Algorithms and Programming IV
MPI Group Communication and MPI-2

Summer Term 2021 | 21.05.2021
Barry Linnert
Objectives of Today‘s Lecture

• Introduction to MPI group communication
• Introduction to MPI-2
Concepts of Non-sequential and Distributed Programming

MPI GROUP COMMUNICATION
Machine Model
Machine and Execution Model
MPI_Bcast

```c
int MPI_Bcast ( void *buffer,
               int count,
               MPI_Datatype datatype,
               int root,
               MPI_Comm comm);
```

- (Blocking) broadcast operation to send \( \text{root} == \text{my\_rank} \) or receive \( \text{root} == \text{my\_rank} \) a message to all participating processes of the communicator \text{comm}.
- The broadcast message must be received by all processes with \text{MPI\_Bcast()}.
- In case several messages were sent the sequence of these will be preserved.
MPI_Bcast

- buffer  – pointer to the buffer in which the message to be sent is located,
- count   – number of elements of type datatype to be sent,
- datatype – type of elements to be sent, all elements of a message must have the same type,
- root    – rank of the sending process,
- comm    – communicator that describes the group of processes that can exchange messages.
MPI_Reduce

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

- Merging the content of different messages in a global reduction operation (accumulation operation) `op` to be stored in a single value in `recvbuf`.
- All processes of the communicator `comm` have to send messages for the reduce operation.
- The root process `root` must provide the receive buffer `recvbuf`.
**MPI_Reduce**

- `sendbuf` – pointer to the buffer in which the message to be sent is located
- `recvbuf` – pointer to the buffer in which the messages are stored when received.
- `count` – number of elements of type `datatype` to be sent
- `datatype` – type of elements to be sent, all elements of a message must have the same type
- `op` – reduction operation
- `root` – rank of the receiving process
- `comm` – communicator that describes the group of processes that can exchange messages
## MPI_Reduce – Operations

<table>
<thead>
<tr>
<th>MPI_Reduce operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Returns the maximum element</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Returns the minimum element</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sums the elements</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Multiplies all elements</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Performs a <em>logical and</em> across the elements</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Performs a <em>bitwise and</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Performs a <em>logical or</em> across the elements</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Performs a <em>bitwise or</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Performs a <em>logical exclusive or</em> across the elements</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Performs a <em>bitwise exclusive or</em> across the bits of the elements</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Returns the max. value and the rank of the proc. that owns it</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Returns the min. value and the rank of the proc. that owns it</td>
</tr>
</tbody>
</table>
MPI_Op_create

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

- To define a special reduce operation to be used with MPI_Reduce().
MPI_Op_create

- function – pointer to the function to be applied to the corresponding operation,
- commute – specifies whether the operation is commutative (commute = 1) or not (commute = 0),
- op – data type of the operation to be applied.
## Extended MPI Datatypes

<table>
<thead>
<tr>
<th>MPI datentyp</th>
<th>Combination of C datentypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>(float, int)</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>(double, int)</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>(long, int)</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>(short, int)</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>(long double, int)</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>(int, int)</td>
</tr>
</tbody>
</table>
MPI_Type_create_struct

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

- Generation of a new structured data type as a combination of MPI basic types.
MPI_Type_create_struct

- count – number of elements in the following arrays,
- array_of_blocklengths
  – specifies the number of elements in each block,
- array_of_displacements
  – specifies the number of bytes to move each block,
- array_of_types
  – specifies the MPI type of the elements of each block,
- new_type – new data type.
MPI_Type_contiguous

```c
int MPI_Type_contiguous (  
    int count,  
    MPI_Datatype old_type),  
    MPI_Datatype *new_type);
```

- Creates a contiguous datatype out of an existing datatype.
MPI_Type_commit

int MPI_Type_commit(
    MPI_Datatype *data_type);

- Commits a datatype to be used by the MPI environment.
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
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++;
    }

    // At this point, the answer, which consists of 100 Complexes, resides on root
    // source code: 12-00.c

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
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++;
    }

    // and, to call it...
...

    // each proc has an array of 100 Complex.
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    // define type Complex for MPI
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype);

    // At this point, the answer, which
    // consists of 100 Complexes,
    // resides on root source code: 12-00.c
typedef struct {
  double real, imag;
} Complex;

// the user-defined function
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++;  
  }
}

// and, to call it...

...  

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE,
                      &ctype );

MPI_Type_commit( &ctype );

// At this point, the answer, which
// consists of 100 Complexes,
// resides on root source code: 12-00.c
```c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
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++;
    }
}
```

// and, to call it...
...

// each proc has an array of 100 Complexes
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

// At this point, the answer, which consists of 100 Complexes, resides on root 
source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
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++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complexes
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE,
                      &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

MPI_Reduce( a, answer, 100, ctype,
            myOp, root, comm );

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

source code: 12-00.c
typedef struct {
    double real, imag;
} Complex;

// the user-defined function
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++;
    }
}

// and, to call it...
...

