Week 3: MPI

Day 03 :: Data types, groups, topologies, error handling, parallel debugging
Other MPI-1 features
Timing

- REAL(8) :: t
  t = MPI_Wtime()  -- not a sub, no ierror!

- Get the current wall-clock time in seconds

- REAL(8) :: tick
  tick = MPI_Wtick() -- not a sub, no ierror!

- Get the resolution of the clock
- If clock is implemented in milliseconds, tick returns 0.001
- Only time differences bigger than this resolution are significant
Emergency brake

- `INTEGER :: comm,err,ierror`
- `MPI_Abort(comm,err,ierror)`
- Terminates all processes in communicator `comm` and returns error code `err` to the operating system
Derived data types
Why and what?

- MPI allows the user to define new data types that can be derived from basic MPI data types or other derived data types

- Useful to:
  - Make code more readable
  - Reduce number of messages and increase their size (faster!)
  - Make code more efficient if messages of the same size/type are repeatedly used (cache!)
Different classes of data types

- **contiguous**: all elements are of the same sub-type and linearly contiguous in memory
- **non-contiguous**: all elements are of the same sub-type and uniformly spaced in memory
- **indexed**: all elements are of the same sub-type and arbitrarily scattered in memory
- **structures**: different elements can be of different sub-types and arbitrarily scattered in memory
Defining a contiguous data type

```fortran
INTEGER :: count, oldtype, newtype, ierror
CALL MPI_Type_contiguous(count, oldtype, &
newtype, ierror)
```

- Defines new data type (handle `newtype`) from `count` contiguous repetitions of `oldtype` (handle or basic MPI data type).
- Used as building blocks for more complex types
Defining a non-contiguous data type

```
INTEGER :: count, oldtype, newtype, ierror
INTEGER :: blocklength, stride
CALL MPI_Type_vector(count, blocklength, &
                     stride, oldtype, newtype, ierror)
```

- Defines new data type (handle `newtype`) from `count` blocks of `blocklength` items of type `oldtype` (handle or basic MPI data type) using a stride of `stride` items.

- Plus many other MPI type construction functions: `MPI_Type_struct`, `MPI_Type_indexed`, ...
Committing and freeing data types

- New MPI data types must be committed before they can be used:

  ```fortran
  INTEGER :: newtype, ierror
  CALL MPI_Type_commit(newtype, ierror)
  ```

- Types that are not needed any more can be freed:

  ```fortran
  INTEGER :: type, ierror
  CALL MPI_Type_free(type, ierror)
  ```
New data types

This:

CALL MPI_Send(buf, count, datatype, dest, tag, & comm, ierror)

Is equivalent to this:

CALL MPI_Type_contiguous(count, datatype, & newtype, ierror)
CALL MPI_Type_commit(newtype, ierror)
CALL MPI_Send(buf, 1, newtype, dest, tag, & comm, ierror)

Defining a new type is more efficient if messages of the same length are frequently used!
Packing and unpacking data

Fewer and larger messages are better than many small messages (latency!).

- **MPI_Pack** provides an alternative to user-defined data types
- Buffers messages in user memory rather than in system memory
- More efficient than types if used only few times
Packing data into a buffer

- \textbf{packdata} is the data to be copied into buffer
- \textbf{datatype} is the data type of the elements
- \textbf{buffer} is the user-allocated target buffer
- \textbf{pos} is the position where to start writing to buffer
Packing data into a buffer

MPI_Pack

Pos (write will start here)

Pos (first empty cell after copied data)
Sending/receiving packed data

CALL MPI_Send(buffer,buf_size,MPI_PACKED, & dest,tag,comm,ierror)
CALL MPI_Recv(buffer,buf_size,MPI_PACKED, & src,tag,comm,status,ierror)

