Recap: MPI & OpenMP

**Threads & processes**

**Process**
- Independent execution units
- Have their own state information and use their own address spaces

**Thread**
- A single process may contain multiple threads
- All threads within a process share the same state and same memory space

**OPENMP WALKTHROUGH**

- Short-lived: threads are created by forking and destroyed by joining them
- Communicate directly through the shared memory

4 parallel processes

- Process
- Thread

Spawned when starting the parallel program and killed when its finished
Typically communicate using MPI in supercomputers
Three components of OpenMP

- Compiler directives and constructs
  - Expresses shared memory parallelization
  - Preceded by sentinel, can compile serial version
- Runtime library routines
  - Small number of library functions
  - Can be discarded in serial version via conditional compiling
- Environment variables
  - Specify the number of threads, etc.

OpenMP directives

- Sentinels precede each OpenMP directive
  - C/C++: `#pragma omp`
  - Fortran free form: `!$omp`
- Compilers that support OpenMP usually require an option (flag) that enables the feature
  - Without an enabling flag the OpenMP sentinels are treated as comments and a serial version will be compiled

Parallel construct

- Defines a parallel region
  - Prior to it only one thread, master
  - Creates a team of threads: master+slave threads
  - At end of the block is a barrier and all shared data is synchronized

Example: Helloworld with OpenMP

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char argv[]){
  int omp_rank;
  #pragma omp parallel private(omp_rank)
  {
    int omp_rank = omp_get_thread_num();
    printf("Hello world! by thread %d", omp_rank);
  }
}
```

```
tn omp_hello.f90 -o omp
> setenv OMP_NUM_THREADS 4
> aprun -n 1 -d 4 ./omp
Hello world! by thread 0
Hello world! by thread 2
Hello world! by thread 3
Hello world! by thread 1

