MPI parallel programming technology

Laboratory of Information Technologies

Joint Institute for Nuclear Research

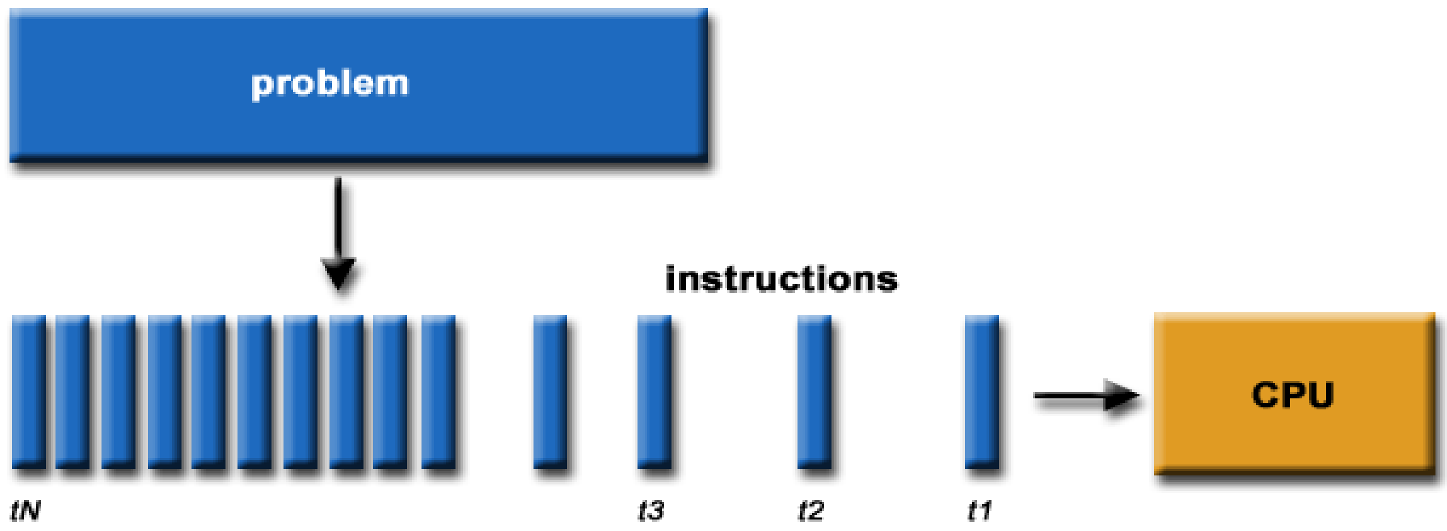
All computers are multiprocessors now

- Look at the Top500 - known list of the most powerful computational system in the world. You'll see - there are no machine with single processor there. All they are clusters of hundreds and thousands processors.
- Even personal computers have now several (2-8) independent processors (kernels).
- The typical program, Fortran- or C-written, uses single processor and cannot use several processors simultaneously! If we want compute faster, we will have to parallelize our program!
- There are several technologies for parallelization.
  MPI is one of them.

Serial computing

Traditionally, software has been written for ***serial*** computation:

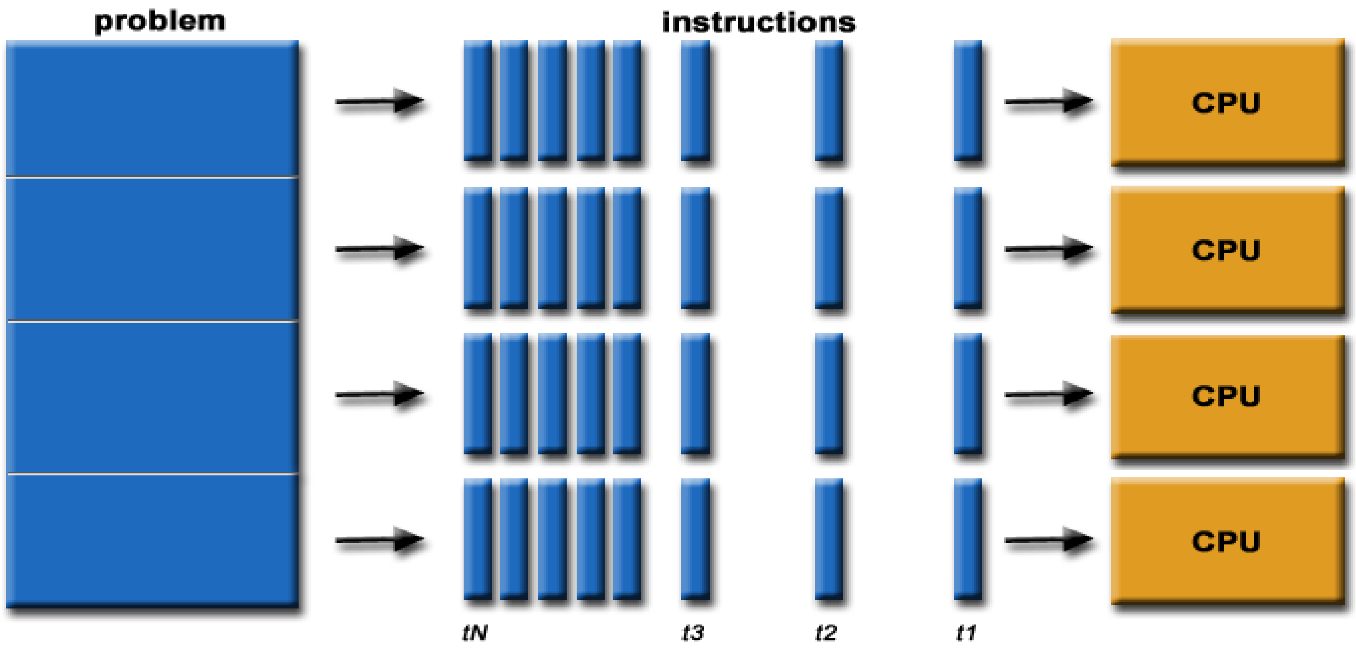
- To be run on a single computer having a single **C**entral **P**rocessing **U**nit (**CPU**);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.



