Large-Scale Parallel Computing

Prof. Dr. Felix Wolf

MESSAGE PASSING INTERFACE PART 3

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

- Message-passing model
- Basic MPI concepts
- Essential MPI functions
- Simple MPI programs
- Virtual topologies
- Point-to-point communication
- Datatypes
- Collective communication

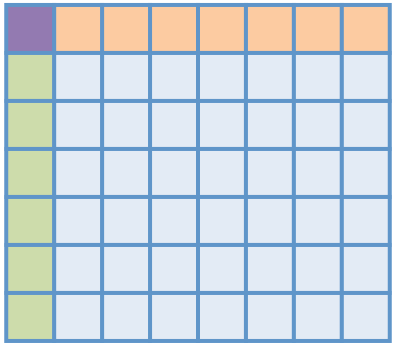
Non-contiguous data

Suppose we want to send column of a matrix stored row-wise

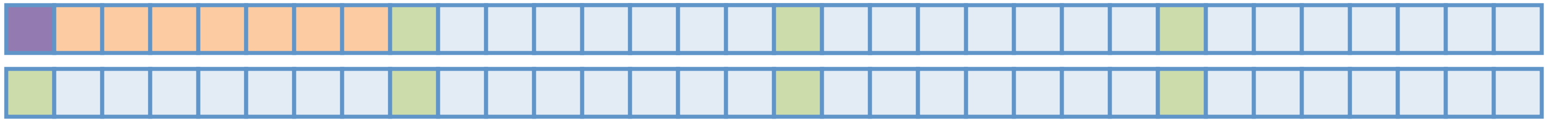
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Row-major and column-major order

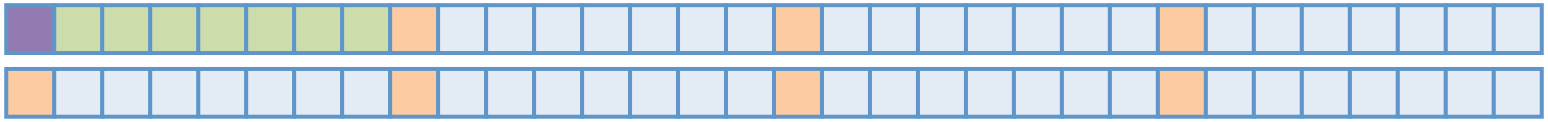
- Two-dimensional matrix



- Memory layout in C (row major)



- Memory layout in Fortran (column major)



Datatypes

- So far, communication involved only a contiguous sequence of identical datatypes
- Often, we need more flexibility
  - Mixing datatypes
    - Example: integer count followed by a sequence of real numbers
  - Non-contiguous data
    - Example: column of a matrix stored in row-major order or sub-block of a matrix
    - Possible solution – packing data into contiguous buffer
    - Disadvantage – local memory-to-memory copy operations at both sender and receiver
- MPI allows the direct transfer of objects of various shapes and sizes

General datatypes

TECHNISCHE UNIVERSITÄT DARMSTADT

- A general datatype is an opaque object that specifies:
  1. A sequence of basic datatypes = type signature

$$Typesig = \{type_0, \ldots, type_{n-1}\}$$

  2. A sequence of integer (byte) displacements
     - Neither required to be positive, distinct, nor in increasing order
- Together also called type map

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

- Together with base address specifies message buffer
  - *i-th* entry starts at $buf + disp_i$
  - Buffer will consist of n values of the types defined in type signature

Extent of a datatype

- The extent is the span from the first byte to the last byte occupied by its entries, rounded up to satisfy alignment requirements

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\} \Rightarrow$$

$$lb(Typemap) = \min_j (displ_j)$$

$$ub(Typemap) = \max_j (displ_j + sizeof(type_j)) + \varepsilon$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