> cc omp_hello.c -o omp
> setenv OMP_NUM_THREADS 4
> aprun -n 1 -d 4 ./omp
Hello world! by thread 2
Hello world! by thread 3
Hello world! by thread 0
Hello world! by thread 1
```
How do the threads interact?

Because of the shared address space threads can "communicate" using shared variables.

Threads often need some private work space together with shared variables.
- For example the index variable of a loop.

Visibility of different variables is defined using data-sharing clauses in the parallel region definition.
- private, firstprivate, lastprivate, shared, default.
- Local variables defined in the parallel region, or functions called within it are private.

Work sharing

Parallel region creates an "Single Program Multiple Data" instance where each thread executes the same code.

How can one split the work between the threads of a parallel region?
- Loop construct
- Single/Master construct
- Sections
- Task construct (in OpenMP 3.0 and above)

Loop constructs

Directive instructing compiler to share the work of a loop
- Fortran: $OMP DO
- C/C++: #pragma omp for
- Directive must be inside a parallel region
- Can also be combined with parallel:
  $OMP PARALLEL DO / #pragma omp parallel for

Loop index is private by default.

Work sharing can be controlled using schedule clause
- static, dynamic, guided, or runtime.

Reduction clause

reduction(operator:var_list)
- Performs reduction on the (scalar) variables in list (sum, max, min, ...)
- Private reduction variable is created for each thread’s partial result.
- Private reduction variable is initialized to operator’s initial value, e.g., 0 for sum.
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable.
Execution controls

- Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
  - I/O, initializations, updating global values, etc.
  - Remember the synchronization!

- OpenMP provides clauses for controlling the execution of code blocks
  - barrier
  - master & single
  - critical

OpenMP summary

- Work sharing
- Parallel do/for
- Threads
- Reduction
- Data visibility
- Synchronization
- Critical section
- Private
- Shared
- Single/Master

Message-passing interface

- MPI is an application programming interface (API) for communication between separate processes
  - The most popular distributed parallel computing method
  - MPI programs are portable and scalable

- MPI is flexible and comprehensive
  - Several communication methods and patterns
  - Parallel IO

- MPI standardization by MPI Forum
  - Latest version is 3.1, version 1.0 in 1994

GETTING STARTED WITH MPI

- Message-passing interface
  - MPI is an application programming interface (API) for communication between separate processes
    - The most popular distributed parallel computing method
    - MPI programs are portable and scalable
  
  - MPI is flexible and comprehensive
    - Several communication methods and patterns
    - Parallel IO

  - MPI standardization by MPI Forum
    - Latest version is 3.1, version 1.0 in 1994
Execution model in MPI

- Parallel program is launched as set of *independent, identical processes*
  - The same program code and instructions
- MPI runtime assigns each process a *rank*
  - Identification of the processes
  - Processes can perform different tasks and handle different data basing on their rank
  - Can reside in different nodes
- The way to launch parallel program is implementation dependent

Communication

- Data is local to the MPI processes
  - They need to *communicate* to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time

MPI point-to-point operations

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

Non-blocking communication

- Non-blocking communication is usually the smarter way to do point-to-point communication in MPI
  - Enables some computing concurrently with communication
  - Avoids many common dead-lock situations
- Non-blocking communication realization
  - MPI_Isend
  - MPI_Irecv
  - MPI_Wait / MPI_Waitall
Collective operations examples

Processes

MPI_Bcast

Send buffer

Recv buffer

MPI_Scatter

Send buffer

Recv buffer

MPI_Gather

Send buffer

Recv buffer

MPI_Alltoall

Send buffer

Recv buffer

Communicators

User-defined communicators

One-to-all collectives

Point-to-point communication

Collective communication

All-to-one collectives

All-to-all collectives

Send & Recv

Sendrecv

Basic MPI summary

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

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

Collective operations examples

MPI datatypes

Communicators

Basic MPI summary
Hybrid MPI+OpenMP programming

Hybrid programming:
Launch threads within MPI tasks

MPI: Processes
- Independent execution units
- MPI launches N processes at application startup

OpenMP: Threads
- Threads share memory space
- Threads are created and destroyed (parallel regions)

Hybrid programming:
Launch threads within MPI tasks

Shared memory programming inside a node, message passing between nodes
Matches well modern supercomputer hardware
Often one or a few MPI tasks per CPU, but one should experiment with the ratio
Hybrid hello

```c
int main(int argc, char *argv[]){
    int my_id, omp_rank;
    int required=MPI_THREAD_FUNNELED, provided;
    MPI_Init_thread(&argc,&argv,required,provided);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
    #pragma omp parallel private(omp_rank)
    {
        omp_rank=omp_get_thread_num();
        printf("I'm thread %d in process %d\n", omp_rank, my_id);
    }
    MPI_Finalize();
}
```

Thread support in MPI

- **MPI only**
  - Only one thread will execute
  - (MPI_THREAD_SINGLE)

- **Hybrid**
  - MPI+OpenMP
  - MPI only on main thread
  - (MPI_THREAD_FUNNELED)

- **OpenMP only**
  - No concurrent MPI calls
  - (MPI_THREAD_SERIALIZED)

  - No restrictions
  - (MPI_THREAD_MULTIPLE)

Hybrid programming styles: fine/coarse grained

- **Fine-grained**
  - Use `omp parallel do/for` on the most intensive loops
  - Possible to hybridize an existing MPI code with little effort and in parts

- **Coarse-grained**
  - Use OpenMP threads to replace MPI tasks
  - Whole (or most of) program within the same parallel region
  - More likely to scale over the whole node, enables all cores to communicate (if supported by MPI implementation)

Potential advantages of the hybrid approach

- Fewer MPI processes for a given amount of cores
  - Improved load balance
  - All-to-all communication bottlenecks alleviated
  - Decreased memory consumption if an implementation uses replicated data
  - Larger domains in domain decomposition: less neighbor data communication, smaller memory consumption, improved load balance
**Potential advantages of the hybrid approach**
- Possibility for dedicating threads for different tasks
  - e.g., dedicated communication thread or parallel I/O
  - Note that OpenMP worksharing constructs (e.g. OMP DO) will not be applicable when dedicating threads

**Disadvantages of hybridization**
- Increased overhead
  - thread creation/destruction
  - Synchronization (implicit barriers, critical regions)
- More complicated programming
  - Code readability and maintainability issues
- Amdahl’s law limits scalability
  - Often only a part of the code is threaded
- Consider thread safety of MPI and other libraries

**Summary**
- Hybrid programming maps well to modern hardware
- In theory, hybrid programming offers several advantages
- In practice, all the advantages can be difficult to realize
- As number of cores inside a node increases, advantages of hybrid approach are likely to become more and more relevant
- MPI provides different levels of thread support
Multiple thread communication

Hybrid programming is relatively straightforward in cases where communication is by only single thread at time
– FUNNELED, SERIALIZED

When multiple threads communicate, also the sending and receiving threads should normally match
– thread specific tags
– thread specific communicators

Thread specific tags

In point-to-point communication thread id can be used as the most significant digit in the tag

Maximum allowed tag is 32767, or more, according to standard
– Using larger tags is unsafe
– In practice often larger, e.g., on Sisu 2097151