Parallel computing

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

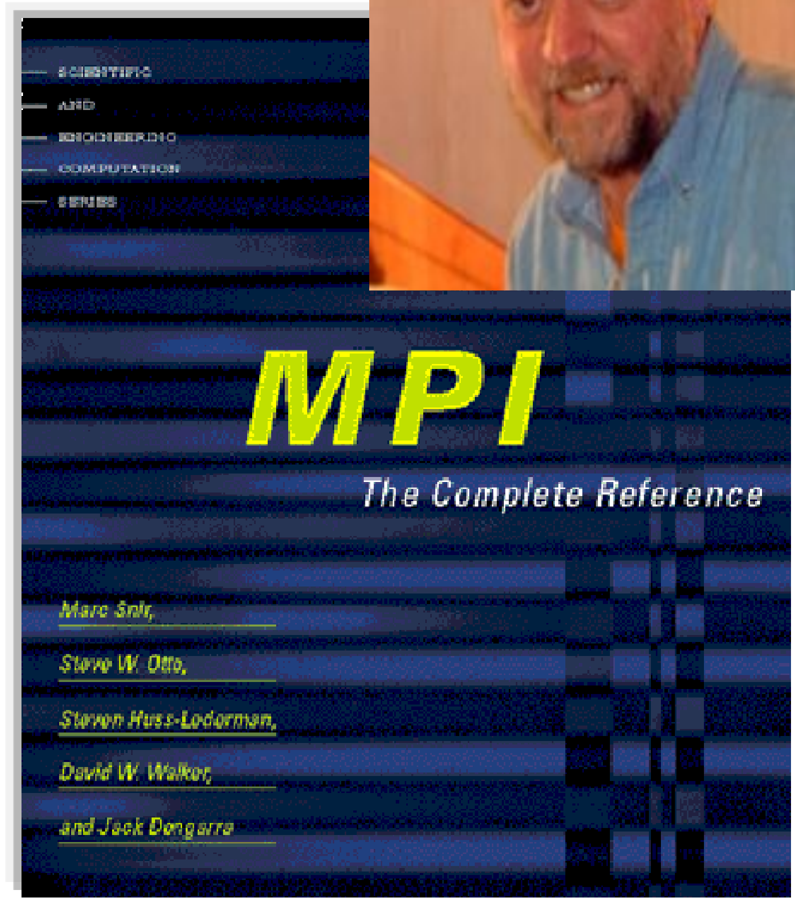
- To be run using multiple CPUs.
- A problem is broken into discrete parts that can be solved concurrently.
- Each part is further broken down to a series of instructions.
- Instructions from each part execute simultaneously on different CPU.



Message Passing Interface (1992)

www.netlib.org/utk/people/JackDongarra/

- Fundamentals
- SPMD-model of programming
- MPI vocabulary
- The basic operations. "Point-to-point" exchanges
- Collective operations
- Simple program examples



SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

MPI

The Complete Reference

Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra

MPI – what is it ?

The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for writing message passing programs.

MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

It is a great set communicative and auxiliary operations for programming with usage Fortran and C languages, arranged as a library.

It works practically in any computational systems, even heterogeneous.

Its 2 fundamentals are: process and message.

Process - the program together with its own data, being executed by processor.

Processes communicate exclusively through messages.

Single Program - Multiple Data (SPMD) Programming Model

- The behavior of all processes is described by the same program. Processes may use different data.
- The group of NP processes is created for task execution.
- Needed interprocess communications in the group are programmed with usage of MPI-library, which dictates the standard for programming.
- Group is identified by integer-type descriptor (communicator).
- Inside the group processes are numerated from 0 to NP-1. Each process knows its own number myProc and total number of processes NP.
- Quasi-simultaneous launch of all NP processes does the OS :

```
mpirun –np  <NP>   <executable file>
```

  NP is given by user but not by the number of processors available!

Modification of sequential code to make it effective for parallelization

Example: **Summation of n numbers.** Sequential code.

```
s = 0
Do i = 1, n
  s = s + a(i)
EndDo
```

Modification: summation process is splitted; we obtain two independent branches that can be parallelized.

```
s = 0
s1 = 0
n2 = n/2
        Do i = 1, n2
          s = s + a(i)
        EndDo
  Do i = n2+1, n
    s1 = s1 + a(i)
  EndDo
s = s+s1
```

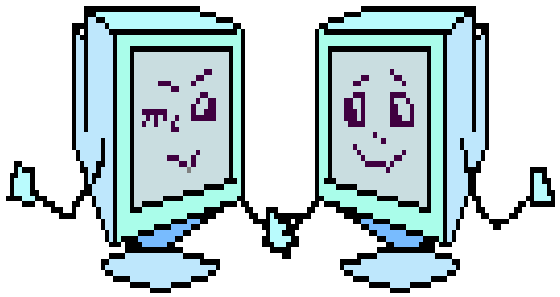
This idea can be generalized for more processors.