Size of a datatype

The size of a datatype is the number of bytes the data takes up

$$size(Typemap) = \sum_j sizeof(type_j)$$

Example

$$\{(int, 0), (char, 4)\}$$

Assumption – integers to be aligned on 4-byte boundaries

$$lb = \min(0,4) = 0$$

$$ub = \max(0+4, 4+1) + 3 = 8$$

$$extent = 8 - 0 = 8$$

$$size = 4 + 1 = 5$$

Query functions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Size of a datatype

```
int MPI_Type_size(MPI_Datatype datatype,
                  MPI_Aint *size)
```

Integer type that can hold an arbitrary address

- Lower bound and extent of a datatype

```
int MPI_Type_get_extent(MPI_Datatype datatype,
                        MPI_Aint *lb,
                        MPI_Aint *extent)
```

  - The upper bound can be obtained by adding the extent to the lower bound

Derived datatypes

- A type map is a general way of describing an arbitrary datatype
  - May be inconvenient if the resulting type map contains a large number of entries
- MPI provides a number of ways to create datatypes without explicitly constructing the type map
  - Ranging from count copies of an existing datatype to a fully general description

Contiguous

- Datatype constructor that makes count copies of an existing one
- If we assume that the old datatype has type map

$$\{(int, 0), (double, 8)\}$$

- Then

```
MPI_Type_contiguous( 2, oldtype, newtype)
```

- Produces a new datatype with type map

$$\{(int, 0), (double, 8), (int, 16), (double, 24)\}$$

Using derived datatypes in communication


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

is exactly the same as

```
MPI_Type_contiguous( count, datatype, &newtype);
MPI_Type_commit( &newtype );
MPI_Send( buffer, 1, newtype, dest, tag, comm );
MPI_Type_free( &newtype );
```

Using derived datatypes in communication (2)

- A datatype object has to be committed before it can be used in a communication
- Uncommitted types can still be used in type constructors
- The system may „compile" at commit time an internal representation for the datatype that facilitates communication and select the most convenient transfer mechanism

Commit and free

- Commit a datatype

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

- Free a datatype

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Vector

- Created by replicating a datatype into locations that consist of equally spaced blocks
  - Each block is obtained by concatenating the same number of copies of the old datatype
  - The spacing between blocks (i.e., the stride) is a multiple of the extent of the old datatype

```
int MPI_Type_vector(
 int count,
 int blocklength,
 int stride,
 MPI_Datatype oldtype,
 MPI_Datatype *newtype)
```

- count = number of blocks
- blocklength = number of elements in each block
- stride = number of elements between the starts of adjacent blocks

Vector - example

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

```
MPI_Type_vector( 8, 2, 8, MPI_DOUBLE, newtype);
```

Indexed

Created by replicating a datatype into a sequence of blocks

- Like in a vector, each block is a concatenation of the old datatype
- However, each block can contain a different number of copies and can have a different displacement (multiple of the old type's extent)

```
int MPI_Type_indexed(int count,
                     int *array_of_blocklengths,
                     int *array_of_displacements,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

Indexed - example

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

```
int blocklengths[] =  {1,2,3,4,5,6,7,8};
int displacements[] =  {0,8,16,24,32,40,48,56};
MPI_Type_indexed( 8, blocklengths, displacements,
                  MPI_DOUBLE, newtype );
```

Struct

- Most general type constructor

```
int MPI_Type_create_struct(int count,
                 int      *array_of_blocklengths,
                 MPI_Aint *array_of_byte_displacements,
                 MPI_Datatype *array_of_types,
                 MPI_Datatype *newtype)
```

- Generalization of indexed constructor
  - Each block replicates different datatypes
  - Note that displacements are given in bytes (!)

Struct - example

- Let type1 have the type map

$$\{(double, 0), (char, 8)\}$$

- Let
  - b = {2, 1, 3}
  - d = {0, 16, 26}
  - t = {MPI_FLOAT, type1, MPI_CHAR}
- Then MPI_Type_struct(3, b, d, t, newtype) returns datatype with type map

$$\{(float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)\}$$

Further constructors

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hvector
  - Same as vector except that stride is given in bytes
- Hindexed
  - Same as indexed except that displacements are given in bytes
- Indexed block
  - Same as indexed except that the block length is the same for all blocks
- Subarray
  - n-dimensional subarray of an n-dimensional array
- Darray
  - Distributed array

Understanding extents