```c
void *v; int flag;
MPI_Comm_get_attr( MPI_COMM_WORLD, MPI_TAG_UB, &v, &flag );
if (flag) 
  printf("Max tag is %d\n", *(int*)v );
```

Thread specific tags

OpenMP parallel region
```c
!$OMP PARALLEL PRIVATE(TID, TIDtag, ierr)
TID = OMP_GET_THREAD_NUM()
! Max TID 16, max tag 16383
TIDtag = 2**10 * TID + tag

! MPI communication to the same thread on another process
CALL MPI_SENDRECV(senddata, n, MPI_REAL, pid, TIDtag, &
  recvdata, n, MPI_REAL, pid, TIDtag, &
  MPI_COMM_WORLD, stat, ierr)
! Details omitted
!$OMP END PARALLEL
```
MPI thread support: collectives

- MPI standard allows multiple threads to call collectives simultaneously
  - Programmer must ensure that the same communicator is not being concurrently used by two different collective communication calls at the same process
- In most cases, even with MPI_THREAD_MULTIPLE it is beneficial to perform the collective communication from a single thread (usually the master thread)
- NOTE: MPI collective communication calls do not guarantee synchronization of the thread order

Thread specific communicators

- Collectives do not have tags
- Thread specific communicators can be created by splitting (MPI_COMM_SPLIT) or by duplicating (MPI_COMM_DUP)

Thread specific communicators

! Total number of OpenMP threads
nthr = OMP_GET_MAX_THREADS()
ALLOCATE(tcomm(nthr))
! Split the communicator
DO thrid=1,nthr
  col = thrid
  CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, col, &
                      procid, tcomm(thrid), ierr)
END DO
! Collective threadwise communication
!$OMP PARALLEL PRIVATE(TID, ierr)
TID = OMP_GET_THREAD_NUM()+1
  ! Collective operations on tcomm(TID)
!$OMP END PARALLEL

MPI thread support levels

- Modern MPI libraries support all threading levels
  - Cray: Set MPICH_MAX_THREAD_SAFETY environment variable to single (default), funneled, serialized, multiple to select the threading level
  - Intel: When compiling with -qopenmp a thread safe version of the MPI library is automatically used
- Note that using MPI_THREAD_MULTIPLE requires the MPI library to internally lock some data structures to avoid race conditions
  - May result in additional overhead in MPI calls

MPI ping-pong benchmark

- Two node benchmark with either 48x1 threads (Pure MPI, SINGLE) or 2x24 threads (MPI+OpenMP, MULTIPLE)
- Every thread sends a vector of length N to the corresponding thread of the other node and then receives a vector back from the other node (and vice versa)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>N=2^{19}=65536</th>
<th>N=2^{20}=524288</th>
<th>N=2^{21}=4194304</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure MPI (48x1)</td>
<td>0.00143 s</td>
<td>0.01252 s</td>
<td>0.11903 s</td>
</tr>
<tr>
<td>MPI+OpenMP (2x24)</td>
<td>0.00185 s</td>
<td>0.01135 s</td>
<td>0.09297 s</td>
</tr>
</tbody>
</table>

MPI ping-pong benchmark results on taito.csc.fi, average time. Measurements performed with EPCC OpenMP/MPI Microbenchmarks, http://www2.epcc.ed.ac.uk/~markb/mpipopenmbench/intro.html
MPI thread support levels: collective example

**MPI Allreduce benchmark**
- Multiple node benchmark with either 384x1 threads (Pure MPI, SINGLE) or 16x24 threads (MPI+OpenMP, MULTIPLE)
- Compute columnwise two-norm of a rowwise distributed matrix with M rows and N=1000 columns. Collective call performed by one thread per MPI rank per iteration.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>M=100k</th>
<th>M=500k</th>
<th>N=1M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure MPI (384x1)</td>
<td>0.0525 s</td>
<td>0.258 s</td>
<td>0.513 s</td>
</tr>
<tr>
<td>MPI+OpenMP (16x24)</td>
<td>0.0738 s</td>
<td>0.263 s</td>
<td>0.484 s</td>
</tr>
</tbody>
</table>

MPI Allreduce benchmark results on taito.csc.fi, total time for 100 iterations.

**MPI+OpenMP thread affinity**

- MPI library must be aware of the underlying OpenMP for correct allocation of resources
  - Oversubscription of CPU cores may cause significant performance penalty
- Additional complexity from batch job schedulers
- Heavily dependent on the platform used!