All NP processes execute the same program asynchronous.

Each process knows its own number myProc and total number of processes NP.

Parallel program can consist of such fragments:

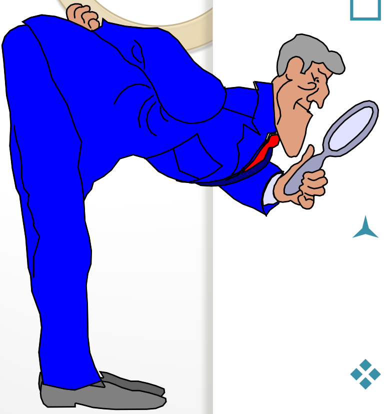
```
If (myProc.eq.1) then
        < work 1 >
else if (myProc.eq.2) then
        < work 2 >
        .     .     .
else
        <work N >
endif
```



MPI vocabulary

- Fortran - mpif.h , C - mpi.h - required for all programs that make MPI library calls.
- MPI_Comm_World – predefined group of all processes being running.
- MPI_Integer, MPI_Real, MPI_Byte …- data types
- MPI_Any_Source, MPI_Any_Tag - jokers (somewhat like *.* in file-systems).
- MPI_Source, MPI_Tag, MPI_Error - separate fields of status for message being received.
- …………..

The basic MPI operations



- MPI_Init(ierr), MPI_Finalize(ierr) – «brackets» of parallel part of the code. All MPI procedures can be called between these operators.
- MPI_Abort(comm) - finish for all processes in group while an error occurs.

- MPI_Comm_Size(comm,NProc,ierr) – how many processes are in the group ?
- MPI_Comm_Rank(comm,myProc,ierr) – what is my number in the group ?

- C-format:

  rc = MPI_Xxxxx(parameter, ...)
- Fortran-format:

  CALL MPI_XXXXX(parameter,..., ierr)
  call mpi_xxxxx(parameter,..., ierr)

C example:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[ ])
{
MPI_Init(&argc, &argv);
        printf ("Hello\n" );                // Parallel section
MPI_Finalize( );
 return 0;
}
```

Word “**Hello**” is printed by each processor of the group. In case we have 5 processes in the group we have to obtain “**Hello**” to be displayed 5 times.

Fortran example:

```
program example
include ’mpif.h’
integer ierr
call MPI_INIT(ierr)
        print *,’Parallel section’        ! Parallel section
call MPI_FINALIZE(ierr)
end
```

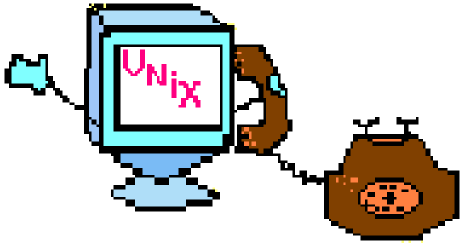
“**Parallel section**” is printed by each process of the group. In case of 5 processes we should obtain “**Parallel section**” to be displayed 5 times.

Fortran example:

```
      Program Hello
      Include 'mpif.h‘
      call MPI_Init(ierr)
      call MPI_Comm_Size(MPI_Comm_World, NProc)
      call MPI_Comm_Rank(MPI_Comm_World, myProc)
      write(*,*) ' Hello, I am ',myProc,
&             ' process of ',Nproc
&             ' in a group ',MPI_COMM_WORLD
      call MPI_Finalize(ierr)
      End
```

Outcome for 3 processes:

```
Hello, I am 0 process of 3 in a group 91
Hello, I am 2 process of 3 in a group 91
Hello, I am 1 process of 3 in a group 91
```



More Fortran Example. “IF” in parallel computing

```
        program example
        include ’mpif.h’
        integer ierr, size, rank
c
c  start parallel section
        call MPI_INIT(ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
        print *,’my rank is’, rank
        if (rank.eq.0)  print *,’size of our group is’, size
        call MPI_FINALIZE(ierr)
c  finish parallel section
        end
```

Each process prints its rank; zero rank process prints the group size:

```
        ’my rank is’, rank
        ’my rank is’, rank
        ’my rank is’, rank
        ’size of our group is’, size
```

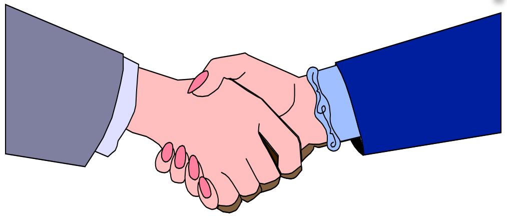
Example in C

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[ ])
{
            int size;                          // quantity of processes in the group
            int num;                           // number of process
MPI_Init(&argc, &argv);
            MPI_Comm_size(MPI_COMM_WORLD, &size);
            MPI_Comm_rank(MPI_COMM_WORLD ,&num);
            printf ("My Rank is %d\n",num );
{           if (num == 0)       printf (“Size %d\n",size );     }
MPI_Finalize( );
 return 0;
}
```

Outcome for 5 processes:

```
            My Rank is 0
            My Rank is 1
            My Rank is 2
            My Rank is 3
            My Rank is 4
            Size 5
```

“Point-to-Point” MPI operations



- MPI_Send(Buf,Cnt,Type,Whom,Tag,comm,ierr)
- MPI_Recv(Buf,Cnt,Type,From,Tag,comm,status,ierr)

Current process myProc sends to process Whom

(receives from process From) Cnt items of data of type Type

into buffer Buf with tag Tag.

