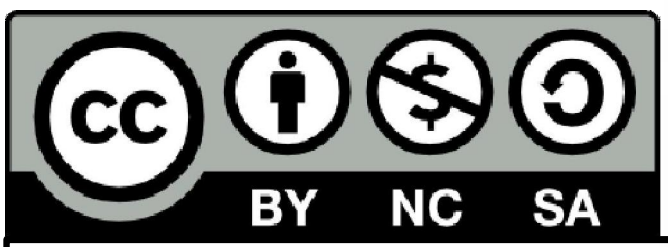
Pekka Manninen

Parallel Programming with MPI

March 25-27, 2013
CSC – IT Center for Science Ltd, Finland

```
!***  subroutine mpi_utils_step_parallel_edge
      implicit none
      integer ele, ierr
          do ele = 1, nfaces
          call mpi_isend(commvec(ele)%field1(commvec(ele)%out_i1,  &
              &                              commvec(ele)%out_j1,  &
              &                              commvec(ele)%out_k1), &
              & 1, commvec(ele)%mpi_type_out1, &
              & commvec(ele)%to_id, commvec(ele)%to_id, &
              & MPI_COMM_WORLD, send_reqs(ele), ierr)
         if(ierr /= MPI_SUCCESS) then
             call pio_abort(ierr)
         end if
         call mpi_isend(commvec(ele)%field2(commvec(ele)%out_i2,  &
              &                             commvec(ele)%out_j2,  &
              &                             commvec(ele)%out_k2), &
              & 1, commvec(ele)%mpi_type_out2, &
              & commvec(ele)%to_id, commvec(ele)%to_id+tag_offset, &
              & MPI_COMM_WORLD, send_reqs(nfaces+ele), ierr)
         if(ierr /= MPI_SUCCESS) then
             call pio_abort(ierr)
         end if
      end do
#ifdef NONBLOCK
      do ele = 1, nfaces
         call mpi_irecv(commvec(ele)%field1(commvec(ele)%in_i1,  &
              &                             commvec(ele)%in_j1,  &
              &                             commvec(ele)%in_k1), &
              & 1, commvec(ele)%mpi_type_in1, &
```



Agenda

Monday

9.00-9.45	Getting started with MPI
10.00-10.30	Point-to-point communication
10.30-10.45	Coffee break
10.45-11.45	Exercises
11.45-12.15	Collective communication
12.15-13.00	Lunch break
13.00-13:30	Collective communication cont'd
13:30-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.15	More about point-to-point communication
15:30-	Exercises

Tuesday

9.00-9.45	Exercises
9.45-10.30	Non-blocking communication
10.30-10.45	Coffee break
10.45-12.15	Exercises
12.15-13.00	Lunch break
13:00-13:30	Defining own communicators
13:30-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.15	Communication topologies
15:15-	Exercises

Wednesday

9.00-9.45	User-defined datatypes
10.00-10.30	Exercises
10.30-10.45	Coffee break
10.45-12.15	Exercises
12.15-13.00	Lunch break
13.00-13:45	MPI I/O
13:45-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.15	MPI I/O cont'd
15:15-15:45	Exercises
15:45-16:15	Performance considerations, course wrap-up

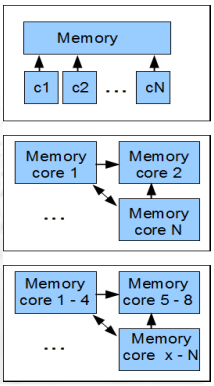
Web resources

- List of MPI functions with detailed descriptions http://mpi.deino.net/mpi_functions/index.htm
- Good online MPI tutorial: https://computing.llnl.gov/tutorials/mpi
- Lots of MPI lecture recordings (slides & videos): http://www.prace-ri.eu/training
- MPI standard http://www.mpi-forum.org/docs/
- MPI Implementations:
  - MPICH2 http://www.mcs.anl.gov/research/projects/mpich2/
  - OpenMPI http://www.open-mpi.org/

Getting started with MPI

Types of parallel computers

- Shared memory
  - all the cores can access the whole memory
- Distributed memory
  - all the cores have their own memory
  - communication is needed in order to access the memory of other cores
- Current supercomputers combine the distributed memory and shared memory approaches



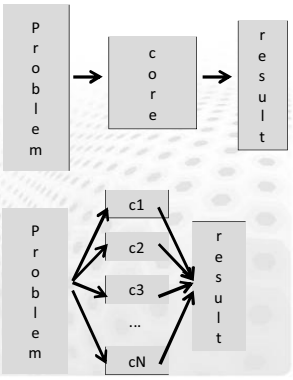
PART I: INTRODUCTION TO PARALLEL COMPUTING

Parallel programming models

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

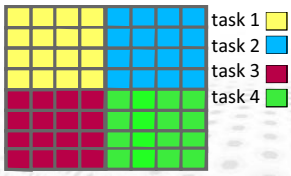
What is parallel computing?

- Serial computing
  - single processing unit (core) is used for solving a problem
  - single task performed at once
- Parallel computing
  - multiple cores are used for solving a problem
  - problem is split into smaller subtasks
  - multiple subtasks are performed *simultaneously*



Exposing parallelism

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



Why parallel computing?

- Solve problems faster
  - parallel programming is required for utilizing multiple cores
- Solve bigger problems
  - parallel computing may allow application to use more memory
  - apply old models to new length and time scales
  - grand challenges
- Solve problems better
  - more precise models

PART II: FIRST ENCOUNTER WITH MPI

Message-Passing Interface

- MPI application programming interface (API) is the most widely used approach for distributed parallel computing
- MPI programming is based on library routines
- MPI programs are portable and scalable
- MPI is flexible and comprehensive
  - large (over 120 procedures)
  - concise (often only 6 procedures are needed)
- MPI standardization by MPI Forum

MPI communicator

- Communicator is an object connecting a group of processes
- Initially, there is always a communicator MPI_COMM_WORLD which contains all the processes
- Most MPI functions require communicator as an argument
  - i.e., in which ”context” the required communication happens
- Users can define own communicators

Execution model

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

Routines of the MPI library

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

Execution model

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

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

Programming MPI

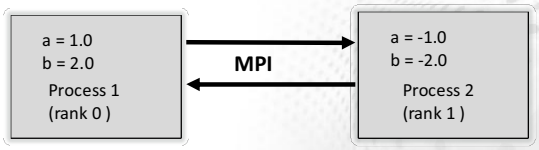
- MPI standard defines interfaces to C and Fortran programming languages
  - There are unofficial bindings to Python, Perl and Java
- C call convention

  `rc = MPI_Xxxx(parameter,...)`
  - some arguments have to passed as pointers
- Fortran call convention

  `CALL MPI_XXXX(parameter,..., rc)`
  - return code in the last argument

Data model

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



First five MPI commands

- Set up the MPI environment

  `MPI_Init()`
- Information about the communicator

  `MPI_Comm_size(comm, size)`

  `MPI_Comm_rank(comm, rank)`
  - Parameters

    comm communicator

    size number of processes in the communicator

    rank rank of this process

First five MPI commands

- Synchronize processes

  `MPI_Barrier(comm)`
- Finalize MPI environment

  `MPI_Finalize()`

First five MPI commands

C & Fortran bindings

```
int MPI_Init(int *argc, char **argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Barrier(MPI_Comm comm)
MPI_Finalize()

MPI_INIT(ierror)
MPI_COMM_SIZE(comm, size, ierror)
MPI_COMM_RANK(comm, rank, ierror)
MPI_BARRIER(comm, ierror)
MPI_FINALIZE(ierror)
integer comm, size, rank, ierror
```

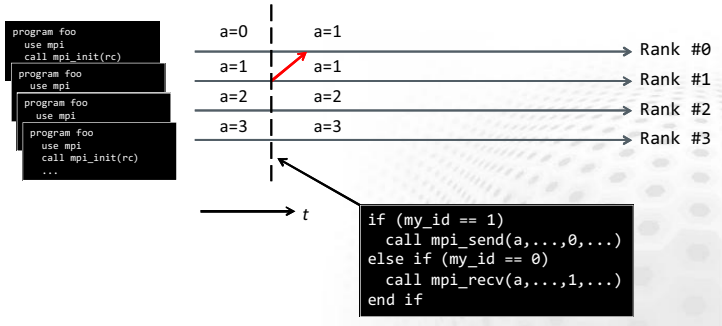
Writing an MPI program

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

Summary

- In MPI, a set of *independent processes* is launched
  - Processes are identified by *ranks*
  - Data is always *local* to the process
- Processes can exchange data by sending and receiving *messages*
- The MPI library contains procedures for
  - Communication = exchanging data between processes
  - Synchronizing processes
  - Communicator manipulation
  - I/O, etc