- Use data type argument MPI_PACKED (endian conversions are only done upon unpacking and not already upon receiving)
Unpacking data from a buffer

```fortran
INTEGER :: datatype, count, buf_size, pos
<datatype>, DIMENSION(count) :: unpackdata
<datatype>, DIMENSION(buf_size) :: buffer

CALL MPI_Unpack(buffer, buf_size, pos, &
                 unpackdata, count, datatype, comm, ierror)
```

- **buffer** is the buffer from which to read data
- **pos** is the position in buffer where to start read
- **unpackdata** is the user-allocated target
- **datatype** is the MPI data type of the elements
Unpacking data from a buffer

`unpackdata` buffer

`MPI_Unpack`

`unpackdata` buffer

`Pos` (first element to read)

`Pos` (first empty cell after read data)
Process groups & Communicators
Process groups and communicators

- MPI provides routines for:
  - defining new process groups from subsets of existing communicators like `MPI_COMM_WORLD`
  - creating new communicators for new process groups
  - performing collective communication within a process group
  - defining virtual topologies of the processes in a communicator
- For library writers who have no control over MPI calls in the user code
- For localizing communication to sub-groups of processors on a very large machine
- Dynamic process exclusion (but they still occupy CPU!)
Process groups

An MPI Group is:

- an ordered set of processes
- each process in a group has a unique integer ID (its rank) **within that group**
- a process can belong to more than one group (but the rank is always relative to the group!)
- groups are “opaque objects”. Therefore, use only MPI-provided routines for manipulating groups via their **handle**.
Communicators

- A communicator is a **group** plus a **context** (unique ID of the communicator; integer handle)
- Context is created and associated with a group when a communicator is created.
- A communicator is an MPI “communication universe” \(\rightarrow\) sender and receiver in ANY communication must specify the same communicator
- Communicators are “opaque objects”
- Groups and communicators are dynamic objects that can be created and destroyed during the execution of the program.
Process groups and communicators

Process group
Context (global, unique)
Communicator
Process groups and communicators

Only the tuple (comm, rank) uniquely identifies a process!
Defining a new communicator

Proceeds in 3 steps:

- Extract the group handle of an existing “base communicator” (local)
- Include or exclude processes from that group into a new group (local)
- Let the system define a new context for the new group and create the new communicator (collective communication!)
Extract the group handle of a communicator

```
INTEGER :: group, comm, ierror
CALL MPI_Comm_group(comm, group, ierror)
```

- **comm** is the communicator (input)
- **group** returns the handle of the underlying process group (output)
Extract processes into a new group

INTEGER :: old_grp, size, new_grp, ierror
INTEGER, DIMENSION(size) :: members

CALL MPI_Group_incl(old_grp, size, members, &
                    new_grp, ierror)

- **old_grp** is the old (basis) group (input)
- **size** is the number of processes to be included in the new group (input)
- **members** are the ranks (in **old_grp**) of the processes to be included in **new_grp** (input)
- **new_grp** is the handle to the new group (output), where the processes have new ranks 0,...,**size**-1.
Define a new communicator

```fortran
INTEGER :: old_comm,new_grp,new_comm,ierror

CALL MPI_Comm_create(old_comm,new_grp,new_comm,ierror)
```

- Determines a context and associates it with the group `new_grp`. Creates the new communicator `new_comm` comprising all processes in `new_grp` underlying the current communicator `old_comm`.

- This is a **collective communication operation** (determine global unique context!) → all processes in `old_comm` must call the same `MPI_Comm_create`. 
Typical use

1. Extract handle of global group from `MPI_COMM_WORLD` using `MPI_Comm_group`
2. Form new group as a subset of global group using `MPI_Group_incl` or `MPI_Group_excl`
3. Create new communicator for new group using `MPI_Comm_create`
4. Determine new rank in new communicator using `MPI_Comm_rank`
5. Conduct communications using any MPI message passing routine
6. When finished, free up new communicator and group (optional) using `MPI_Comm_free` and `MPI_Group_free`
Creating several communicators at once