// each proc has an array of 100 Complex.
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;

// define type Complex for MPI
MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
MPI_Type_commit( &ctype );

// create the complex-product user-op
MPI_Op_create( myProd, True, &myOp );

MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

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

source code: 12-00.c
**MPI_Gather**

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

− Gathers together values from a group of processes of the communicator `comm` (without reduction operation).
− The elements are stored in order of the numbers of the processes involved.
MPI_Gather

- `sendbuf` – pointer to the buffer in which the message to be sent is located,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the messages are stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `root` – rank of the receiving process,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Gatherv

```c
int MPI_Gatherv ( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, const int recvcounts[],
                  const int displs[],
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm);
```

- Gathers together values out of messages with different size from a group of processes of the communicator `comm`.
- The root process specifies the number (`recvcounts`) and position of the storage within the receive buffer (`displs`).
MPI_Scatter

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

- Distribution of individual data (same size) by messages to all the processes of the communicator comm.
- The data is distributed and sent according to the numbers of the target processes.
MPI_Scatter

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `root` – rank of the receiving process,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Scatterv

```c
int MPI_Scatterv ( void *sendbuf,
                    const int sendcounts[],
                    const int displs[],
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int recvcount,
                    MPI_Datatype recvtype,
                    int root,
                    MPI_Comm comm);
```

- Distribution of individual data of different sizes by messages to all the processes of the communicator `comm`. 
**MPI_Allgather**

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

- Gathers together data from messages from all processes of the communicator `comm` and distribute it to all involved processes.
- The elements are stored in the order of the numbers of the processes involved.
MPI_Allgather

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `sendcount` – number of elements of type `datatype` to be sent,
- `sendtype` – type of elements to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `recvcount` – number of elements of type `datatype` to be received,
- `recvtype` – type of elements to be sent, all elements of a message must have the same type,
- `comm` – communicator that describes the group of processes that can exchange messages.
MPI_Allgatherv

```c
int MPI_Allgatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm);
```

- Gathering of individual messages from all processes of the communicator `comm` (without reduction operation) and distribution to all involved processes.
MPI_Allreduce

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

- Merging the content of different messages in a global reduction operation (accumulation operation) \( op \) and distributes the result back to all processes of the communicator \( \text{comm} \).
### MPI_Allreduce

- `sendbuf` – pointer to the buffer containing the data to be sent,
- `recvbuf` – pointer to the buffer in which the message is stored when received,
- `count` – number of elements of type `datatype` to be received,
- `datatype` – type of elements to be sent, all elements of a message must have the same type,
- `op` – reduction operation,
- `comm` – communicator that describes the group of processes that can exchange messages.
// part of simple MPI program with MPI_Allreduce
// to multiply a matrix and a vector

int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];

local_m = m / p;
for (i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j < local_m; j++)
        sum[i] = sum[i] + a[i][j] * local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);

source code: 12-01.c
MPI_Alltoall

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

- Total exchange of (individual) messages of equal size of all processes of the communicator `comm` (without reduction operation).
MPI_Alltoall

- **sendbuf** – pointer to the buffer containing the data to be sent,
- **sendcount** – number of elements of type datatype to be sent,
- **sendtype** – type of elements to be sent,
- **recvbuf** – pointer to the buffer in which the message is stored when received,
- **recvcount** – number of elements of type datatype to be received,
- **recvtype** – type of elements to be sent, all elements of a message must have the same type,
- **comm** – communicator that describes the group of processes that can exchange messages.
MPI_Alltoallv

int MPI_Alltoallv ( void *sendbuf,
        const int sendcounts[],
        const int sdispls[],
        MPI_Datatype sendtype,
        void *recvbuf,
        const int recvcounts[],
        const int rdispls[],
        MPI_Datatype recvtype,
        MPI_Comm comm);

- Total exchange of (individual) messages with different size of all processes of the communicator comm (without reduction operation).
MPI_Comm_group

```c
int MPI_Comm_group (MPI_Comm comm,
                     MPI_Group *group);
```

- Returns all processes assigned to the communicator `comm` in the group data structure `group`. 
MPI_Group_union

```c
int MPI_Group_union ( MPI_Group group1,
                      MPI_Group group2,
                      MPI_Group *new_group);
```

- Merges the processes of the groups `group1` and `group2` into a group `new_group`.
MPI_Group_intersection

```c
int MPI_Group_intersection ( MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group);
```

- Produces a new group `new_group` as the intersection of the processes of the groups `group1` and `group2`. 
MPI_Group_difference

```c
int MPI_Group_difference ( MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group);
```

- Makes a new group `new_group` from the difference of the groups `group1` and `group2`.
MPI_Group_incl

int MPI_Group_incl (MPI_Group group,
                     int p,
                     const int ranks[],
                     MPI_Group *new_group);