```
char *buffer;
MPI_Send( buffer, n, datatype, …);
```

sends the same data as

```
char *buffer;
MPI_Type_get_extent( datatype, &lb, &extent);
for ( i=0; i<n; i++)
  MPI_Send( buffer + (i * extent), 1, datatype, …);
```

- The extent of a datatype is not its size
- It is closest to being the stride of a datatype
  - From the start of one instance to the start of another instance in a contiguous type

Lower bound and upper bound markers

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sometimes convenient to explicitly define the lower and upper bound of a type map
  - Allows a datatype to be defined with holes at its beginning or end
  - Allows alignment rules used to compute lower and upper bounds to be overridden. Some compilers allow changing default alignment rules for some structures

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                            MPI_Aint lb, MPI_Aint extent,
                            MPI_Datatype *newtype)
```

– Creates a modified data type with new lower and upper bounds
– Does not affect the size of the datatype but its extent

Lower and upper bounds - example

```
MPI_Type_create_resized( MPI_INT, -4, 12, type1 );
```

Creates a datatype with type map

$$\{(lb\_marker, -4), (int, 0), (ub\_marker, 8)\}$$

- The markers lb_marker and ub_marker are conceptual datatypes that occupy no space
- The extent of the datatype is 12, its size is **`sizeof(int)`**

```
MPI_Type_contiguous( 2, type1, type2 );
```

Creates a datatype with type map

$$\{(lb\_marker, -4), (int, 0), (\mathrm{int}, 12), (ub\_marker, 20)\}$$

Summary datatypes

- Motivation
  - Non-contiguous data
  - Mixing of datatypes
- General datatype described by type map

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

- Size is number of bytes data takes up
- Extent is distance between first and last byte
- Derived datatypes can be composed recursively from potentially non-contiguous blocks of existing datatypes