- **MPI_Comm_split** can create several communicators at once.
- It’s a collective communication operation.
- Each process calls it with some value of *color*.
- A rank key determines the ordering of the processes in the new communicator. IF \( \text{rank}_\text{key}(a) < \text{rank}_\text{key}(b) \) in the old communicator, then also \( \text{rank}(a) < \text{rank}(b) \) in the new one.
- Each process only gets back the handle of the communicator to which it belongs.
Creating several communicators at once

INTEGER :: old_comm,color,rank_key
INTEGER :: new_comm,ierror

CALL MPI_Comm_split(old_comm,color,rank_key &
                     new_comm,ierror)

- **old_comm** is the old communicator
- All processes that call the function with the same value of **color** will end up in the same new comm.
- **rank_key** determines the rank order in **new_comm**
- **new_comm** is the new communicator to which the present process belongs new (with a new rank!)
Excluding processes from the new comm.

```fortran
INTEGER :: old_comm, rank_key, new_comm, ierror

CALL MPI_Comm_split(old_comm, MPI_UNDEFINED, &
    rank_key, new_comm, ierror)

! new_comm will be MPI_COMM_NULL

§ All processes calling MPI_Comm_split with a color value of MPI_UNDEFINED will not be included in any new communicator
§ new_comm will contain the value MPI_COMM_NULL
§ They are still included in old_comm
```
Virtual Topologies
What are MPI topologies?

- Additional information (besides group and context) can be associated (cached) with any MPI communicator.
- MPI uses these attributes to define topologies.
- (virtual) topologies are alternative addressing schemes for the processes in a communicator.
- They are virtual topologies because they don’t correlate with the underlying hardware network topology.
Types of MPI topologies

- Cartesian grid
  - Processes are laid out on a Cartesian grid of arbitrary dimension

- Graph
  - Superset of Cartesian grid topologies
  - Processes can have arbitrary neighborhood relations as given by the edges in the graph
  - Not discussed in this course → see literature.
Virtual Cartesian process topologies

- Arranges the processes in a $n$-dimensional Cartesian grid
- Allows addressing them by their grid indices
- Indices start at 0 and use row-major layout
Most important MPI topology functions

- **MPI_Cart_create** creates a Cartesian process topology and associates it with a communicator (*collective communication op!*).

- **MPI_Cart_rank** returns the rank of a process given its grid coordinates *(local)*.

- **MPI_Cart_coords** returns the grid coordinates of a process given its rank *(local)*.

- **MPI_Cart_shift** can be used to determine the neighbors for a communication operation *(local).*.

- **MPI_Cart_get** inquires about a topology *(local).*.
Creating a new Cartesian topology

```fortran
INTEGER :: old_comm, ndim, cart_comm, ierror
LOGICAL :: reorder
INTEGER, DIMENSION(ndim) :: dim_sizes
LOGICAL, DIMENSION(ndim) :: wrap
CALL MPI_Cart_create(old_comm, ndim, dim_sizes, wrap, reorder, cart_comm, ierror)
```

- `old_comm` is the old communicator (in)
- `ndim` is the number of space dimensions (in)
- `dim_sizes` is the number of processes to be used in each dimension (in)
- `wrap` has true values for periodic dimensions (in)
- `reorder` is true to allow process reordering (in)
- `cart_comm` is the new Cartesian communicator (out)
Reorder and wrap

- If `reorder.EQ..TRUE..`, MPI is allowed to reorder (change the ranks of) the processes in order to optimize their mapping onto the physical processors (cores).
- If `wrap.EQ..TRUE..` in any dimension, boundary processes are neighbors. E.g.: if `wrap(1)=.TRUE..`, then ranks 0 and 6 are neighbors in the grid. If also `wrap(2)=.TRUE..`, 0 is also a neighbor of 2.
Getting the rank of a process