Instead of From and Tag may be jokers:

MPI_Any_Source, MPI_Any_Tag.

!!! When the process From is sending the data BUF via MPI_Send subroutine to the Whom process – the Whom process should call the corresponding MPI_Recv procedure of receiving the data BUF from the process From.

Program exampleA

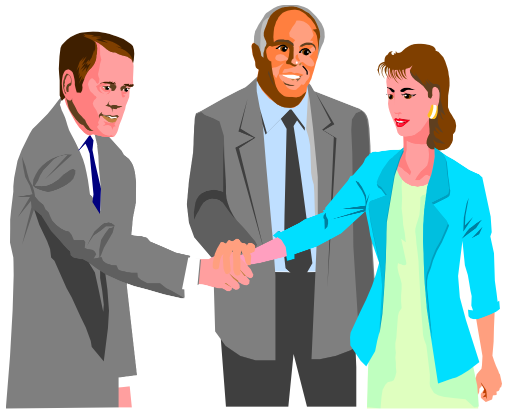
```
C    0-process sends value a to another processes;  prints rank and value a
C    Another processes receive b from the 0-process;  print rank and value b
     include 'mpif.h'
     integer ierr, size, rank, size_minus_1
     real a,b
     integer status(MPI_STATUS_SIZE)
     call MPI_INIT(ierr)
     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
     if(rank .eq. 0) then
        a = 10.0
        size_minus_1 = size-1
        do  i=1,size_minus_1
           call MPI_SEND(a, 1, MPI_REAL, i, 5, MPI_COMM_WORLD, ierr)
        enddo
        print *, 'process ', rank,' a = ', a
     else
        call MPI_RECV(b, 1, MPI_REAL, 0, 5, MPI_COMM_WORLD, status, ierr)
        print *, 'process ', rank,' b = ', b
     end if
     call MPI_FINALIZE(ierr)
     end
```

“MPI MINIMUM”

MPI_Init
MPI_Finalize
MPI_Comm_size
MPI_Comm_rank
+
MPI_Send
MPI_Recv

- These six MPI-procedures are enough to practically arrange parallelism.
- All another procedures optimize exchange and simplify the code.

The collective MPI-operations



MPI_Barrier(comm,ierr)

MPI_Bcast(Buf,Cnt,Type,root,comm,ierr)

MPI_Gather(Sbuf,Scnt,Styp,Rbuf,Rcnt,Rtyp,root,comm,ierr)

MPI_Scatter(Sbuf,Scnt,Styp,Rbuf,Rcnt,Rtyp,root,comm,ierr)

MPI_Reduce(Sbuf,Rbuf,Cnt,Type,Op,root,comm,ierr)

MPI_AllReduce(Sbuf,Rbuf,Cnt,Type,Op,comm,ierr)

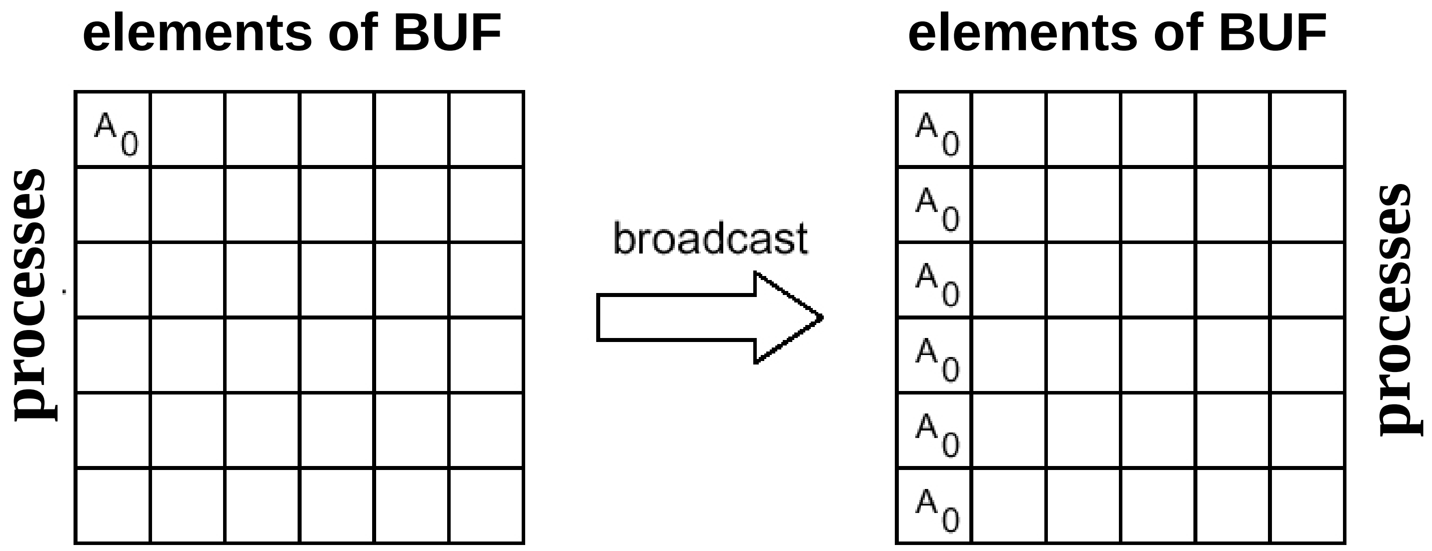
Here root-process initiates the operation, and the others works according the operation semantics.