MPI execution model summary



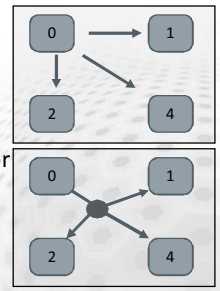
Point-to-point communication

MPI point-to-point operations

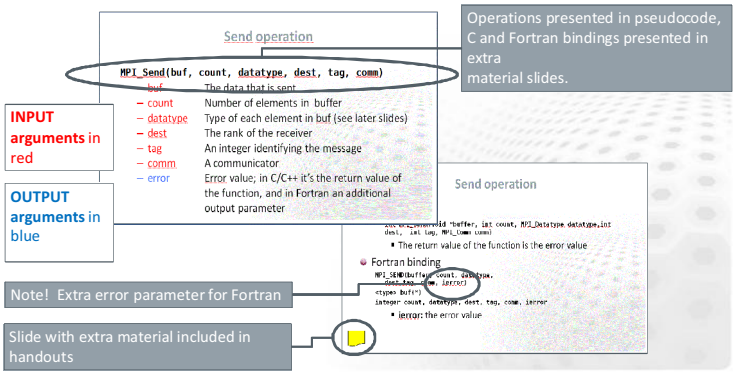
- Each message (envelope) contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data.
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

MPI communication

- MPI processes are *independent*, they *communicate* to coordinate work
- Point-to-point communication
  - Messages are sent between two processes, others not being affected (nor aware) of the communication
- Collective communication
  - Involving a number of processes at the same time
  - All processes participate



Presenting syntax



PART I: BASIC POINT-TO-POINT OPERATIONS

Send operation

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

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

MPI point-to-point operations

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

Send operation

- C/C++ binding

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

  - The return value of the function is the error value
- Fortran binding

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

  - **ierror:** the error value

Receive operation

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

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

MPI datatypes

MPI type	Fortran type
MPI_CHARACTER	CHARACTER
MPI_INTEGER	**INTEGER**
MPI_REAL	**REAL**
MPI_REAL8	REAL*8 (non-standard)
MPI_DOUBLE_PRECISION	**DOUBLE PRECISION**
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	

Receive operation

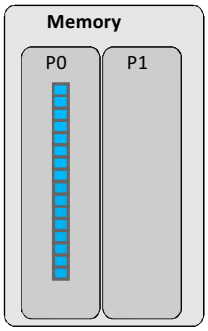
- C/C++ binding

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

- Fortran binding

```
mpi_recv(buf, count, datatype, source, tag, comm, status, ierror)
<type>, dimension(*) :: buf
integer :: count, datatype, source, tag, comm, ierror
integer, dimension(MPI_STATUS_SIZE) :: status
```

Case study: parallel sum

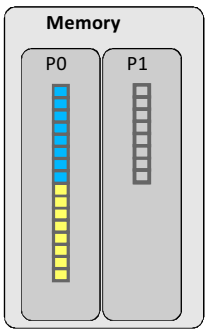


- Array originally on process #0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segments
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

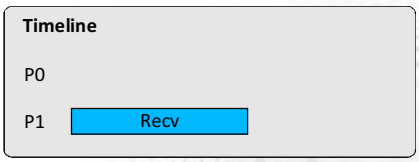
MPI datatypes

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

Case study: parallel sum



Step 1.1: Receive operation in scatter

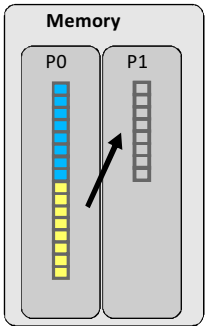


P1 posts a receive to receive half of the array from P0

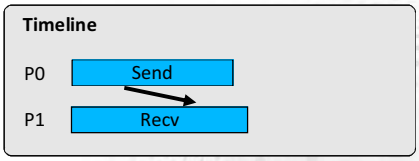
MPI datatypes

MPI type	C type
MPI_CHAR	signed char
MPI_SHORT	short int
MPI_INT	**int**
MPI_LONG	long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	long double
MPI_BYTE	

Case study: parallel sum

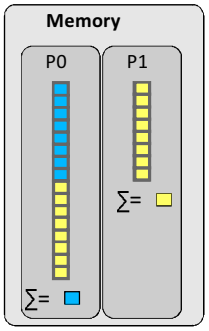


Step 1.2: Send operation in scatter

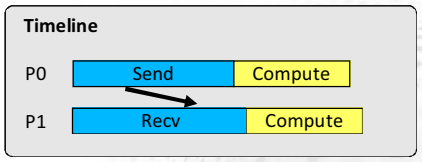


P0 posts a send to send the lower part of the array to P1

Case study: parallel sum



Step 2: Compute the sum in parallel

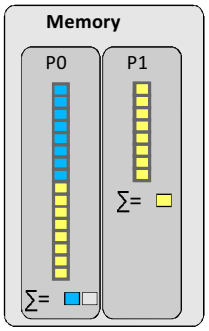


P0 & P1 computes their parallel sums and store them locally

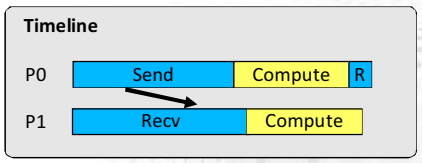
Exercise session

- Write, compile and run a Hello World style dummy program employing MPI
- Do the Exercise 1 a

Case study: parallel sum



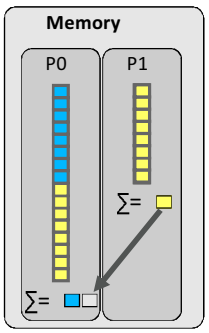
Step 3.1: Receive operation in reduction



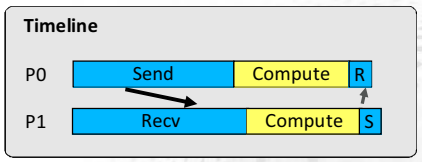
P0 posts a receive to receive partial sum

PART II: MORE ABOUT POINT-TO-POINT OPERATIONS

Case study: parallel sum



Step 3.2: send operation in reduction

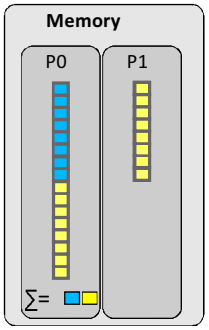


P1 posts a send with partial sum

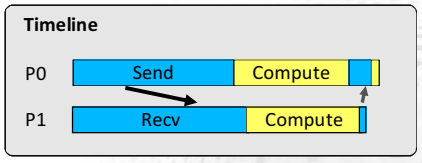
Blocking routines & deadlocks

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

Case study: parallel sum



Step 4: Compute final answer



P0 sums the partial sums

Point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data

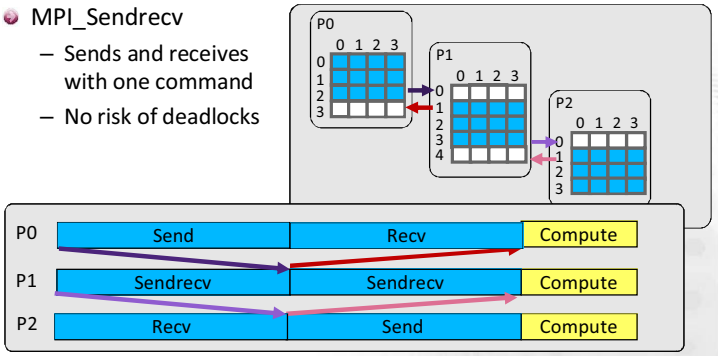
Combined send & receive

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

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

CS2: MPI_Sendrecv

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



Combined send & receive

- C/C++ binding

```
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 )
```

- Fortran binding

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

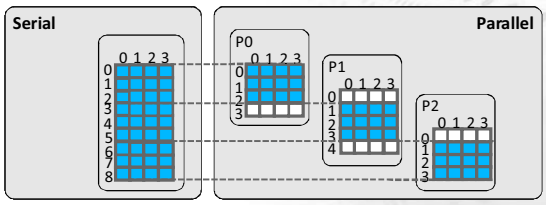
Special parameter values

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

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

Case study 2: Domain decomposition

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



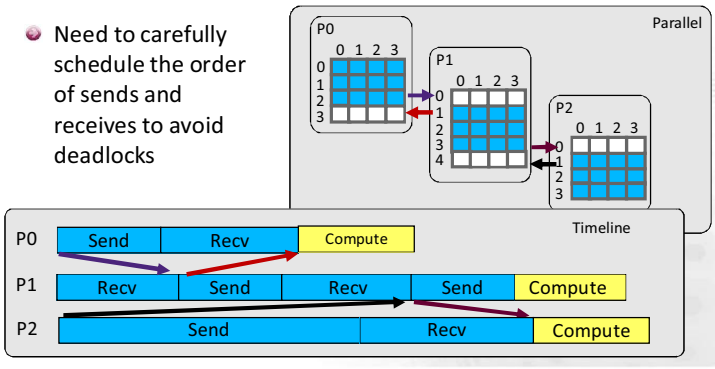
Special parameter values

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

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

CS2: One iteration step

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



Status parameter

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

Status parameter

- Received elements

  Use the function

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

- Tag of the received message

  C: status.MPI_TAG, Fortran: status(MPI_TAG)