```
Example (incorrect): oversubscription of resources

Example (correct): better use of hardware resources
```

**MPI+OpenMP thread affinity (Cray)**

- Cray application launcher aprun fully supports interplay between MPI processes and OpenMP threads
- Some aprun flags related to thread affinity
  - `-d <depth>`, reserve depth CPU cores for each MPI task. Recommended to be set equal to OMP_NUM_THREADS
  - `-cc <list|cpu|numa_node|none>`, set CPU affinity for threads within each MPI task. Each thread is pinned to a CPU core (cpu) or to a NUMA node (numa_node). With none the affinity control is disabled

**MPI+OpenMP thread affinity (Intel)**

- Intel MPI library allows affinity control of MPI domains via I_MPI_PIN, I_MPI_PIN_DOMAIN and I_MPI_PIN_ORDER environment variables
- Affinity control of the MPI library only applies to the placement of the MPI processes
  - OMP_PLACES (or KMP AFFINITY) environment variables must be used to pin threads to cores within MPI domains
**Summary**

- Hybrid programming maps well to modern HPC systems
- MPI provides different levels of thread support
  - Impact on the algorithms used
  - Impact on the achievable performance
- Thread affinity matters
  - See manuals and user guides of the system if performance does not match what is expected

**MPI+OpenMP thread affinity (Intel)**

- MPI domain affinity via explicit shape setting
  
  \[ \text{I\_MPI\_PIN\_DOMAIN} = \langle \text{size} \rangle[:\langle \text{layout} \rangle] \]

  - \( \text{size} = \langle \text{auto|omp|n} \rangle \) defines the number of logical processors within each domain. Can be either the value of \text{OMP\_NUM\_THREADS (omp)}, automatically computed from the formula \#cpu/#proc (auto) or an explicit value (n)
  
  - \( \text{layout} = \langle \text{compact|platform|scatter} \rangle \) defines the ordering of the domain members within the node.

- Intel MPI domain affinity control can be disabled with

  \[ \text{I\_MPI\_PIN\_DOMAIN} = \text{none} \text{ or } \text{I\_MPI\_PIN} = \text{disable} \]
User-defined datatypes

- Use elementary datatypes as building blocks to define new types to match the communication, e.g.
  - 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

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

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
  ```c
  MPI_Type_commit(newtype)
  ```
  `newtype` the new datatype to commit
- A type should be freed after it is no longer needed
  ```c
  MPI_Type_free(newtype)
  ```
  `newtype` the datatype for decommision
### Example: sending rows of a matrix in Fortran