Op = (MPI_Min, MPI_Max, MPI_Sum, MPI_Prod ...)

MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERR)

- **This procedure should be called by all processes in the group.**
- **COUNT elements of BUF are sent by the ROOT process to all processes in the group COMM.**
- **Result: All processes in the group COMM have the BUF of the COUNT elements.**

Vertical: processes;
Horizontal: elements of BUF.



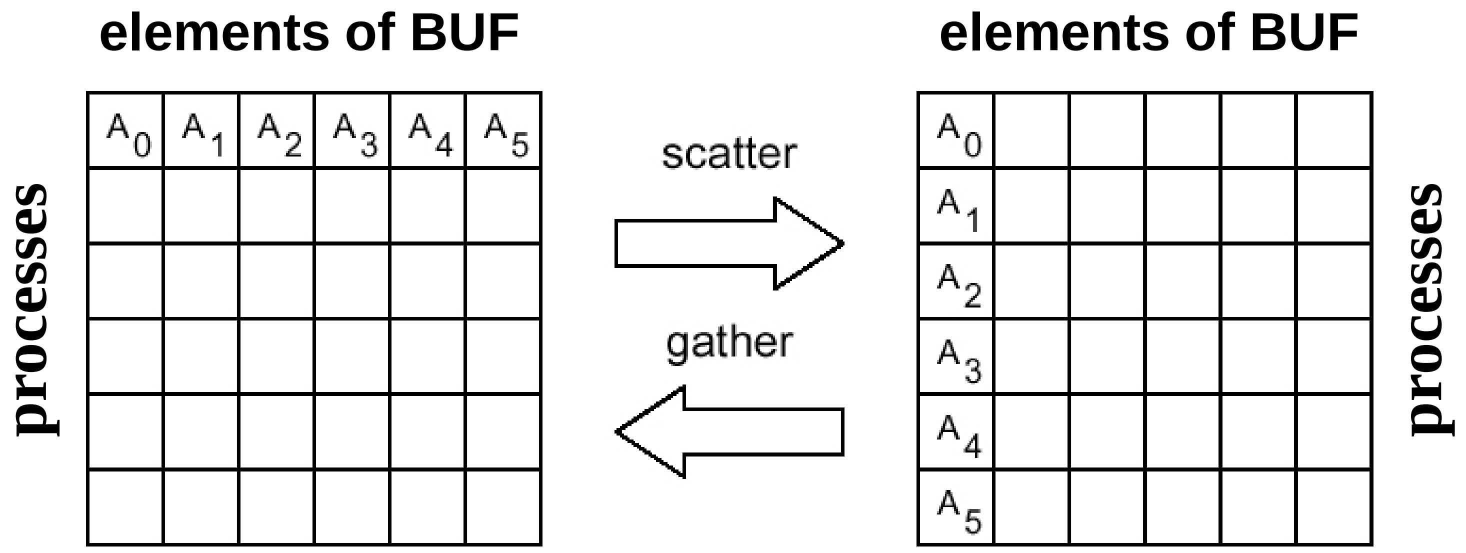
MPI_SCATTER(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, ROOT, COMM, IERR)

- This procedure should be called by all processes in the group.
- SBUF is split on equal portions of the length SCOUNT.
- These portions of SCOUNT elements of SBUF are sent from the ROOT process to the RBUF of the RCOUNT length to all processes in the group COMM

Order is determined the process ranks.

Result: Each process in the group have own portion of SBUF of the SCOUNT elements.