- Rank of the sender

  C: status.MPI_SOURCE, Fortran: status(MPI_SOURCE)

Summary

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

Exercise session

- Do the Exercises 1 b-f
- Start working on the Game of Life, Exercise 7 a

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

Introduction

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

Example
MPI_Barrier
makes each task hold until all tasks have called it

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

MPI_Bcast

C & Fortran bindings

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

MPI_BCAST(buffer, count, datatype, root, comm, ierror)
type buffer(*)
integer count, datatype, root, comm, ierror
```

Introduction

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

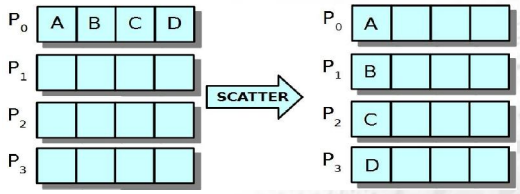
```
if (my_id == 0) then
   do i = 1, ntasks-1
     call mpi_send(a, 1048576, &
          MPI_REAL, i, tag, &
          MPI_COMM_WORLD, rc)
   end do
else
   call mpi_recv(a, 1048576, &
          MPI_REAL, 0, tag, &
       MPI_COMM_WORLD, status, rc)
end if
```

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

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

Scattering

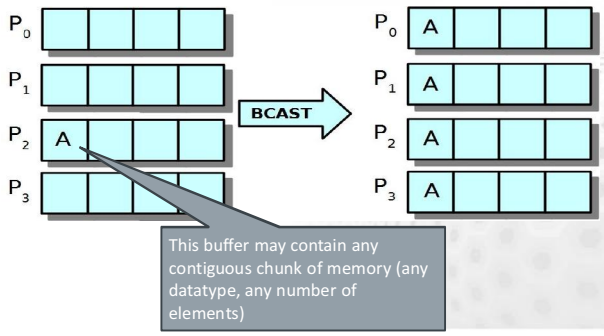
- Send equal amount of data from one process to others



- Segments A, B, … may contain multiple elements

Broadcasting

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



Scattering

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

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

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

MPI_Scatter

C & Fortran bindings

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

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
            recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, recvcount, sendtype, recvtype, root, comm,
        ierror
```

MPI_Scatterv

C & Fortran bindings

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

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,
             recvcount, recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcounts(*), displs(*), recvcount, sendtype,
        recvtype, root, comm, ierror
```

One-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)
if (my_id==3) print *, a(:)
```

```
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(:)
```

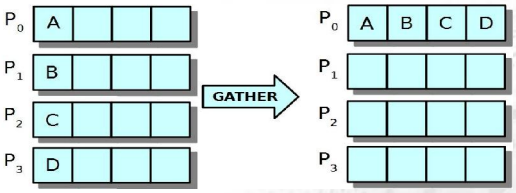
Assume 4 MPI tasks. What would the (full) program print?

```
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
```

```
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
```

Gathering

- Collect data from all the process to one process



- Segments A, B, … may contain multiple elements

Varying-sized scatter

- Like MPI_Scatter, but messages can have different sizes and displacements

```
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 in receive buffer
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

Gathering

- MPI_Gather: Collect equal share of data (in *sendbuf*) 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

Scatterv example

```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
  sendcnts = (/ 1, 2, 3, 4 /)
  displs = (/ 0, 1, 3, 6 /)
end if
call mpi_scatterv(a, sendcnts, &
     displs, MPI_INTEGER,&
     aloc, 4, MPI_INTEGER, &
     0, MPI_COMM_WORLD, rc)
```

```
A.  1 2 3
B.  7 8 9 10
C.  1 2 3 4 5 6 7 8 9 10
```

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

MPI_Gather

C and Fortran bindings

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

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
           recvtype, root, comm, ierror)
type     sendbuf(*), recvbuf(*)
integer sendcount, recvcount, sendtype, recvtype, root, comm,
        ierror
```

Varying-sized gather

- MPI_Gatherv is similar to MPI_Gather, but allows for varying amounts of data

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

sendbuf	send buffer
sendcount	number of elements in send buffer
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcnts	array of number of elements to receive from each task
displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf where to put the incoming data from process i
recvtype	data type of recv buffer elements
root	rank of receiving process
comm	communicator

Reduce operation

- Available reduction operations (argument op)

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

MPI_Gatherv

C and Fortran bindings

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

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
            displs, recvtype, root, comm, ierror)

type    sendbuf(*), recvbuf(*)
integer sendcount, recvcounts(*), displs(*), sendtype,
        recvtype, root, comm, ierror
```

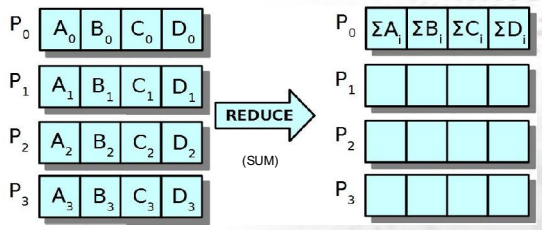
Reduce operation

C and Fortran bindings

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

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root,
           comm, ierror)

type sendbuf(*), recvbuf(*)
integer count, datatype, op, root, comm, ierror
```

Reduce operation

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

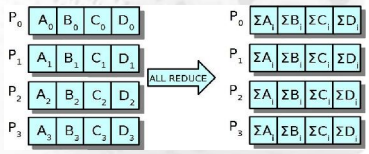


Global reduce operation

- 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	send buffer
recvbuf	receive buffer
count	number of elements in send buffer
datatype	data type of elements in send buffer
op	operation
comm	communicator



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 of send buffer
op	operation
root	rank of root process
comm	communicator

MPI_Allreduce

C & Fortran bindings

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

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm,
              ierror)

type :: sendbuf(*), recvbuf(*)
integer :: count, datatype, op, comm, ierror
```

Allreduce example: parallel dot product

```
real :: a(1024), aloc(128)
...
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)
```

```
> mpirun -np 8 ./a.out
 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
```

MPI_Reduce_scatter

C & Fortran bindings

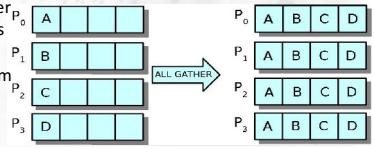
```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf,
                       int* recvcounts, MPI_Datatype datatype,
                       MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype,
                   op, comm, ierror)

type :: sendbuf(*), recvbuf(*)
integer :: recvcounts(*), datatype, op, comm, ierror
```

All-to-one plus one-to-all

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

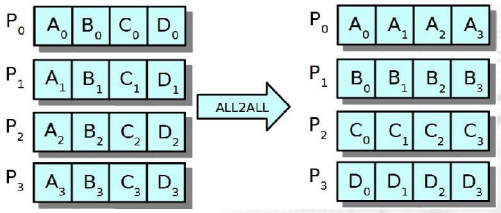
```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
  recvcount, recvtype, comm)
```

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



From each to every

- Send a distinct message from each task to every task



- "Transpose" like operation

MPI_Allgather

C & Fortran bindings

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

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
              recvtype, comm, ierror)

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

From each to every

- MPI_Alltoall 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 to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer elements
comm	communicator

All-to-one plus one-to-all

- MPI_Reduce_scatter applies a reduction operation to sendbuf over the tasks and scatters the result according to the values in recvcounts
  - Compare: MPI_Reduce + MPI_Scatter

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts,
                   datatype, op, comm)
```

sendbuf	send buffer
recvbuf	receive buffer
recvcounts	array specifying the number of elements in result distributed to each process
datatype	data type of elements of input buffer
op	operation
comm	communicator

MPI_Alltoall

C & Fortran bindings

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

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf,
             recvcount,  recvtype, comm, ierror)

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

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)
```

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

```
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
```

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
- A performance consideration: All-to-all are expensive operations, avoid them if possible

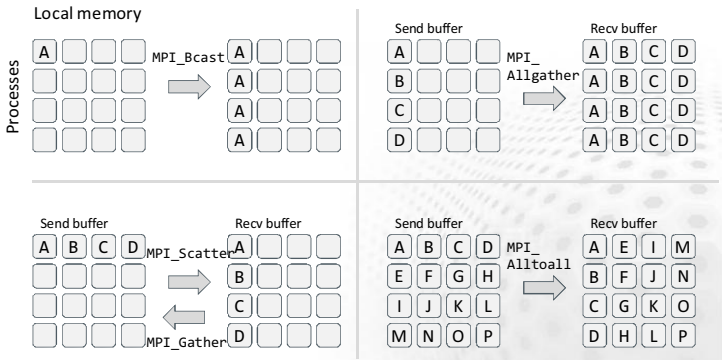
Varying-sized all-to-all

- MPI_Alltoallv is similar to MPI_Alltoall, but messages can have different sizes and displacements