- `cart_comm` is the Cartesian communicator (in)
- `coords` are the Cartesian grid coordinates of the process (in)
- `rank` is the linear rank (in `cart_comm`) of the process (out)
Getting the Cartesian coordinates of a process

```plaintext
INTEGER :: cart_comm, rank, ierror
INTEGER, DIMENSION(ndim) :: coords

CALL MPI_Cart_coords(cart_comm, rank, ndim, &
                     coords, ierror)
```

- **cart_comm** is the Cartesian communicator (in)
- **rank** is the linear rank (in **cart_comm**) of the process (in)
- **ndim** are the number of space dimensions (in)
- **coords** are the Cartesian grid coordinates of the process (out)
Exchanging data with the neighbors

- Frequently, data needs to be exchanged with the neighbor processes (e.g. ghost layer update)
- Given a Cartesian topology, MPI provides a function to determine (on each process) the source and destination ranks to used in a subsequent `MPI_SendRecv`
- If boundaries are not periodic, source and destination ranks can be `MPI_PROC_NULL`

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>1</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>2</td>
<td>(0,2)</td>
<td>(0,3)</td>
<td>(0,4)</td>
</tr>
<tr>
<td>3</td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>4</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>5</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
</tr>
<tr>
<td>6</td>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
<tr>
<td>7</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
</tr>
<tr>
<td>8</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
</tr>
</tbody>
</table>
Shifting data in the grid

```
INTEGER :: cart_comm, dir, shift, src, dest
INTEGER :: ierror

CALL MPI_Cart_shift(cart_comm, dir, shift, src, dest, ierror)
```

- `cart_comm` is the Cartesian communicator (in)
- `dir` is the direction (dimension) of the shift (in)
- `shift` is the shift stride, positive values indicate directions of increasing coord. 1 for neighbors (in)
- `src, dest` are the ranks to be used in `MPI_SendRecv` (out)
Example

<table>
<thead>
<tr>
<th>dim=1 (first is 0!)</th>
<th>shift=4</th>
<th>dim=1</th>
<th>shift=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>jloc</td>
<td>kloc</td>
<td>jloc</td>
<td>kloc</td>
</tr>
<tr>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
<tr>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,1)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,1)</td>
<td>(2,2)</td>
</tr>
</tbody>
</table>
Inquiring about a topology

```
INTEGER :: cart_comm, maxd, ierror
INTEGER, DIMENSION(maxd) :: coords, dims
LOGICAL, DIMENSION(maxd) :: wrap

CALL MPI_Cart_get(cart_comm, maxd, dims, &
                   wrap, coords, ierror)
```

- `cart_comm` is the Cartesian communicator (in)
- `maxd` is the maximum number of dimensions for which the arguments were allocated (in)
- `dims` returns the number of processes in each dimension (out)
- `wrap` returns for each dimension if it’s periodic (out)
- `coords` returns the Cartesian coordinates of the calling process (out)
Slicing and sectioning of grids

- Grids can be partitioned into grids of lower dim.
- This creates several new (disjoint!) communicators at once, whose union is the old communicator.
- Useful to, e.g., generate communicators for the rows and columns of a matrix.
- New communicators are created by only retaining dimensions that are explicitly included.
- Is a collective communication operation and must be called by all processes in the communicator synchronously.
Slicing a Cartesian topology

INTEGER :: cart_comm,new_comm,ierror
LOGICAL, DIMENSION(ndim) :: keep_dim

CALL MPI_Cart_sub(cart_comm,keep_dim,  &
                   new_comm,ierror)

- **cart_comm** is the Cartesian communicator (in)
- **keep_dim** is .TRUE. for all dimensions that should be included in the sub grids (in)
- **new_comm** returns the new communicator on each calling process. Also has Cartesian topology (of dim #keep_dim) associated! (out)
MPI error handling
How are MPI errors handled?