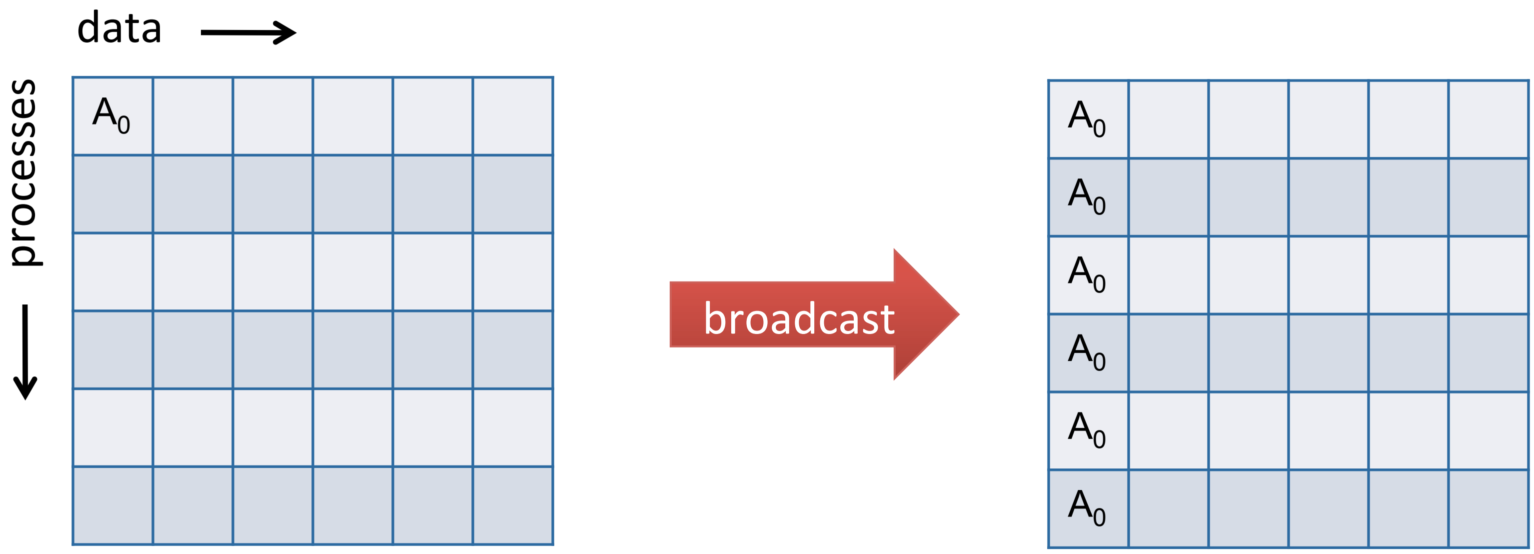
Collective operations

TECHNISCHE UNIVERSITÄT DARMSTADT

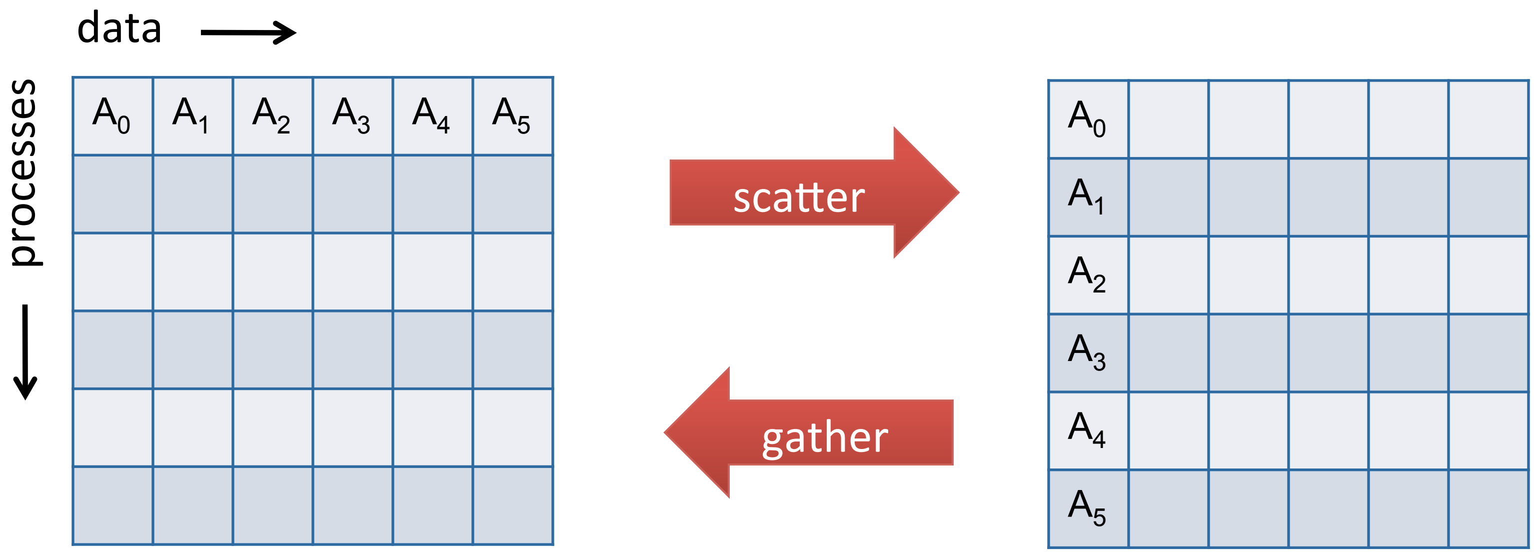
- Operation that involves a group of processes

Barrier synchronization across all members	MPI_Barrier
Broadcast from one member to all members	MPI_Bcast
Gather data from all members to one member / all members	MPI_Gather MPI_Gatherv MPI_Allgather MPI_Allgatherv
Scatter data from one member to all members	MPI_Scatter MPI_Scatterv
Complete exchange (scatter / gather) from all members to all members	MPI_Alltoall MPI_Alltoallv MPI_Alltoallw
Global reduction operations where the result is returned to one member / all members	MPI_Reduce MPI_Allreduce
Combined reduction and scatter	MPI_Reduce_scatter
Scan across all members of a group	MPI_Scan MPI_Exscan