```
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype,
              recvbuf, recvcounts, rdispls, recvtype,
              comm)
```

sendbuf	send buffer	recvcounts	maximum numbers of elements that can be received from each process
sendcounts	number of elements to send to each process	rdispls	displacements relative to recvbuf
sdispls	displacements relative to sendbuf	recvtype	data type of receive buffer elements
sendtype	data type of send buffer elements	comm	communicator
recvbuf	receive buffer		

Main collectives recap



MPI_Alltoallv

C & Fortran bindings

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


MPI_ALLTOALLV(sendbuf,sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, ierror)

type :: sendbuf(*), recvbuf(*)
integer :: sendcounts(*), recvcounts(*), sdispls(*), rdispls(*),
        sendtype, recvtype, comm, ierror
```

Exercise session

- Do the Exercises 2 a and b

Common mistakes with collectives

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

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

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

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

Non-blocking send

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

Parameters

Similar to MPI_Send but has an request parameter

buf send buffer cannot 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

Non-blocking communication

- *Non-blocking* sends and receives: MPI_Isend & MPI_Irecv
  - returns immediately and sends/receives in background
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations

Non-blocking send

- C/C++ binding

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

- Fortran binding

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Non-blocking communication

- 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)
- You can mix non-blocking and blocking routines
  - Receive MPI_Isend with MPI_Recv or vice versa

Non-blocking receive

```
MPI_Irecv(buf, count, datatype, dest, tag,
          comm, request)
```

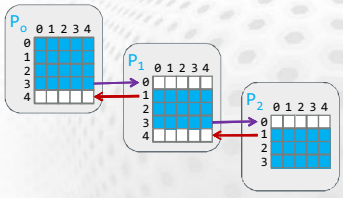
parameters similar to MPI_Recv but has no status parameter

buf 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

Typical usage pattern

```
MPI_Irecv(ghost_data)
MPI_Isend(border_data)
Compute(ghost_independent_data)
MPI_Waitall(receives)
Compute(border_data)
MPI_Waitall(sends)
```



Non-blocking receive

- C/C++ binding

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

- Fortran binding

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

Waiting for a non-blocking operation

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

Parameters

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

Additional completion operations

- other useful routines:
  - **MPI_Waitany**
  - **MPI_Waitsome**
  - **MPI_Test**
  - MPI_Testall
  - MPI_Testany
  - MPI_Testsome
  - MPI_Probe

Waiting for a non-blocking operation

- C/C++ binding

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

- Fortran binding

```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER :: REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

Wait for non-blocking operations

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

Parameters

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

Waiting for several non-blocking operations

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

Parameters

count	number of requests
requests	array of requests
status	array of statuses for the operations that are waited for

MPI_Waitall returns when *all* operations identified by the array of requests are complete

Wait for non-blocking operations

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

Parameters

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

A call to MPI_Waitsome returns when one or more operation identified by the array of requests is complete

Waiting for several non-blocking operations

- C/C++ binding

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

- Fortran binding

```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
INTEGER :: COUNT, ARRAY_OF_REQUESTS(:),
  ARRAY_OF_STATUSES(MPI_STATUS_SIZE,:), IERROR
```

Non-blocking test for non-blocking operations

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

Parameters

request	request
flag	True if operation has completed
status	status for the completed operations

A call to MPI_Test is non-blocking

Allows one to schedule alternative activities while periodically checking for completion

Non-blocking collectives

- The MPI standard version 3.0 introduces non-blocking collectives
  - MPI_Ibcast, MPI_Igather, MPI_Iscatter,…
  - They return immediately, and completion has to be waited
- Routine interfaces similar to the blocking ones but with an additional request argument for MPI_Wait
- Mixing blocking and non-blocking collectives for the same transmit is not possible

Summary

- Non-blocking communication is usually the smarter way to do point-to-point communication in MPI
- Non-blocking communication realization
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Wait`
- Non-blocking collectives available soon as well

Exercise session

- Do the Exercise 3
- Bonus exercise (a solution can be shown on request): modify the MPI implementation of the Game of Life (Ex 7a) such that the communication is done with non-blocking routines.
- Bonus exercise 2: modify the non-blocking GoL such that the board update and the boundary communication is being overlapped.

Creating a communicator

- MPI_Comm_split creates new communicators based on 'colors' and 'keys'

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

comm	communicator handle
color	control of subset assignment, processes with the same color belong to the same new communicator
key	control of rank assignment
newcomm	new communicator handle

If color = MPI_UNDEFINED, a process does not belong to any of the new communicators

Communicators

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

Creating a communicator

- C and Fortran bindings

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

MPI_COMM_SPLIT (comm, color, key, newcomm, rc)
integer comm, color, key, newcomm, rc
```

- Return code values

MPI_SUCCESS	No error; MPI routine completed successfully.
MPI_ERR_COMM	Invalid communicator. A common error is to use a null communicator in a call
MPI_ERR_INTERN	This error is returned when some part of the implementation is unable to acquire memory.

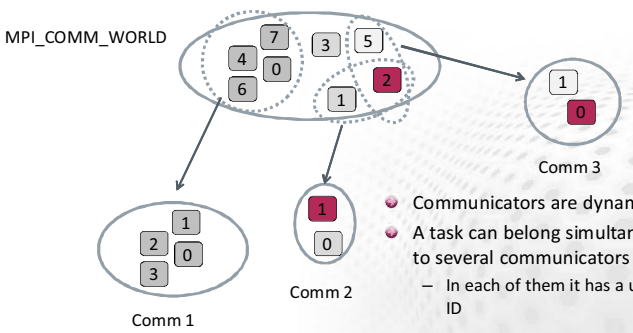
PART I: CREATING OWN COMMUNICATORS

Creating a communicator

```
if (myid%2 == 0) {
  color = 1;
} else {
  color = 2;
}
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);
MPI_Comm_rank(subcomm, &mysubid);
printf ("I am rank %d in MPI_COMM_WORLD, but %d in
        Comm %d.\n",myid, mysubid, color);
```

```
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

Grouping processes in communicators



- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - In each of them it has a unique ID

Communicator manipulation

MPI_Comm_size	Returns number of processes in communicator's group
MPI_Comm_rank	Returns rank of calling process in communicator's group
MPI_Comm_compare	Compares two communicators
MPI_Comm_dup	Duplicates a communicator
MPI_Comm_free	Marks a communicator for deallocation

Exercise session

- Do the Exercise 4
- Unless you have a working MPI version (Ex 7a) of the GoL, it would now be a good time to complete it

Creating a communication topology

- C and Fortran bindings

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

MPI_CART_CREATE(old_comm, ndims, dims, periods,
    reorder, comm_cart, rc)
  integer :: old_comm, ndims, dims(:), comm_cart, rc
  logical :: reorder, periods(:)
```

PART II: PROCESS TOPOLOGIES

Ranks and coordinates

- Translate a rank to coordinates

  MPI_Cart_coords(comm, rank, maxdim, coords)

comm	Cartesian communicator
rank	rank to convert
maxdim	dimension of *coords*
coords	coordinates in Cartesian topology that corresponds to *rank*

Process topologies

- MPI process topologies allow for simple referencing scheme of processes
  - Cartesian and graph topologies are supported
  - Process topology defines a new communicator
- MPI topologies are virtual
  - No relation to the physical structure of the computer
  - Data mapping "more natural" only to the programmer
- Usually no performance benefits
  - But code becomes more compact and readable

Ranks and coordinates

- Translate a set of coordinates to a rank

  MPI_Cart_rank(comm, coords, rank)

comm	Cartesian communicator
coords	array of coordinates
rank	a rank corresponding to *coords*

Creating a communication topology

- New communicator with processes ordered in a Cartesian grid

  MPI_Cart_create(oldcomm, ndims, dims, periods, reorder, newcomm)

oldcomm	communicator
ndims	dimension of the Cartesian topology
dims	integer array (size ndims) that defines the number of processes in each dimension
periods	array that defines the periodicity of each dimension
reorder	is MPI allowed to renumber the ranks
newcomm	new Cartesian communicator

Ranks and coordinates

- C and Fortran bindings

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdim, int *coords)
MPI_CART_COORDS(comm, rank, maxdim, coords, rc)
   integer :: comm, rank, maxdim, coords(:), rc

int MPI_Cart_rank(MPI_Comm comm, int *coords, int rank)
MPI_CART_RANK(comm, coords, rank, rc)
   integer :: comm, coords(:), rank, rc
```

Creating a communication topology

```
dims(1)=4
dims(2)=4
period=(/ .true. , .true. /)

call mpi_cart_create(mpi_comm_world, 2,&
    dims, period, .true., comm2d, rc)
call mpi_comm_rank(comm2d, my_id, rc)
call mpi_cart_coords(comm2d, my_id, 2,&
    coords, rc)
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Neighborhood collectives on process topologies

- A new feature of MPI 3.0 are in-build routines for exchanging data with the nearest neighbors
  - With Cartesian topologies, only nearest neighbor communication (corresponding to MPI_Cart_shift with displ=1) is supported
- These routines include MPI_Neighbor_allgather, MPI_Neighbor_alltoall
  - Varying-size variants as well as non-blocking versions available

Communication in a topology

- Counting sources/destinations on the grid
  - for e.g. elegant nearest-neighbor communication

```
MPI_Cart_shift(comm, direction, displ, source,
               dest)
```

comm	Cartesian communicator
direction	shift direction (e.g. 0 or 1 in 2D)
displ	shift displacement (1 for next cell etc, < 0 for source from "down"/"right" directions)
source	rank of source process
dest	rank of destination process

Note that both source and dest are output parameters. The coordinates of the calling task is implicit input.