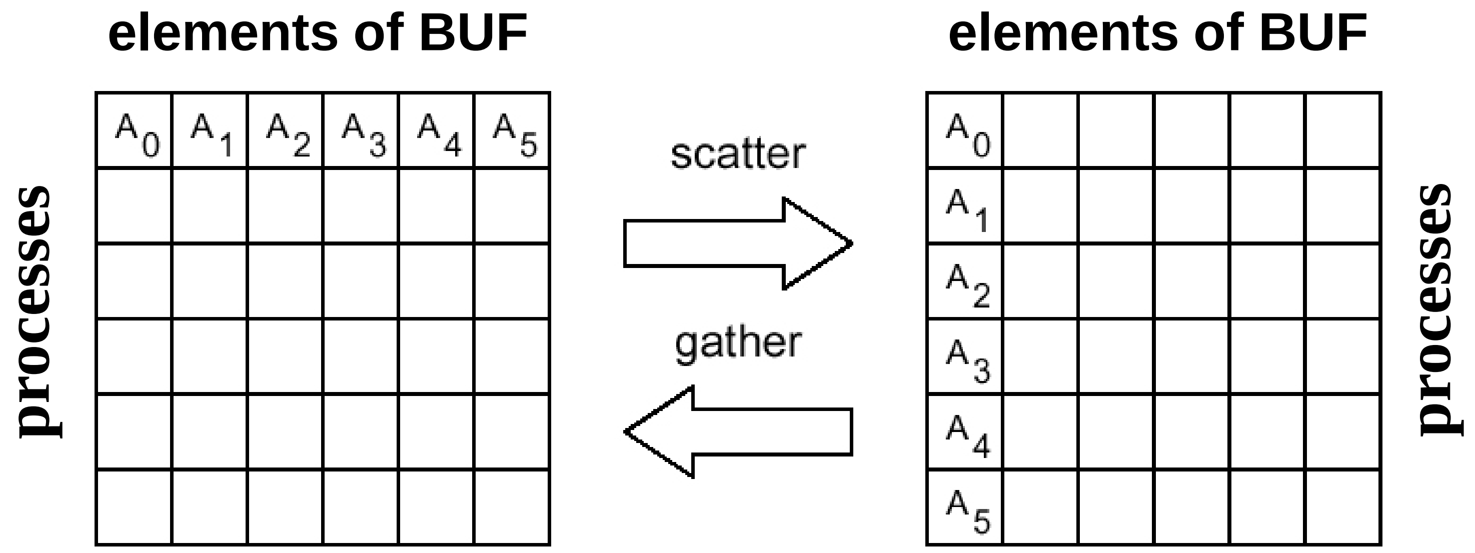
Vertical: processes;
Horizontal: elements of BUF.



MPI_GATHER(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, ROOT, COMM, IERR)

- **This procedure should be called by all processes in the group.**
- **Each process of group COMM sends SCOUNT elements of its SBUF to the ROOT process, to the RBUF of the RCOUNT length.**
- **Result: In the ROOT process, the RBUF contains SIZE portions of SCOUNT elements sent by each process.**
- **Portions are placed in RBUF in accordance to the RANK of processes.**

Vertical: processes;
Horizontal: elements of BUF.



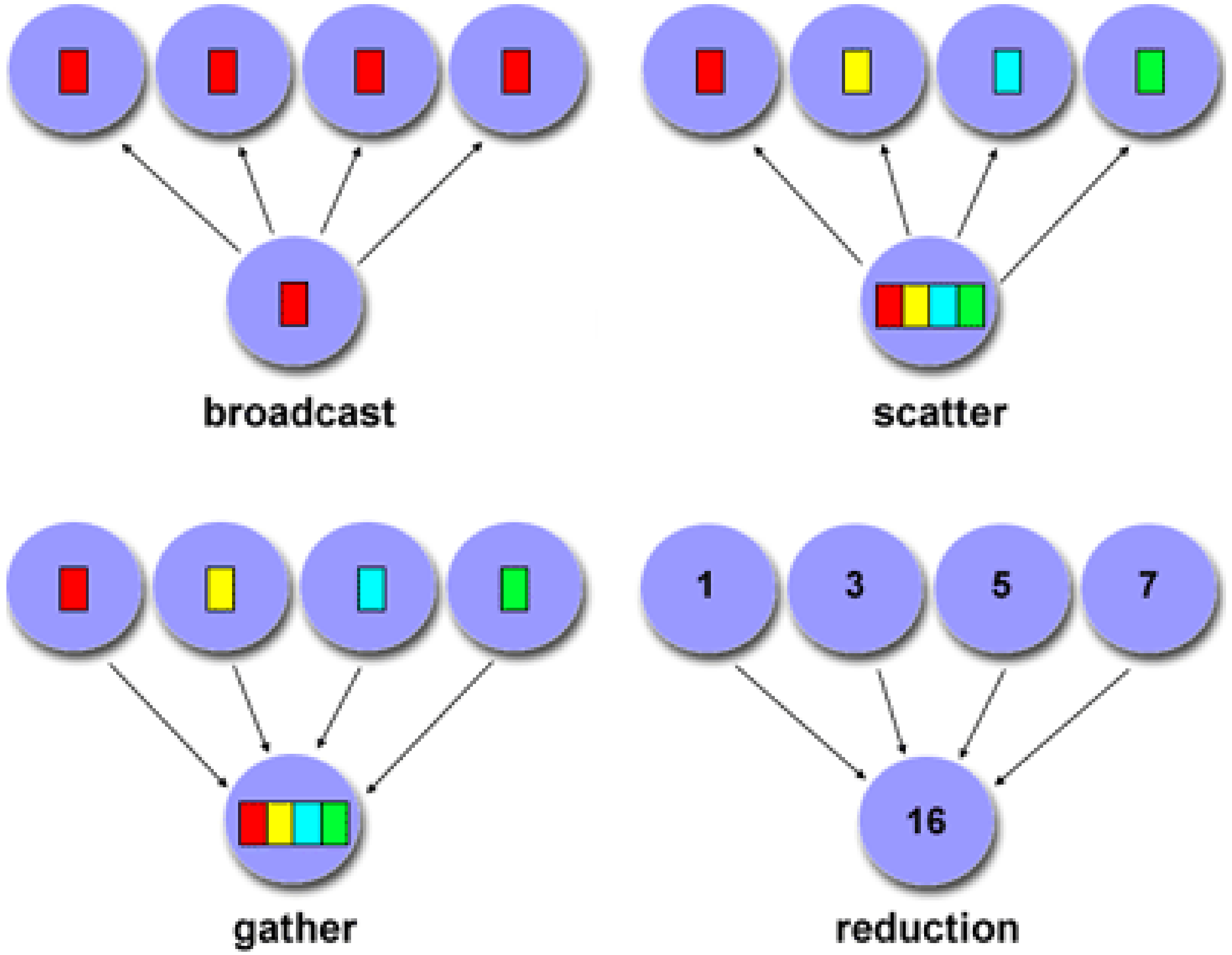
MPI_REDUCE(SBUF, RBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERR)