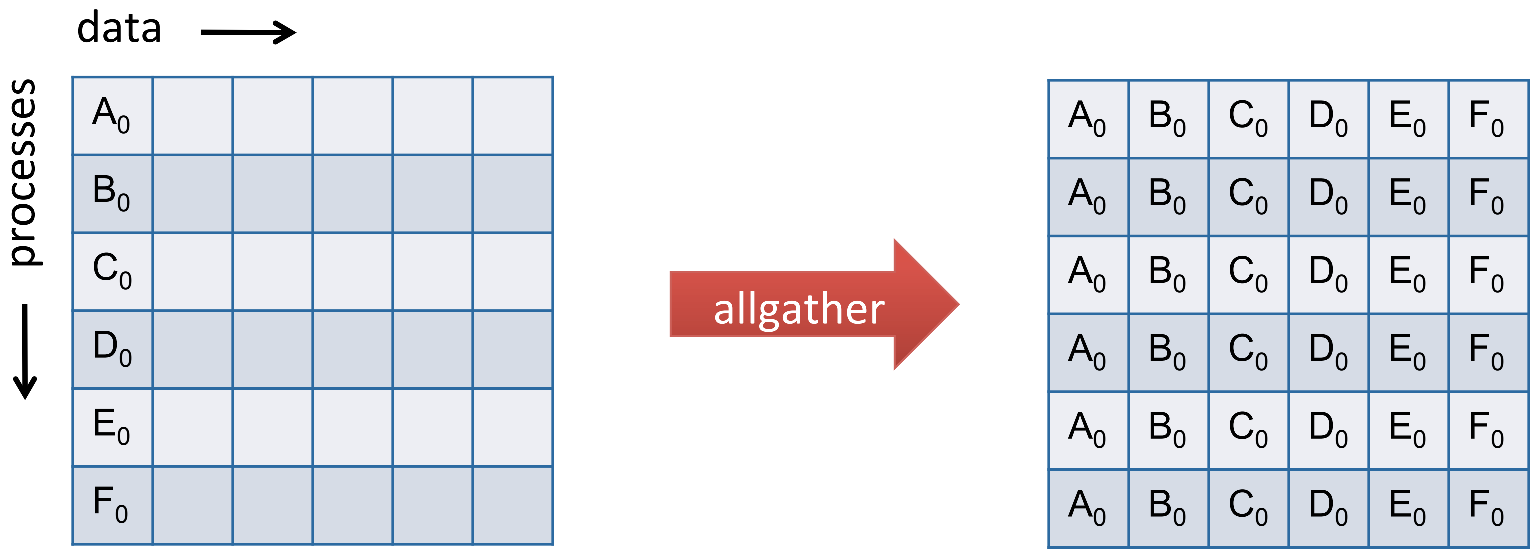
Broadcast



Scatter and gather



Allgather


data
processes
A0
B0
C0
D0
E0
F0
allgather
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0

Complete exchange

data →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

complete exchange

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Classification

All-to-all	
All processes contribute to the result and all processes receive the result	MPI_Allgather, MPI_Allgatherv MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw MPI_Allreduce, MPI_Reduce_scatter MPI_Barrier
All-to-one	
All processes contribute to the result and one process receives the result	MPI_Gather, MPI_Gatherv MPI_Reduce
One-to-all	
All processes receive the result	MPI_Bcast MPI_Scatter, MPI_Scatterv
Other	
Do not fit into one of the above categories	MPI_Scan, MPI_Exscan

Collective communication rules

- Several collective routines such as broadcast or gather have a single originating or receiving process
  - Such a process is called the root
  - Some arguments are significant only at root and are ignored by all others
- Type matching more strict than in the point-to-point case
  - The amount of data sent must exactly match the amount of data specified by the receiver
  - Different typemaps between sender and receiver still allowed
- Blocking
  - Collective calls may return as soon as their participation in the collective communication is complete
  - A collective call may or may not have the effect of synchronizing all calling processes

Collective communication rules (2)