With a non-periodic grid, source or dest can land outside of the grid; then MPI_PROC_NULL is returned.

Summary

- In real-world applications it is very often beneficial to divide MPI ranks into subsets
  - Conceptual division for readability etc
  - Improving parallel scalability by avoiding global synchronizations
- MPI allows for ordering processes into topologies
  - Readability, programmer performance
  - Topology forms a new communicator
  - MPI 3.0 introduces topology-aware collectives

Communication in a topology

- C and Fortran bindings

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int
   *source, int *dest )

MPI_CART_SHIFT(comm, direction, displ, source, dest, rc)
   integer :: comm, direction, displ, source, dest, rc
```

Exercise session

- Do the Exercise 5
- Bonus: Decompose the GoL board in two dimensions. Employ a Cartesian process topology. See Exercise 7b.

Halo exchange

```
dims(1)=4
dims(2)=4
period =(/ .true. , .true. /)

call mpi_cart_create(mpi_comm_world, 2,&
    dims, period, .true., comm2d, rc)
call mpi_cart_shift(comm2d,0,1,nbr_up,nbr_down,rc)
call mpi_cart_shift(comm2d,1,1,nbr_left,nbr_right,rc)
...
call mpi_sendrecv(hor_send, msglen, mpi_double_precision, nbr_left,&
     tag_left, hor_recv, msglen, mpi_double_precision, nbr_right,&
     tag_left, comm2d, mpi_status_ignore, rc)
...
call mpi_sendrecv(vert_send, msglen, mpi_double_precision, nbr_up,&
     tag_up, vert_recv, msglen, mpi_double_precision, nbr_down,&
     tag_up, comm2d, mpi_status_ignore, rc)
...
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

User-defined datatypes

- User-defined datatypes can be used both in point-to-point communication and collective communication
- The datatype instructs where to take the data when sending or where to put data when receiving
  - Non-contiguous data in sending process can be received as contiguous or vice versa

MPI datatypes

- MPI datatypes are used for communication purposes
  - Datatype tells MPI where to take the data when sending or where to put data when receiving
- Elementary datatypes (MPI_INT, MPI_REAL, ...)
  - Different types in Fortran and C, correspond to languages basic types
  - Enable communication using contiguous memory sequence of identical elements (e.g. vector or matrix)

Using user-defined datatypes

- A new datatype is created from existing ones with a datatype constructor
  - Several routines for different special cases
- A new datatype must be committed before using it

```
MPI_Type_commit(newtype)
```

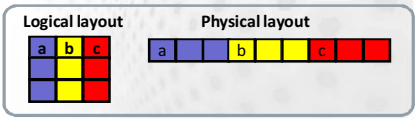
  newtype the new datatype to commit
- A type should be freed after it is no longer needed

```
MPI_Type_free(newtype)
```

  newtype the datatype for decommision

Sending a matrix row (Fortran)

- Row of a matrix is not contiguous in memory in Fortran
- Several options for sending a row:
  - Use several send commands for each element of a row
  - Copy data to temporary buffer and send that with one send command
  - Create a matching datatype and send all data with one send command



Some selected datatype constructors

MPI_Type_contiguous	contiguous datatypes
MPI_Type_vector	regularly spaced datatype
MPI_Type_indexed	variably spaced datatype
MPI_Type_create_subarray	subarray within a multi-dimensional array
MPI_Type_create_struct	fully general datatype

User-defined datatypes

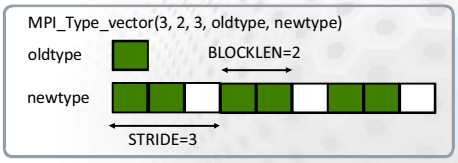
- Use elementary datatypes as building blocks
- Enable communication of
  - Non-contiguous data with a single MPI call, e.g. rows or columns of a matrix
  - Heterogeneous data (structs in C, types in Fortran)
- Provide higher level of programming & efficiency
  - Code is more compact and maintainable
  - Communication of non-contiguous data is more efficient
- Needed for getting the most out of MPI I/O

MPI_TYPE_VECTOR

- Creates a new type from equally spaced identical blocks

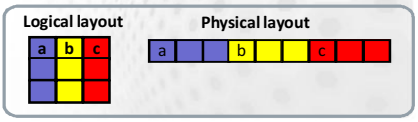
```
MPI_Type_vector(count, blocklen, stride, oldtype,
                newtype)
```

count number of blocks
blocklen number of elements in each block
stride displacement between the blocks



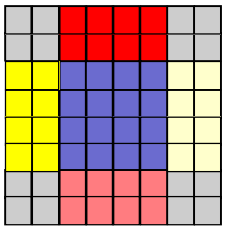
Example: sending rows of a matrix in Fortran

```
integer, parameter :: n=3, m=3
real, dimension(n,m) :: a
integer :: rowtype
! create a derived type
call mpi_type_vector(m, 1, n, mpi_real, rowtype, ierr)
call mpi_type_commit(rowtype, ierr)
! send a row
call mpi_send(a, 1, rowtype, dest, tag, comm, ierr)
! free the type after it is not needed
call mpi_type_free(rowtype, ierr)
```



Example: halo exchange with user defined types

Two-dimensional grid with two-element ghost layers



```
int array_size[2] = {8,8};
int x_size[2] = {2,4};
int xl_start[2] = {0,2};
MPI_Type_create_subarray(2, array_size, x_size,
      xl_start, MPI_ORDER_C, MPI_DOUBLE,
      &xl_boundary);
```

```
int array_size[2] = {8,8};
int y_size[2] = {4,2};
int yd_start[2] = {2,0};
MPI_Type_create_subarray(2, array_size, y_size,
      yd_start, MPI_ORDER_C, MPI_DOUBLE,
      &yd_boundary);
```

MPI_TYPE_INDEXED

- Creates a new type from blocks comprising identical elements
  - The size and displacements of the blocks may vary

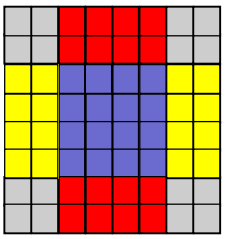
```
MPI_Type_indexed(count, blocklens, displs,
                 oldtype, newtype)
```

count	number of blocks
blocklens	lengths of the blocks (array)
displs	displacements (array) in extent of oldtypes

count = 3
blocklens = (/2,3,1/)
disps = (/0,3,8/)

Example: halo exchange with user defined types

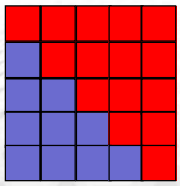
Two-dimensional grid with two-element ghost layers



```
MPI_Sendrecv(array, 1, xl_boundary, nbr_left,
 tag_left, array, 1, xr_boundary, nbr_right,
 tag_right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(array, 1, xr_boundary, nbr_right,
 tag_right, array, 1, xl_boundary, nbr_left,
 tag_left, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(array, 1, yd_boundary, nbr_down,
 tag_down, array, 1, yu_boundary, nbr_up, tag_up,
 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(array, 1, yu_boundary, nbr_up, tag_up,
 array, 1, yd_boundary, nbr_down, tag_down,
 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Example: an upper triangular matrix

```
/* Upper triangular matrix */
double a[100][100];
int  disp[100], blocklen[100], int i;
MPI_Datatype upper;
/* compute start and size of rows */
for (i=0;i<100;i++)
  {
    disp[i]=100*i+i;
    blocklen[i]=100-i;
  }
/* create a datatype for upper triangular matrix */
MPI_Type_indexed(100,blocklen,disp,MPI_DOUBLE,&upper);
MPI_Type_commit(&upper);
/* ... send it ... */
MPI_Send(a,1,upper,dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&upper);
```



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 )
```

MPI_TYPE_CREATE_SUBARRAY

- Creates a type describing an *N*-dimensional subarray within an *N*-dimensional array

