allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: count, ierror
  type(mpi_datatype) :: datatype
  type(mpi_op) :: op
  type(mpi_comm) :: comm

mpi_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: recvcounts(*), ierror
  type(mpi_datatype) :: datatype
  type(mpi_op) :: op
  type(mpi_comm) :: comm
```

Fortran interfaces for collective operations

```
mpi_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror)
  type:: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_alltoallv(sendbuf,sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, &
              comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), recvcounts(*), sdispls(*), rdispls(*), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
```


Available reduction operations

Operation	Meaning
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location

Operation	Meaning
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

Fortran interfaces for user-defined communicators

```
mpi_comm_split(comm, color, key, newcomm, ierror)
  integer :: color, key, ierror
  type(mpi_comm) :: comm, newcomm

mpi_comm_compare(comm1, comm2, result, ierror)
  integer :: result, ierror
  type(mpi_comm) :: comm1, comm2

mpi_comm_dup(comm, newcomm, ierror)
  integer :: ierror
  type(mpi_comm) :: comm, newcomm

mpi_comm_free(comm, ierror)
  integer :: ierror
  type(mpi_comm) :: comm
```


Fortran interfaces for non-blocking operations

```
mpi_isend(buf, count, datatype, dest, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, dest, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_irecv(buf, count, datatype, source, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, source, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_wait(request, status, ierror)
  integer :: ierror
  type(mpi_request) :: request
  type(mpi_status) :: status

mpi_waitall(count, array_of_requests, array_of_statuses, ierror)
  integer :: count, ierror
  type(mpi_request) :: array_of_requests(:)
  type(mpi_status) :: array_of_statuses(:)
```

Fortran interfaces for Cartesian process topologies

```
mpi_cart_create(old_comm, ndims, dims, periods, reorder, comm_cart, ierror)
  integer :: ndims, dims(:), ierror
  type(mpi_comm) :: old_comm, comm_cart
  logical :: reorder, periods(:)

mpi_cart_coords(comm, rank, maxdim, coords, ierror)
  integer :: rank, maxdim, coords(:), ierror
  type(mpi_comm) :: comm

mpi_cart_rank(comm, coords, rank, ierror)
  integer :: coords(:), rank, ierror
  type(mpi_comm) :: comm

mpi_cart_shift(comm, direction, displ, low, high, ierror)
  integer :: direction, displ, low, high, ierror
  type(mpi_comm) :: comm
```


Fortran interfaces for datatype routines

```
mpi_type_commit(type, ierror)
  type(mpi_datatype) :: type
  integer :: ierror

mpi_type_free(type, ierror)
  type(mpi_datatype) :: type
  integer :: ierror

mpi_type_contiguous(count, oldtype, newtype, ierror)
  integer :: count, ierror
  type(mpi_datatype) :: oldtype, newtype

mpi_type_vector(count, block, stride, oldtype, newtype, ierror)
  integer :: count, block, stride, ierror
  type(mpi_datatype) :: oldtype, newtype
```

Fortran interfaces for datatype routines

```
mpi_type_indexed(count, blocks, displs, oldtype, newtype, ierror)
  integer :: count, ierror
  integer, dimension(count) :: blocks, displs
  type(mpi_datatype) :: oldtype, newtype

mpi_type_create_subarray(ndims, sizes, subsizes, starts, order, oldtype, newtype, ierror)
  integer :: ndims, order, ierror
  integer, dimension(ndims) :: sizes, subsizes, starts
  type(mpi_datatype) :: oldtype, newtype

mpi_type_create_struct(count, blocklengths, displacements, types, newtype, ierror)
  integer :: count, blocklengths(count), ierror
  type(mpi_datatype) :: types(count), newtype
  integer(kind=mpi_address_kind) :: displacements(count)
```


Fortran interfaces for one-sided routines

```
mpi_win_create(base, size, disp_unit, info, comm, win, ierror)
  <type> :: base(*)
  integer(kind=mpi_address_kind) :: size
  integer :: disp_unit, ierror
  type(mpi_info) :: info
  type(mpi_comm) :: comm
  type(mpi_win)  :: win

mpi_win_fence(assert, win, ierror)
  integer :: assert, ierror
  type(mpi_win)  :: win

mpi_put(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, &
        target_datatype, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_win)  :: win
```

Fortran interfaces for one-sided routines

```
mpi_get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count,
        target_datatype, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_win)  :: win

mpi_accumulate(origin_addr, origin_count, origin_datatype, target_rank, target_disp, &
               target_count, target_datatype, op, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_op) :: op
  type(mpi_win)  :: win
```


Fortran interfaces for MPI I/O routines

```
mpi_file_open(comm, filename, amode, info, fh, ierror)
  integer :: amode, ierror
  character* :: filename
  type(mpi_info) :: info
  type(mpi_file) :: fh
  type(mpi_comm) :: comm

mpi_file_close(fh, ierror)
  integer :: ierror
  type(mpi_file) :: fh

mpi_file_seek(fh, offset, whence, ierror)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: whence, ierror
  type(mpi_file) :: fh
```

Fortran interfaces for MPI I/O routines

```
mpi_file_read(fh, buf, count, datatype, status, ierror)
mpi_file_write(fh, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status

mpi_file_read_at(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at(fh, offset, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```


Fortran interfaces for MPI I/O routines

```
mpi_file_set_view(fh, disp, etype, filetype, datarep, info, ierror)
  integer :: ierror
  integer(kind=MPI_OFFSET_KIND) :: disp
  type(mpi_info) :: info
  character* :: datarep
  type(mpi_file) :: fh
  type(mpi_datatype) :: etype, datatype

mpi_file_read_all(fh, buf, count, datatype, status, ierror)
mpi_file_write_all(fh, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

Fortran interfaces for MPI I/O routines

```
mpi_file_read_at_all(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at_all(fh, offset, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```


Fortra interfaces for environmental inquiries

```
mpi_get_processor_name(name, resultlen, ierror)
  character(len=MPI_MAX_PROCESSOR_NAME) :: name
  integer :: resultlen, ierror
```