- All processes in the communicator must call the collective routine
- Often, collective communication can occur “in place” with the output buffer being identical to the input buffer
  - Specified by providing MPI_IN_PLACE instead of send or receive buffer, depending on the operation performed

Barrier synchronization

```
int MPI_Barrier(MPI_Comm *comm)
```

- Blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call
- Applications
  - Correctness (e.g., ready send, I/O)
  - Performance (e.g., limiting the number of messages in transit)

Gather

Number of items to be received from each process – not the total number of items

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

- Each process sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order
  - Outcome is as if each of the n processes in the group executed a call to

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

and the root had executed n calls to

```
MPI_Recv(recvbuf + i*recvcount*extent(receivetype),
         recvcount, receivetype, i, …);
```

Gather (2)

- Type signatures of sendcount, sendtype on each process must be identical to recvcount, recvtype at the root

Gatherv

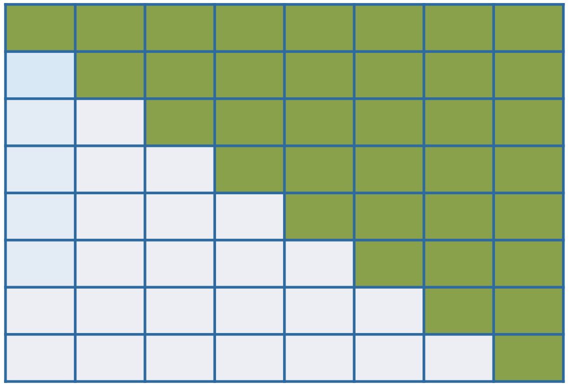
- Extends the functionality of gather by allowing
  - A varying count of data from each process
  - More flexibility as to where the data is placed on the root via displacements

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

- The data received from process j is placed into recvbuf of the root process beginning at displs[j] elements (in terms of the recvtype)

Example

- The root process gathers 8-i doubles from each rank i and places them into the rows of an 8x8 array, filling the upper triangular matrix



Example (2)

```
int myrank;
double recvcounts[] = {8,7,6,5,4,3,2,1};
double displs[] =     {0,9,18,27,36,45,54,63};

MPI_Comm_rank(comm, &myrank);
MPI_Gatherv(sendbuf,
            8-myrank,                 /* send count */
            MPI_DOUBLE,               /* send datatype */
            8x8_array,                /* receive buffer */
            recvcounts,               /* receive counts */
            displs,                   /* displacements */
            MPI_DOUBLE,               /* receive datatype */
            root,
            comm);
```

Scatter and scatterv

- Scatter is the inverse operation to gather

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

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

All-to-all exchange

- MPI_Allgather / MPI_Allgatherv
  - Can be thought of as a gather / gatherv, but where all processes receive the result, instead of just the root
- MPI_Alltoall / MPI_Alltoallv
  - An extension of allgather to the case where each process sends distinct data to each of the receivers
  - Alltoallv allows displacements to be specified for senders and receivers
- MPI_Alltoallw
  - Most general form of complete exchange
  - Allows separate specification of count, displacement, and datatype; displacement is specified in bytes

All-to-all exchange (2)

- MPI_Allreduce
  - Like a normal reduce, except that result appears in the receive buffer of all group members
- MPI_Reduce_scatter_block and MPI_Reduce_scatter
  - The result of the reduce operation is scattered to all group members

Reduction operations

TECHNISCHE UNIVERSITÄT DARMSTADT

- MPI_Reduce
- MPI_Allreduce
- MPI_Reduce_scatter_block
- MPI_Reduce_scatter
- MPI_Scan
- MPI_Exscan

Predefined reduction operations

TECHNISCHE UNIVERSITÄT DARMSTADT

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MINLOC and MAXLOC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Operators that can be used to compute an extreme value (min/max) and the rank of the process containing this value

MPI_MAXLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \max(u,v)$$

and

$$k = \begin{cases} i & \text{if} \quad u > v \\ \min(i,j) & \text{if} \quad u = v \\ j & \text{if} \quad u < v \end{cases}$$