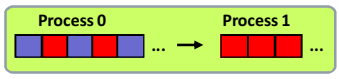
```
MPI_Type_create_subarray(ndims, sizes, subsizes,
                         offsets, order, oldtype, newtype)
```

ndims	number of array dimensions
sizes	number of array elements in each dimension (array)
subsizes	number of subarray elements in each dimension (array)
offsets	starting point of subarray in each dimension (array)
order	storage order of the array. Either MPI_ORDER_C or MPI_ORDER_FORTRAN

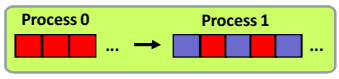
Fortran interfaces for datatype routines

```
mpi_type_commit(type, rc)
  integer :: type, rc

mpi_type_free(type, rc)
  integer :: type, rc

mpi_type_contiguous(count, oldtype, newtype, rc)
  integer :: count, oldtype, newtype, rc

mpi_type_vector(count, block, stride, oldtype, newtype, rc)
  integer :: count, block, stride, oldtype, newtype, rc

mpi_type_indexed(count, blocks, displs, oldtype, newtype, rc)
  integer :: count, oldtype, newtype, rc
  integer, dimension(count) :: blocks, displs

mpi_type_create_subarray(ndims, sizes, subsizes, starts, order,
  oldtype, newtype, rc)
  integer :: ndims, order, oldtype, newtype, rc
  integer, dimension(ndims) :: sizes, subsizes, starts
```

From non-contiguous to contiguous data




```
if (myid == 0)
   MPI_Type_vector(n, 1, 2,
                   MPI_FLOAT, &newtype)
   ...
   MPI_Send(A, 1, newtype, 1, ...)
else
   MPI_Recv(B, n, MPI_FLOAT,0, ...)
```




```
if (myid == 0)
   MPI_Send(A, n, MPI_FLOAT, 1, ...)
else
   MPI_Type_vector(n, 1, 2, MPI_FLOAT,
                   &newtype)
   ...
   MPI_Recv(B, 1, newtype,0, ...)
```

Performance

- Performance depends on the datatype - more general datatypes are often slower
- Overhead is potentially reduced by:
  - Sending one long message instead of many small messages
  - Avoiding the need to pack data in temporary buffers
- Performance should be tested on target platforms
- Example: MPI_Type_vector with blocksize=2 and stride=20 (Cray XT5)
  - Performance almost 10x better than naïve manual packing

Summary

- Derived types enable communication of non-contiguous or heterogenous data with single MPI calls
  - Improves maintainability of program
  - Allows optimizations by the system
  - Performance is implementation dependent
- Life cycle of a user-defined type: create, commit, free
- MPI provides constructors for several specific types

Exercise session

- Do the Exercise 6
- Bonus: (Continue with the) Exercise 7b

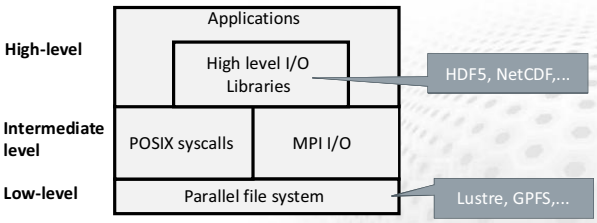
Parallel I/O with MPI

Parallel I/O

- New challenges
  - Number of tasks is rising rapidly
  - The size of the data is also rapidly increasing
  - Gap between computing power vs. I/O rates increasing rapidly
- The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!

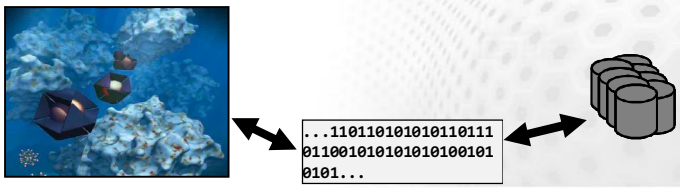
PART I: INTRODUCTION TO PARALLEL I/O

I/O layers



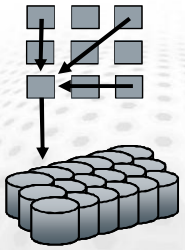
Parallel I/O

- How to convert internal data structures and domains to files which are a streams of bytes?
- How to get the data efficiently from thousands of nodes of a supercomputer to physical disks?



Parallel POSIX I/O

- Spokesman strategy
  - One process takes care of all I/O using normal (POSIX) routines
  - Requires a lot of communication
  - Writing/reading slow, single writer not able to fully utilize filesystem
  - Does not scale, single writer is a bottleneck
  - Can be good option when the amount of data is small (e.g. input files)

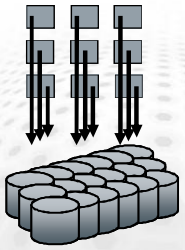


Parallel I/O

- Good I/O is non-trivial
  - Performance, scalability, reliability
  - Ease of use of output (number of files, format)
  - Portability
- One cannot achieve all of the above - one needs to prioritize

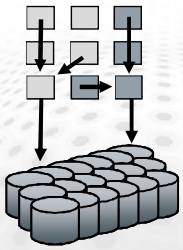
Parallel POSIX I/O

- Every man for himself
  - Each process writes its local results to a separate file
  - Good bandwidth
  - Difficult to handle a huge number of files in later analysis
  - Can overwhelm filesystem (for example Lustre metadata)



Parallel POSIX I/O

- Subset of writers/readers
  - Good compromise
  - Most difficult to implement
  - Number of readers/writers often chosen to be sqrt(ntasks)
  - If readers/writers are dedicated then some computational resources are wasted



Basic concepts in MPI I/O

- File *view*
  - part of a parallel file which is visible to process
  - enables efficient noncontiguous access to file
- *Collective* and *independent* I/O
  - Collective = MPI coordinates the reads and writes of processes
  - Independent = no coordination by MPI

PART II: MPI I/O BASICS

Opening & Closing files

- All processes in a communicator open a file using

```
MPI_File_open(comm, filename, mode, info, fhandle)
```

comm	communicator that performs parallel I/O
mode	MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_CREATE, MPI_MODE_RDWR, …
info	Hints to implementation for optimal performance (No hints: MPI_INFO_NULL)
fhandle	parallel file handle

Can be combined with + in Fortran and | in C/C++

- File is closed using

```
MPI_File_close(fhandle)
```

MPI I/O

- Defines parallel operations for reading and writing files
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O
- Potentially good performance, easy to use (compared with implementing the same algorithms on your own)
- Portable programming *interface*
  - By default, binary *files* are not portable

File pointer

- Each process moves its local file pointer (individual file pointer) with

```
MPI_File_seek(fhandle, disp, whence)
```

disp	Displacement in bytes (with default file view)
whence	MPI_SEEK_SET: the pointer is set to offset MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset MPI_SEEK_END: the pointer is set to the end of file plus offset

Basic concepts in MPI I/O

- File *handle*
  - data structure which is used for accessing the file
- File *pointer*
  - *position* in the file where to read or write
  - can be individual for all processes or shared between the processes
  - accessed through file handle

File reading

- Read file at individual file pointer

```
MPI_File_read(fhandle, buf, count, datatype,
    status)
```

buf	Buffer in memory where to read the data
count	number of elements to read
datatype	datatype of elements to read
status	similar to status in MPI_Recv, amount of data read can be determined by MPI_Get_count

  - Updates position of file pointer after reading
  - Not thread safe

File writing

- Similar to reading
  - File opened with MPI_MODE_WRONLY or MPI_MODE_CREATE
- Write file at individual file pointer

```
MPI_File_write(fhandle, buf, count, datatype,
    status)
```

  - Updates position of file pointer after writing
  - Not thread safe

File writing, explicit offset

- Determine location within the write statement (explicit offset)

```
MPI_File_write_at(fhandle, disp, buf, count,
 datatype, status)
```

  - Thread-safe
  - The file pointer is neither used or incremented

Example: parallel write

```
program output
 use mpi
 implicit none
 integer :: err, i, myid, file, intsize
 integer :: status(mpi_status_size)
 integer, parameter :: count=100
 integer, dimension(count) :: buf
 integer(kind=mpi_offset_kind) :: disp
 call mpi_init(err)
 call mpi_comm_rank(mpi_comm_world, myid,&
   err)
 do i = 1, count
  buf(i) = myid * count + i
 end do
...
```

Multiple processes write to a binary file 'test'. First process writes integers 1-100 to the beginning of the file, etc.

Example: parallel read

Note: Same number of processes for reading and writing is assumed in this example.