```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)
```

### Datatype constructor examples

<table>
<thead>
<tr>
<th>Datatype Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Type_contiguous</td>
<td>contiguous datatypes</td>
</tr>
<tr>
<td>MPI_Type_vector</td>
<td>regularly spaced datatype</td>
</tr>
<tr>
<td>MPI_Type_indexed</td>
<td>variably spaced datatype</td>
</tr>
<tr>
<td>MPI_Type_create_subarray</td>
<td>subarray within a multi-dimensional array</td>
</tr>
<tr>
<td>MPI_Type_create_hvector</td>
<td>like vector, but uses bytes for spacings</td>
</tr>
<tr>
<td>MPI_Type_create_hindexed</td>
<td>like index, but uses bytes for spacings</td>
</tr>
<tr>
<td>MPI_Type_create_struct</td>
<td>fully general datatype</td>
</tr>
</tbody>
</table>

### Understanding datatypes: typemap

- A datatype is defined by a typemap
  - pairs of basic types and displacements (in bytes)
  - E.g. `MPI_INT={(int,0)}`

### Understanding datatypes: extent

- The **extent** of a datatype describes the stride at which elements are laid out in memory
  - Padding may occur due to data alignment reasons
- Size of datatype is the actual size of the data
  - For basic types, extent and size are equal
  - Extent and size of derived datatype are often different
Datatype constructors: 
**MPI_TYPE_CREATE_STRUCT**

The most general type constructor, creates a new type from heterogeneous blocks
- E.g. Fortran 9x types and C structures
- Input is the typemap

Example: sending a C struct

```c
struct ParticleStruct {
  int charge; /* particle charge */
  double coord[3]; /* particle coords */
  double velocity[3]; /* particle velocity vector components */
};

struct ParticleStruct particle[1000];
MPI_Datatype Particletype;

MPI_Datatype type[3]={MPI_INT, MPI_DOUBLE, MPI_DOUBLE};
int blocklen[3]={1,3,3};
MPI_Aint disp[3]={0, sizeof(double), 4*sizeof(double)};
...

MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);

MPI_Send(particle, 1000, Particletype, dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&Particletype);
```

Determining displacements

- The previous example defines and assumes a certain alignment for the data within the structure
- The displacements can also be determined by using function

  ```c
  MPI_Get_address(pointer, address)
  ```
  - The address of the variable is returned, which can then be used for determining relative displacements
Determining displacements

/* Structure for particles */
struct ParticleStruct {
  int charge;  /* particle charge */
  double coords[3]; /* particle coords */
  double velocities[3]; /* velocity components */
};

struct ParticleStruct particle[1000];
...
MPI_Aint disp[3];
MPI_Get_address(&particle[0].charge, &disp[0]);
MPI_Get_address(&particle[0].coords, &disp[1]);
MPI_Get_address(&particle[0].velocities, &disp[2]);
/* Make displacements relative */
disp[2] -= disp[0];
disp[1] -= disp[0];
disp[0] = 0;
...

Gaps between datatypes

- Sending of an array of the ParticleStruct structures may have a portability issue: it assumes that array elements are packed in memory
  - Implicit assumption: the extent of the datatype was the same as the size of the C struct
  - This is not necessarily the case
- If there are gaps in memory between the successive structures, sending does not work correctly

Getting and setting extent

- Extent of a datatype can be obtained with the routine
  
  ```
  MPI_Type_get_extent(datatype, lb, extent)
  ```
  
  datatype  datatype handle
  lb        lower boundary of datatype
  extent    extent of datatype

- Extent can be changed with routine
  
  ```
  MPI_Type_create_resized(oldtype, newlb, newextent, newtype)
  ```
  
  oldtype    input
  newlb      lower boundary for datatype
  newextent  extent of new datatype

- The new, resized datatype must be committed before use

Example: sending an array of structs portably