- **This procedure should be called by all processes in the group.**
- **The global operation OP is performed under COUNT elements of the RBUF of all processes in the group COMM.**
- **Result of operation OP is placed to the RBUF of the ROOT process.**

Examples of global operations:
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
MPI_MINLOC, MPI_MAXLOC
MPI_LAND, MPI_LOR

MPI_ALLREDUCE(SBUF, RBUF, COUNT, DATATYPE, OP, COMM, IERR)

Result of operation OP is placed to the RBUF to all processes in the group COMM.


broadcast
scatter
gather
1
3
5
7
16
reduction

Program exampleA (REPEAT from previous presentation)

```
C     0-process sends value a to another processes;  prints rank and value a
C     Another processes receive b from the 0-process;  print rank and value b
      include 'mpif.h'
      integer ierr, size, rank, size_minus_1
      real a,b
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      if(rank .eq. 0) then
         a = 10.0
         size_minus_1 = size-1
         do  i=1,size_minus_1
            call MPI_SEND(a, 1, MPI_REAL, i, 5, MPI_COMM_WORLD, ierr)
         enddo
         print *, 'process ', rank,' a = ', a
      else
         call MPI_RECV(b, 1, MPI_REAL, 0, 5, MPI_COMM_WORLD, status, ierr)
         print *, 'process ', rank,' b = ', b
      end if
      call MPI_FINALIZE(ierr)
      end
```

Program exampleB (Modification of exampleA)

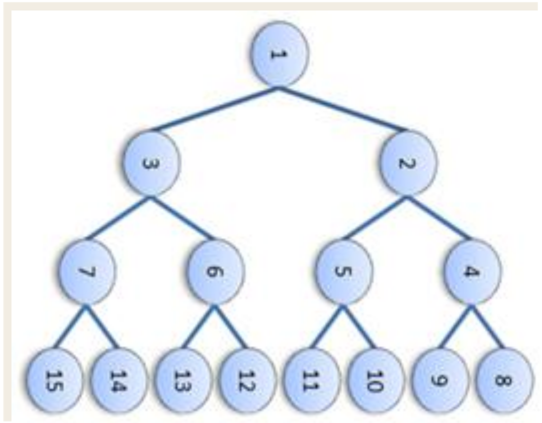
```
C    0-process sends value a to all processes; prints rank and value a
C    All other processes receive a from the 0-process; print rank and value a
        include 'mpif.h'
        integer ierr, size, rank
        real a,b
        integer status(MPI_STATUS_SIZE)
        call MPI_INIT(ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

        if (rank .eq. 0)      a = 10.0

        MPI_BCAST(a, 1, MPI_REAL, 0, MPI_COMM_WORLD, IERR)

        print *, 'process ', rank,' a = ', a

        call MPI_FINALIZE(ierr)
        end
```