MPI_MINLOC

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

where

$$w = \min(u,v)$$

and

$$k = \begin{cases} i & \text{if} \quad u < v \\ \min(i,j) & \text{if} \quad u = v \\ j & \text{if} \quad u > v \end{cases}$$

MINLOC and MAXLOC (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

- These reduction operations are defined to operate on arguments that consist of a pair
- The C binding of MPI provides suitable pair types

Name	Description
MPI_FLOAT_INT	float and integer
MPI_DOUBLE_INT	double and integer
MPI_LONG_INT	long and integer
MPI_2INT	pair of integers
MPI_SHORT_INT	short integer and integer
MPI_LONG_DOUBLE_INT	long double and integer

Example

- Each process has an array of 30 doubles. For each of the 30 entires, compute the value and rank of the process containing the largest value

```
/* each process has an array of 30 double: ain[30] */
double ain[30], aout[30];
int ind[30];
struct {
  double val;
  int rank;
} in[30], out[30];
int i, myrank, root = 0;
```

Example (2)

```
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
  in[i].val = ain[i];
  in[i].rank = myrank;
}
MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
/* At this point, the answer resides on process root */
if (myrank == root) {
  /* read ranks out */
  for (i=0; i<30; ++i) {
    aout[i] = out[i].val;
    ind[i] = out[i].rank;
  }
}
```

Scan

- Inclusive scan
  - Performs a prefix reduction on data distributed across the group

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

  - The operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of the processes with ranks 0,…,i (inclusive)
- Exclusive scan (same arguments)
  - For processes with rank i > 1, the operation returns, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0,…,i-1 (inclusive)

User-defined reduction operations

```
int MPI_Op_create(MPI_User_function *function,
                  int commute,
                  MPI_Op *op)
```

- Binds a user-defined reduction operation to an operation handle that can subsequently be used in a collective reduction call
  - The user-defined operation is assumed to be associative, allowing the order of the evaluation to be changed
  - If commute = true, then it is also assumed to be commutative
  - If commute = false, then the order of arguments is fixed and defined to be in ascending rank order
- Can be deleted using

```
int MPI_Op_free(MPI_Op *op)
```

User-defined reduction operations (2)

- The user function must have the following prototype

```
void MPI_User_function(void *invec, void *inoutvec,
                       int *len, MPI_Datatype *datatype)
```

  - We can think of invec and inoutvec as arrays of len elements that the function is combining
  - The results of the reduction overwrites values inoutvec
  - Each invocation of the function results in the element-wise evaluation of the reduction operator
  - inoutvec[i] = invec[i] ⊙ inoutvec[i]

Example: user-defined function

```
typedef struct {
  double real, imag;
} Complex;

void myProd( Complex *in, Complex *inout,
             int *len, MPI_Datatype *dptr ) {
  int i;
  Complex c;

  for (i=0; i< *len; ++i) {
    c.real = inout->real*in->real - inout->imag*in->imag;
    c.imag = inout->real*in->imag + inout->imag*in->real;
    *inout = c;
    in++; inout++;
  }
}
```

Example: how to use it

```
/* each process has an array of 100 Complexes */

Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

/* explain to MPI how type Complex is defined */
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
MPI_Type_commit( &ctype );

/* create the complex-product user-op */
MPI_Op_create( myProd, 1, &myOp );
MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

/* At this point, the answer, which consists of 100
   Complexes, resides on process root */
```

Summary collective operations

- Motivation – convenience and efficiency
- Classification
  - All-to-all, all-to-one, one-to-all, other
- Rules
  - Some operations have distinct root process
  - All processes of a communicator must call the operation
- Synchronization
  - Explicit synchronization via barrier
  - Some operations may involve implicit synchronization
- NOTE: So far, we covered only blocking operations

This is what we learned

- Message-passing model
- Point-to-point communication
- Virtual topologies
- Datatypes
- Blocking collective communication

Features not covered so far

- Intercommunicators
- Profiling interface
- Error handling
- Dynamic process management
- One-sided communication
- Multithreaded MPI applications
- File I/O
- Non-blocking collectives
- Neighborhood collectives