- Each communicator has an error handler associated with it.
- When an MPI error occurs, MPI calls that error handler.
- The default error handler (do nothing) is set in `MPI_Init`.
- User can change the error handler and write own error handlers as well.
Most important MPI error handling functions

- `MPI_Errhandler_set` sets an error handler
- `MPI_Errhandler_get` gets an error handler
- `MPI_Errhandler_free` frees a no longer used error handler
- `MPI_Errhandler_create` creates a new, user-defined error handler
- `MPI_Error_class` determines the class of an error
- `MPI_Error_string` determines the symbolic error message of an error
Setting the error handler

```
INTEGER :: comm, handler, ierror
CALL MPI_Errhandler_set(comm, handler, ierror)
```

- **comm** is the communicator (in)
- **handler** is the error handler (in). Either a predefined one: `MPI_ERRORS ARE_FATAL, MPI_ERRORS RETURN` (ierror of that routine contains the error code) or a user-defined subroutine.
Inquiring the error handler

```
INTEGER :: comm, handler, ierror
CALL MPI_Errhandler_get(comm, handler, ierror)
```

- **comm** is the communicator (in)
- **handler** is the error handler (out). Either one of **MPI_ERRORS_ARE_FATAL** or **MPI_ERRORS_RETURN**, or a user-defined error handler.
Check the class of an error

```fortran
INTEGER :: code, class, ierror

CALL MPI_Error_class(code, class, ierror)
```

- `code` is the (implementation specific!) MPI error code returned in `ierror` of a failed subroutine (in).
- `class` is the (standardized) error class (out)
Most important MPI error classes

- Identified by handles via `mpif.h`
- Many more error classes exist!

<table>
<thead>
<tr>
<th>MPI error</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>No error</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>Buffer too small</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>Invalid communicator</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>Invalid rank</td>
</tr>
<tr>
<td>MPI_ERR_INTERN</td>
<td>Internal MPI error</td>
</tr>
</tbody>
</table>
Getting a symbolic error message

```plaintext
INTEGER :: code, len, ierror
CHARACTER(LEN=MPI_MAX_ERROR_STRING) :: string

CALL MPI_Error_string(code, string, len, ierror)
```

- **code** is the (implementation specific!) MPI error code returned in **ierror** of a failed subroutine (in).
- **string** is the symbolic error string (out)
- **len** is the length (in characters) of the returned error string (out)
Creating a user-defined error handler

INTEGER :: handler, ierror
EXTERNAL :: Errhand

CALL MPI_Errhandler_create(Errhand, handler, & ierror)

- **Errhand** is the **EXTERNAL** subroutine to be called upon MPI errors (in)
- **handler** is the new MPI error handler ID (out) that can then be set/activated using **MPI_Errhandler_set**
User-defined error handlers

- The **EXTERNAL** subroutine to be used as an error handler must have the interface:

  ```fortran
  SUBROUTINE Errhand(comm,error_code)
  INTEGER :: comm,error_code
  ```

- The first argument passed to the subroutine upon error is the communicator in which the error occurred. The second argument is the MPI error code.
Destroying a user-defined error handler

\[
\text{INTEGER :: handler, ierror} \\
\text{CALL MPI_Errhandler_free(handler, ierror)}
\]

- \textbf{handler} is the error handler ID (opaque object) to be destroyed
Parallel debugging
Debugging is hard (and tedious!)

*It is virtually impossible to predict the behavior of an erroneous program*

- Same errors may produce different error messages on different computer systems (or none at all!)
- Programs often do not fail at the location of the error, but somewhere else
- Parallel programs can exhibit non-deterministic behavior (important difference to sequential programs!)
Non-determinism

- The exact sequence of computations and communications performed in a parallel program may vary from execution to execution.
- There is no global clock in a distributed-memory system.
- Processes run at independent speeds and with unpredictable OS-load fluctuations.
- Different processes can be mapped to different (hardware) processors at different executions of the program.
- Messages are sent in unpredictable order.
MPI bugs that lead to non-determinism

- Order of identical messages between a fixed pair of communication partners is unpredictable unless unique message tags are used.
- Order of messages arriving is unpredictable if `MPI_ANY_SOURCE` or `MPI_ANY_TAG` are used.
- If not checked, leads to wrong results if subsequent mathematical operations are not commutative.