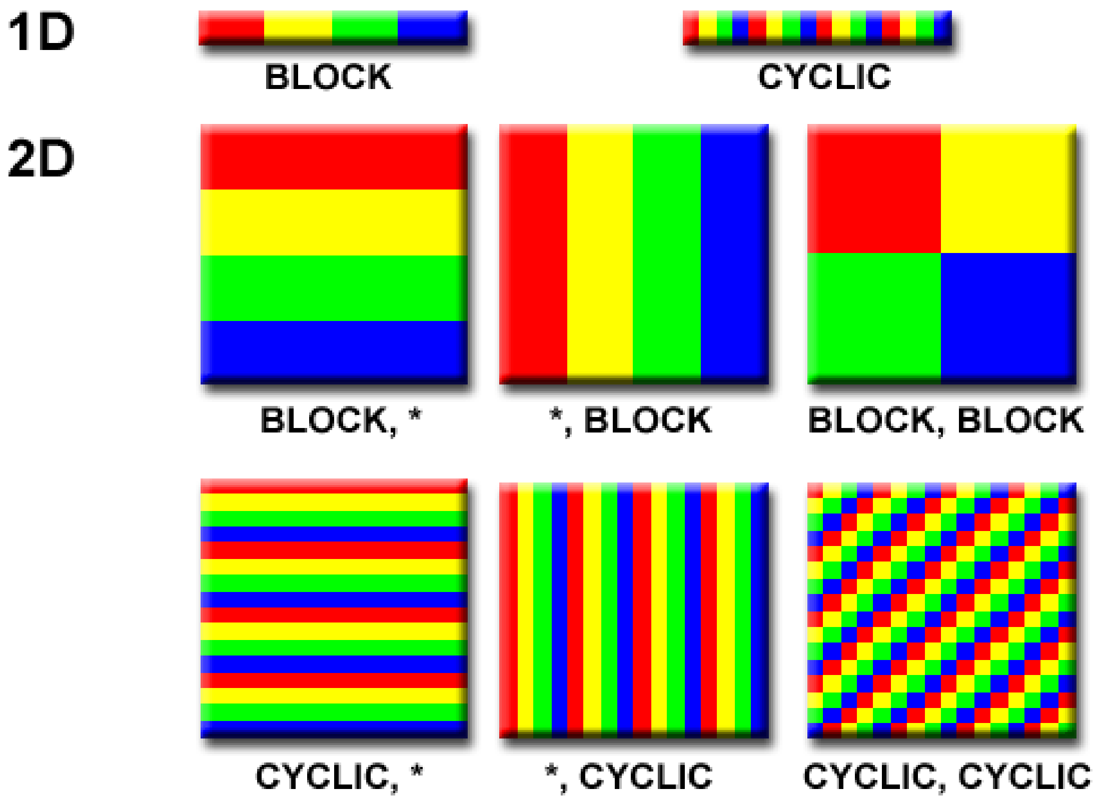


Example : collective integrating for function F(x) of I variable on (A,B) interval

```
Include 'mpif.h'
external F                                            ! Function being integrated
data A/0/,  B/1/                                      ! Limits of integrating
call MPI_Init(ierr)
call MPI_Comm_Size(MPI_Comm_World, NProc,ierr)        ! How many of us?
call MPI_Comm_Rank(MPI_Comm_World, myProc,ierr)       ! Who am I ?
dx=(B-A)/Nproc                                        ! Divide the interval equally
a1=A+myProc*dx                                        ! Between all processes
b1=a1+dx
s1=Common_Integration( F, a1, b1 )                    ! – the general librarian program
call MPI_AllReduce(s1,S,1,MPI_Real, MPI_Sum, MPI_Comm_World,ierr)
call MPI_Finalize(ierr)
End
```

There are different ways to partition data

One of the first steps in designing a parallel program is to break the problem into discrete parts of work that can be distributed to multiple tasks.



The way for distribution N columns of matrix between NProc processes. It's a way N1 (divide onto contiguous pieces - "block")

```
nc=N/NProc                    ! columns distribution for processes (we assume N>NProc)
nrest=mod(N,NProc)            ! =0 if N has been entirely divided onto NProc
if(myProc.lt.nrest) then
   nc1=1+(nc+1)*myProc        ! almost equally
   nc2=nc1+nc                 ! 1-st and last columns for myProc
else
   nc1=1+(nc+1)*nrest+nc*(myProc-nrest)
   nc2=nc1+nc-1               ! Some processes  got  a bit less jobs!
endif
if(nc1.le.nc2) then
    write(*,*) ' Process',myProc, '  started for columns from',nc1,' to',nc2
 else
    write(*,*) ' Process',myProc,'  does nothing!'
endif
```

It looks complicate, but it works, and the results will be convenient for sending to master

The way N2: cyclic

Let NProc processes do N units of collective job, working together. Then instead of the complete loop

```
do job=1,N
```

each process performs a reduced loop:

```
do job=1+myProc,N,NProc
     .....  One unit of job is performed ...
enddo
```

It is clear and convenient: we need not take care in direct division N by NProc ! The difficulties while sending results to master are possible! It is the simplest way to divide the whole job between several workers.

The Amdahl's law Gene Amdahl, 1967

Describes a limit of achievement progress while program parallelization.

Let **0 <= S <= 1** – the part of computational operations in our program, which must be performed strongly sequentially. Than, trying to use **P** processes simultaneously instead of one, we can reach an acceleration **A** not more than in

$$A = \frac{1}{S + \frac{1-S}{P}}$$

Directing **P** at the infinity, we have a limit: **A<1/S**.

Particularly, if **S>0.1**, then **A < 10** during any **P**.

But if **S=0** (that is an absolutely unreal case in reality), then **A** = **P**.

So, decide by yourself, corresponds your efforts while parallelization to future profit, or not! Efforts expected to be rather big .

So – even if we take a lot of processors we cannot infinitely accelerate execution of our code because **S** cannot be zero in actuality.

Beside, we should also account the time of exchange between processors.

Run MPI-program

- Add a module to environment:

  module add openmpi/1.6.5

- Compilation (f77 -> mpif77, cc -> mpicc) :

  mpif77 example.f   **mpicc example.c**

- Run 3 processes (interactive mode):

  mpirun -n 3 a.out

- Launch batch jobs:

  sbatch <script_mpi >

- script_mpi:

```
#!/bin/sh
#SBATCH -n 5
mpiexec ./a.out
```

References:

- http://www.mpi-forum.org/ - Message Passing Interface Forum.
- http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

  MPI: The complete Reference. MIT Press, Cambridge, Massachusetts, 1997.

  Authors: Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, Jack Dongarra.
- http://www.open-mpi.org/ - The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners.
- http://www.parallel.ru , http://parallel.ru/docs/mpich/html/
- https://computing.llnl.gov/tutorials/parallel_comp/ - Introduction to Parallel Computing.
- https://computing.llnl.gov/tutorials/mpi/ - Message Passing Interface (MPI).
- http://www.lam-mpi.org/tutorials/bindings/ - C, C++, and Fortran bindings for MPI-1.2