File offset determined explicitly

```
...
 call mpi_file_open(mpi_comm_world, 'test', &
      mpi_mode_rdonly, mpi_info_null, file, err)
 intsize = sizeof(count)
 disp = myid * count * intsize
 call mpi_file_read_at(file, disp, buf, &
      count, mpi_integer, status, err)
 call mpi_file_close(file, err)
 call mpi_finalize(err)
end program output
```

Example: parallel write

Note: File (and total data) size depends on number of processes in this example

File offset determined by MPI_File_seek

```
...
 call mpi_file_open(mpi_comm_world, 'test', &
      mpi_mode_wronly + mpi_mode_create, &
      mpi_info_null, file, err)
 call mpi_type_size(mpi_integer, intsize,err)
 disp = myid * count * intsize
 call call mpi_file_seek(file, disp, mpi_seek_set, err)
 call mpi_file_write(file, buf, count, mpi_integer, &
      status, err)
 call mpi_file_close(file, err)
 call mpi_finalize(err)
end program output
```

Collective operations

- I/O can be performed *collectively* by all processes in a communicator
  - MPI_File_read**_all**
  - MPI_File_write**_all**
  - MPI_File_read_at**_all**
  - MPI_File_write_at**_all**
- Same parameters as in independent I/O functions (MPI_File_read etc)

File reading, explicit offset

- The location to read or write can be determined also explicitly with

```
MPI_File_read_at(fhandle, disp, buf, count,
    datatype, status)
```

disp displacement in bytes (with the default file view) from the beginning of file

  - Thread-safe
  - The file pointer is neither referred or incremented

Collective operations

- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous

Non-blocking MPI I/O

- Non-blocking independent I/O is similar to non-blocking send/recv routines
  - MPI_File_iread
  - MPI_File_iwrite
  - MPI_File_iread_at
  - MPI_File_iwrite_at
- Wait for completion using MPI_Test, MPI_Wait, etc.
- Can be used to overlap I/O with computation

File view

```
MPI_File_set_view(fhandle, disp, etype,
   filetype, datarep, info)
```

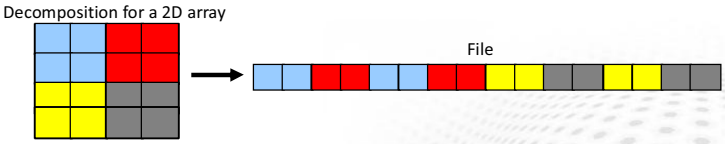
disp	Offset from beginning of file. Always in bytes
etype	Basic MPI type or user defined type Basic unit of data access
filetype	Same type as etype or user defined type constructed of etype
datarep	Data representation (can be adjusted for portability) "native": store in same format as in memory
info	Hints for implementation that can improve performance MPI_INFO_NULL: No hints

The values for datarep and the extents of etype must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in must be committed.

Exercise session

- Do the Exercise 7 b

File view for non-contiguous data

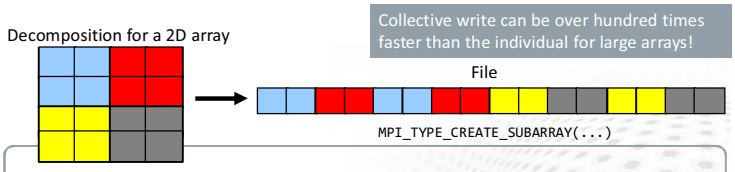


- Each process has to access small pieces of data scattered throughout a file
- Very expensive if implemented with separate reads/writes
- Use file type to implement the non-contiguous access

PART III: NON-CONTIGUOUS DATA ACCESS WITH MPI I/O

File view for non-contiguous data

Collective write can be over hundred times faster than the individual for large arrays!




```
...
integer, dimension(2,2) :: array
...
call mpi_type_create_subarray(2, sizes, subsizes, starts, mpi_integer, &
     mpi_order_c, filetype, err)
call mpi_type_commit(filetype)
disp = 0
call mpi_file_set_view(file, disp, mpi_integer, filetype, 'native', &
     mpi_info_null, err)
call mpi_file_write_all(file, buf, count, mpi_integer, status, err)
```

File view

- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- The *file view* defines which portion of a file is visible to a process
- A file view consists of three components
  - displacement: number of bytes to skip from the beginning of file
  - etype: type of data accessed, defines unit for offsets
  - filetype: portion of file visible to a process

Common mistakes with MPI I/O

- ✘ Not defining file offsets as MPI_Offset in C and integer (kind=MPI_OFFSET_KIND) in Fortran
- ✘ In Fortran, passing the offset or displacement directly as a constant (e.g., 0)
  - It has to be stored in a variable

Summary

- MPI I/O: MPI library is responsible for communication for parallel I/O access
  - File access coordinated through the file handle
- File views enable non-contiguous access patterns
- Collective I/O can enable the actual disk access to remain contiguous

Fortran interfaces for MPI I/O routines

```
mpi_file_set_view(fh, disp, etype, filetype, datarep, info)
   integer :: fh, disp, etype, filetype, info
   character* :: datarep
mpi_file_read_all(fh, buf, count, datatype, status)
mpi_file_read_at_all(fh, offset, buf, count, datatype, status)
mpi_file_write_all(fh, buf, count, datatype, status)
mpi_file_write_at_all(fh, offset, buf, count, datatype, status)
```

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)
```

Exercise session

- Complete any unfinished exercises of the course.

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)
```

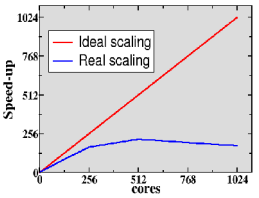
Fortran interfaces for MPI I/O routines

```
mpi_file_open(comm, filename, amode, info, fh, ierr)
   integer       ::  comm, amode, info, fh, ierr
   character* ::  filename
mpi_file_close(fh, ierr)
mpi_file_seek(fh, offset, whence)
integer :: fh, offset, whence
mpi_file_read(fh, buf, count, datatype, status)
   integer :: fh, buf, count, datatype, status(mpi_status_size)
mpi_file_read_at(fh, offset, buf, count, datatype, status)
   integer :: fh, offset, buf, count, datatype
   integer, dimension(mpi_status_size) :: status
mpi_file_write(fh, buf, count, datatype, status)
mpi_file_write_at(fh, offset, buf, count, datatype, status)
```

Performance considerations

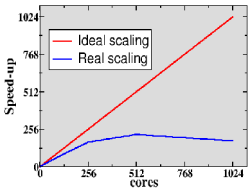
Why does scaling end?

- Amount of data per process small - computation takes little time compared to communication
- Load imbalance
- Communication that scales badly with N_{proc}
  - E.g., all-to-all collectives
- Congestion on network – too many messages or lots of data
- Amdahl's law in general
  - E.g., I/O



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



Performance measurement

- Most basic information: total wall clock time
  - Built-in timers in the program (e.g. MPI_Wtime)
  - System commands (e.g. time) or batch system statistics
- Built-in timers can provide also more fine-grained information
  - have to be inserted by hand
  - typically, no information about hardware related issues e.g. cache utilization
  - information about load imbalance and communication statistics of parallel program is difficult to obtain

Parallel computing concepts

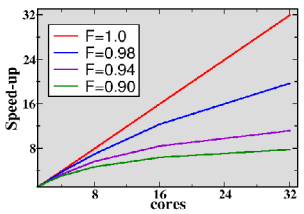
- Parallel programs contain often sequential parts
- *Amdahl's law* gives the maximum speed-up in the presence of non-parallelizable parts

Maximum speed-up:

$$\frac{1}{(1-F)+F/N}$$

F: parallel fraction

N: number of cores



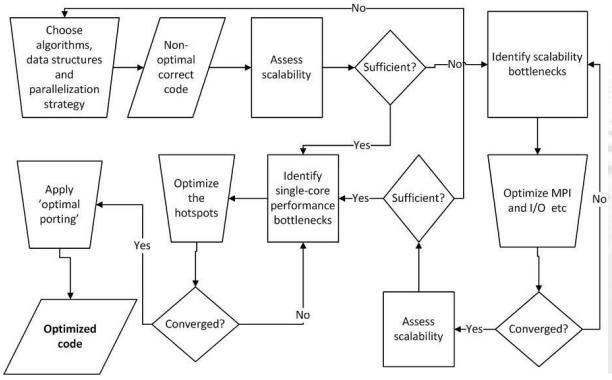
Performance measurement

- For more insight we need to employ *performance analysis tools*
  - Top time consuming routines (*profile*)
  - Load balance across processes and threads
  - Parallel overhead
  - Communication patterns
  - Hardware utilization details
- HPC platforms usually have performance analysis suites
  - CrayPAT, Scalasca, Paraver, Tau,...