Make sure to always specify all communications uniquely!
Basic debugging techniques

- Examine the source code
- Use a different (more picky) compiler or a different machine/OS
- Add debugging output to the program
- Use a symbolic debugger
Examine the source code

- Most “parallel bugs” are actually sequential bugs, such as:
  - Forgot to allocate memory
  - Out-of-bound index in an array
  - Subroutine interfaces in the `MODULE` file and in the `SUBROUTINE` declaration do not match
  - Variables of wrong data type passed to a subroutine
  - Reallocating an `ALLOCATABLE` in a subroutine
  - Exceeding the system’s stack memory (try unlimit the stack size and then move data to a module!)
  - …
Frequently made MPI programming mistakes in F90/95:

- Use non-contiguous sub-arrays in non-blocking MPI calls
- Forget `MPI_ADDRESS` before a non-blocking `Wait`
- `MPI_Send` and `MPI_Recv` use different data types
- Trying to receive data that has never been sent (symptom: call to `MPI_Recv` hangs)
- Deadlocks
- Message tags are not matching
- Source or destination rank are wrong in MPI routines

Examine the source code
Rank 0 and 1 want to exchange the values of x and y. How will the code behave? What are possible solutions?
Add debugging output

```fortran
CALL MPI_Barrier(MPI_COMM_WORLD,ierror)
PRINT*, 'rank ', myrank, ' --> data:', x
CALL MPI_Abort(MPI_COMM_WORLD,err,ierror)
STOP
```

- Add the above snippet for parallel output
- Always print the rank of the process as well!
- Barrier makes sure that all processes get there
- The **PRINT/STOP** method is often the best method for parallel debugging
- Run program with **small** toy data (mind the stack!)
Problems with debugging output

- May change the behavior of the program (different compiler optimizations / cache coherence).
- May introduce new bugs or hide the effects of existing bugs (e.g. stack problems).
- May alter the non-deterministic behavior of the program (new collective communications!).
- Is not scalable to large numbers of processors.
- Can drastically reduce performance.
- Some OS may have problems when many processes try to write output at once.
Add debugging output

Make sure that communications behave as expected

- Print out the arguments passed to and returned from all MPI communication functions.
- Use MPE log calls (not covered in this course) and the MPE tools to analyze the behavior of the MPI messages and trace them.
A “recipe” for parallel debugging

1. Turn off all compiler optimizations and eliminate all compiler warnings (use the most picky compiler).
2. Make sure the program works on a single processor.
3. Make sure all communication operations are doing what they should ([PRINT/STOP method]) on different (even, odd, powers of 2, prime), rather small numbers of processors and small test data.
4. Turn compiler optimizations on and repeat all test.
A conundrum in parallel debugging

- Program works on a single processor
- Program also works on multiple processors with the same test data
- Program does not run on multiple processors with the real data
- Real data are too large to test on a single processor

Check for stack size overflows and locally defined dynamic arrays!
Brief overview of parallel debuggers

- Parallel debuggers are not very well developed yet and are subject to active research.
- Generating output from a parallel program can be very difficult (large data).
- Analyzing the output from a parallel program is difficult because the order is non-deterministic.
- Sometimes it’s enough to start the program in a sequential debugger (such as gdb or dbx) with:
  
  ```
  gdb mpirun -np 1 program
  ```
VAMPIR

- Communication analyzer
- Generates timelines and statistics based on trace files
- Useful to detect deadlocks
- Available for Linux, MacOS, and Windows
- Commercial (GWT-TUD)
- Actively maintained and developed
TotalView

- Symbolic debugger
- Need to compile with `-g` to include symbols
- Memory analyzer
- Performance tracer
- Parallel (MPI, OpenMP, openACC, CUDA) debugger and tracer
- XeonPhi support
- Specific Fortran support
- For Linux, MacOS, SPARC, CrayOS, AIX
- Commercial (RogueWave Software)