- Creates a new group new_group from the processes of an existing group
  group by taking only the p processes listed in ranks.
**MPI_Group_excl**

```c
int MPI_Group_excl (MPI_Group group,
                   int p,
                   const int ranks[],
                   MPI_Group *new_group);
```

- Creates a new group `new_group` from the processes of an existing group `group` by not adopting the `p` processes listed in `ranks`. 
MPI_Group_compare

```c
int MPI_Group_compare ( MPI_Group group1,
                        MPI_Group group2,
                        int *res);
```

- Compares two groups and stores the result in `res`.
- For groups with the same processes in the same order `MPI_IDENT` is returned, for groups with the same processes in different orders `MPI_SIMILAR` and for different groups `MPI_UNEQUAL`.
MPI_Group_free

```c
int MPI_Group_free (MPI_Group *group);
```

- Releases the data structure holding the group `group`. 
MPI_Comm_create

`int MPI_Comm_create (MPI_Comm comm, MPI_Group group, MPI_Comm new_comm);`

- Creates a new communicator `new_comm`, which addresses the processes of the `group` `group`, from the existing communicator `comm`. 
MPI_Comm_compare

int MPI_Comm_compare ( MPI_Comm comm1,
                      MPI_Comm comm2,
                      int *res);

- Compares the two communicators \texttt{comm1} and \texttt{comm2} and stores the result in \texttt{res}.

- If \texttt{comm1} and \texttt{comm2} point to the same data structure, \texttt{MPI_IDENT} is returned. If there are different communicators with the same processes in the same order \texttt{MPI_CONGRUENT} is returned, if there are communicators with the same processes in different orders \texttt{MPI_SIMILAR} and if there are different communicators \texttt{MPI_UNEQUAL} is returned.
MPI_Comm_dup

```c
int MPI_Comm_dup ( MPI_Comm comm,
                  MPI_Comm new_comm);
```

- Creates a new communicator `new_comm` with the same processes in the same order as the communicator `comm`. 
MPI_Comm_split

int MPI_Comm_split (MPI_Comm comm,
                   int color,
                   int key,
                   MPI_Comm *new_comm);

- Split the processes of the communicator comm according to the values color in the order key and return the communicator in which the corresponding process is found.
- If a process has not set the value color, it will not be found in any of the created communicators.
MPI_Comm_free

```c
int MPI_Comm_free ( MPI_Comm *comm);
```

- Releases the communicator `comm` after all message transmissions performed with this communicator have been completed.
MPI_Wtime

double MPI_Wtime (void);

- Returns the time in seconds after a certain time. The elapsed processing time can be determined from the difference between second calls.
Concepts of Non-sequential and Distributed Programming

MPI-2
MPI-2

- MPI-2 is an extension of the MPI standard.
- It introduces new functions to support
  - memory transfers,
  - dynamic process management,
  - input/output operations.
- In particular, applications with very high resource requirements will be programmed with dynamic process generation and dynamic runtime behavior.
**MPI_Comm_spawn**

```c
int MPI_Comm_spawn (const char *command,
                    char *argv[],
                    int maxprocs,
                    MPI_Info info,
                    int root,
                    MPI_Comm comm,
                    MPI_Comm *intercomm,
                    int array_of_errcodes[]);
```

- Creates a number of `maxprocs` new MPI processes that execute the `command` program. The child processes still have to call `MPI_Init()` for execution.
MPI_Comm_spawn

The parameters for MPI_Comm_spawn are:

- **command** – program executed by the child processes,
- **argv** – arguments passed to the child processes,
- **maxprocs** – number of child processes to be created,
- **info** – process information or MPI_INFO_NULL, to transfer the administration to the runtime system,
- **root** – rank of the parent process,
- **comm** – communicator of the parent process,
- **intercomm** – communicator, the group of child processes,
- **array_of_errcodes** – error code per child process.
**MPI\_Comm\_get\_parent**

```c
int MPI_Comm_get_parent ( MPI_Comm *parent);
```

- Returns the communicator of the parent process in `parent`. 
**MPI_Comm_spawn_multiple**

```c
int MPI_Comm_spawn_multiple ( int count,
const char *commands[],
char **argv[],
int maxprocs[],
MPI_Info infos[],
int root,
MPI_Comm comm,
MPI_Comm *intercomm,
int array_of_errcodes[]);
```

- Starts `count` many command programs, each with `maxprocs` many processes.
- It creates one communicator for all child processes.
MPI_Comm_spawn_multiple

- count – number of programs to be started,
- command – program executed by the child processes,
- argv – arguments passed to the child processes,
- maxprocs – number of child processes to be created,
- info – process information or MPI_INFO_NULL, to transfer the administration to the runtime system,
- root – rank of the parent process,
- comm – communicator of the parent process,
- intercomm – communicator, the group of child processes,
- array_of_errcodes – error code per child process.
Concepts of Non-sequential and Distributed Programming

NEXT LECTURE
Design and Implementation of Parallel Applications

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2021)