Parallel computing concepts

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

Application optimization flow chart



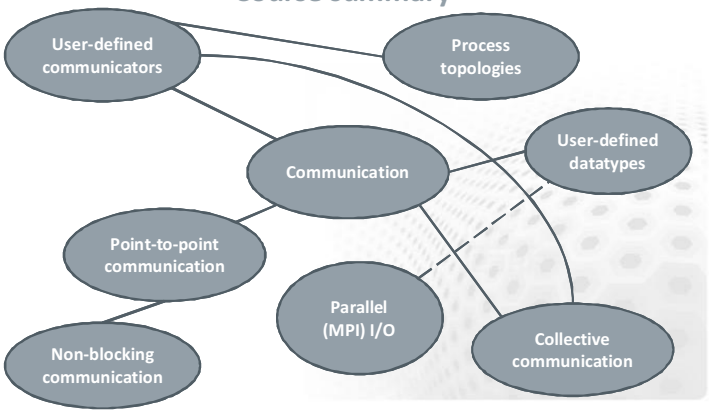
Efficient MPI programming style

- Use collectives!
  - If a collective call can do it for you, it will outperform all point-to-point constructs
- Avoid unnecessary memory copies
  - E.g. using array sections in Fortran
  - Derived datatypes are much faster
- Do not perform ordered set of sends or recvs
  - Employ a collective
  - Or point-to-point with `MPI_ANY_SOURCE` instead

Efficient MPI programming style

- Reduce latency: Send one big message instead of several small messages
  - Do not pack manually however, but use derived datatypes
- Do not ask for the stuff you do not need
  - Do not send dummy messages but use `MPI_PROC_NULL`
  - Do not request for a status if you don't employ it (but use `MPI_STATUS_IGNORE`)
- Mind the I/O
  - Without parallelization, I/O will become a bottleneck in all parallel applications -> use MPI I/O

Course summary



EXERCISE ASSIGNMENTS

Practicalities

Computing servers

We will use CSC's Cray supercomputer Louhi for the exercises. Log onto Louhi using the provided username and password, e.g.

```
ssh –X trng10@louhi.csc.fi
```

Alteratively, feel free to use the local workstations or your own Linux/Mac laptop and GNU compiler (you will also need a working MPI installation).

For editing Fortran program files you can use e.g. Emacs editor:

```
emacs test.f90
```

Also other popular editors (vim, nano) are available.

Compilation and running of MPI programs on Louhi

Compilation and execution of MPI programs can also be done via the **ftn** and **cc** wrappers and the **aprun** scheduler:

```
ftn test.f90 -o test
aprun –n 2 ./test
Hello world!
Hello world!
```

These are specific to Cray systems. If you are using the classroom workstation or your own laptop, you will most likely have to use **mpif90** or **mpicc** wrappers to compile MPI programs and **mpirun** command to launch them.

Point-to-point communication

1. Message chain

Write a simple program where every processor sends data to the next one. Let ntasks be the number of the tasks, and my_id the rank of the current process. Your program should work as follows:

- Every task with a rank less than ntasks-1 sends a message to task myid+1. For example, task 0 sends a message to task 1.
- The message content is an integer array, where each element is initialized to my_id
- The message tag is the receiver's id number.
- The sender prints out the number of elements it sends and the tag number.
- All tasks with rank ≥ 1 receive messages.
- Each receiver prints out their my_id, and the first element in the received array.

a. Implement the program described above using MPI_Send and MPI_Recv.

b. Extract from the status parameter how much data was received, and print out this information from all receivers.

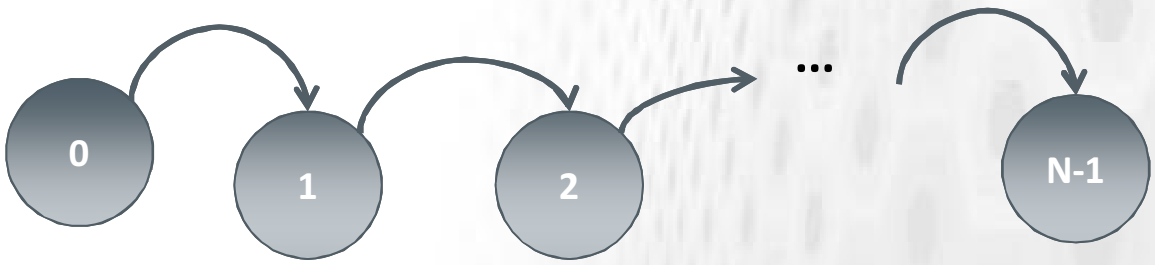
c. Use MPI_ANY_TAG when receiving. Print out the tag of the received message based on the status message.

d. Use MPI_ANY_SOURCE and use the status information to find out the sender. Print out this piece of information.

e. Can the code be simplified using MPI_PROC_NULL?

f. Use MPI_Sendrecv instead of MPI_Send and MPI_Recv.

Example solutions are given in files **ex1_msg-chain**(.c|.f90).



Collective operations & communicators

2. Matrix-vector multiplication

a) Implement a parallel matrix-vector multiplication **Ax**=**y**, first assuming that N (=length of the vectors) is dividable by the number of MPI processes. The most straightforward (and not the optimal algorithm) implementation is to distribute the matrix **A** and replicate **x** to each task. Use collective operations for distributing and replicating and for compiling the result **y** back to all processes.

   The approaches in C and Fortran will differ from each other because of the different layout of multi-dimensional arrays in memory.

b) Generalize the program in to allow for arbitrary N (hint: use the varying-size versions of the same collectives).

The answers are given in **ex2a_mxv** and **ex2b_mxv**.

3. Non-blocking message chain

Modify the program written in Exercise 1 to use non-blocking sends and receives (MPI_Isend and MPI_Irecv). Make sure to wait for the communication to finish before printing the results. An example solution is in the file **ex3_msg-chain-nonblock**.

4. Monte Carlo computation of π

In this exercise, the approximation for π is computed by generating random point pairs (x, y) in the square [-1,1]x[-1,1]. Then the value of π is obtained from the ratio of (points that fall into the unit circle)/(total number of points). We will implement this in a scheme, where one task will be the random number generator while the others determine whether the points are in the unit circle.

Insert into the skeleton code **ex4_mc_pi_0** a declaration for two communicators, *world* and *workers.* The *world* communicator is equal to `MPI_COMM_WORLD`, but the *workers* communicator will contain all the processes except the random number server (that is only one process). The solution is in **ex4_mc_pi**.

Advanced MPI

5. Cartesian grid

Write a test program where

- The processes are arranged into a 2D Cartesian grid
- Every task prints out their linear rank together with its coordinates in the process grid
- Every task prints out the linear rank of their nearest neighboring processes

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Run the code with both periodic and non-periodic boundaries and experiment with the direction and displacement parameters of the `MPI_Cart_shift` routine. The solution is given in **ex5_cart_test**(.f90|.c).

6. Message chain revisited

Starting from the ”message chain” of Exercises 1 & 2, implement a similar communication pattern but now the message should consist of

a) First row

b) First column

c) Diagonal elements

d) A submatrix consisting of the elements A(2:5,2:5)

of a 10x10 matrix with all elements initialized to the value of sender’s rank id. Use derived datatypes all the way. A solution is being provided in **ex6_matrix_msg**.

7. Game of Life

The *Game of Life* (GoL) is a cellular automaton devised by John Horton Conway in 1970, see **http://en.wikipedia.org/wiki/Conway's_Game_of_Life**. The game consists of two dimensional orthogonal grid of cells. Cells are in two possible states, *alive* or *dead*. Each cell is correlated with its eight neighbours, and at each time step the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation
- Any live cell with more than three live neighbours dies, as if by overcrowding
- Any live cell with two or three live neighbours lives on to the next generation
- Any dead cell with exactly three live neighbours becomes a live cell

Compile and run a serial reference implementation gol_ser(.f90|.c) and run the program of e.g. 500x500 board for 100 iterations. With the commands **`xview`** or **`eog`** you can view the images (.pbm) and see how the automaton looks like after those. You can also animate the board development by loading the ImageMagick module,

```
module load ImageMagick
```

and using the utilities found in it; first doing

```
convert -delay 30 -geometry 512x512 life_*.pbm life.gif
```

and then displaying the animation with

```
animate life.gif
```

a) Parallelize the GoL program with MPI, by dividing the board in columns and assigning one column to one task - a domain decomposition, that is. The tasks are able to update the board independently everywhere else than on the column boundaries - there the communication of a single column with the nearest neighbor is needed (the board is periodic, so the first column of the board is 'connected' to the last column). This is realized by having additional ghost layers on each of the local columns, that contain the boundary data of the neighboring tasks. The periodicity in the other direction is accounted as earlier. When printing out the board, all tasks send their local parts to one task that prints out the board. Insert the proper MPI routines into a skeleton code available at **ex7a_gol_mpi_0** (search for "TODO"s). A solution using MPI_Sendrecv in **ex7a_gol_mpi**. You can as well start from the serial version. Feel free to use other approaches to perform the halo exchange.

b) Decompose the GoL board in two dimensions by introducing a Cartesian process topology and rewriting the communication routines of the program to employ it. Create a new MPI datatype for sending and receiving rows (Fortran) or columns (C) of the board. A solution is in **ex7b_gol_2d**.

c) Add a feature to the GoL program that enables the user to start the program from a completed situation (i.e. not from scratch every time). This checkpointing will dump the situation of the whole board to disk every now and then; in a format that can be read in afterwards. Use MPI I/O to accomplish this. A solution is provided in **ex7c_gol_mpiio**. A starting point is provided in **ex7c_gol_mpiio_0**.

Notes