```c
struct ParticleStruct particle[1000];
MPI_Datatype particletype, oldtype;
MPI_Aint lb, extent;
...
/* Check that the extent is correct
MPI_Type_get_extent(particletype, &lb, &extent);
if ( extent != sizeof(particle[0]) ) {
  oldtype = Particletype;
  MPI_Type_create_resized(oldtype, 0, sizeof(particle[0]), &particletype);
  MPI_Type_commit(&particletype);
}
MPI_Type_free( &oldtype);
...```

The MPI_Aint values are tracked separately and are passed to the routines as pointers.
Other ways for communicating non-uniform data

- Non-contiguous data by manual packing
  - Copy data into or out from temporary buffer
  - Use MPI_Pack and MPI_Unpack functions
  - Performance will likely be an issue
- Structures and types as continuous stream of bytes:
  Communicate everything using MPI_BYTE
  - Portability can be an issue - be careful

```c
struct ParticleStruct particle[1000];
int psize;
psize = sizeof(particle[0]);
MPI_Send(particle, 1000*psize, MPI_BYTE, ...);
```

Summary

- User-defined types enable communication of non-contiguous or heterogeneous data with single MPI communication operations
  - Improves code readability & portability
  - Allows optimizations by the MPI runtime
- This time we focused on the most general type specification: MPI_Type_create_struct
- Introduced the concepts of extent and typemap
Communication topologies

Process topologies

- MPI process topologies allow for simple referencing scheme of processes
  - Process topology defines a new communicator
  - We will focus on Cartesian topologies, although graph topologies are also supported

- 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

Creating a communicator ordered in Cartesian grid

\[
\text{MPI\_Cart\_create(oldcomm, ndims, dims, periods, reorder, newcomm)}
\]

- 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

Translating rank to coordinates

Checking the Cartesian communication topology coordinates for a specific rank

\[
\text{MPI\_Cart\_coords(comm, rank, maxdim, coords)}
\]

- comm: Cartesian communicator
- rank: rank to convert
- maxdim: dimension of coords
- coords: coordinates in Cartesian topology that corresponds to rank
Translating coordinates to rank

Checking the rank of the process at specific Cartesian communication topology coordinates

MPI_Cart_rank(comm, coords, rank)

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

Creating a Cartesian communication topology

dims(1)=4
dims(2)=4
period=(/ .true., .true. /)
call mpi_cart_create(comm2d, my_id, rc)
call mpi_comm_rank(comm2d, my_id, rc)
call mpi_cart_coords(comm2d, my_id, 2, & coords, rc)

Halo exchange

call mpi_cart_shift(comm2d,0,1,nbr_up,nbr_down,rc) 
call mpi_sendrecv(comm2d, tag_left, buf(n+1,1), 1, coltype, nbr_left, & tag_left, buf(1,n+1), 1, coltype, nbr_right, & tag_left, comm2d, mpi_status_ignore, rc)

How to communicate in a Cartesian topology

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

comm Cartesian communicator
direction shift direction (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! Both source and dest are output parameters. The coordinates of the calling task is implicit input.

"Shifts" (steps) within the grid define sources/destinations

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

Halo exchange

call mpi_cart_shift(comm2d,0,1,nbr_up,nbr_down,rc)
call mpi_sendrecv(comm2d, tag_left, buf(n+1,1), 1, rowtype, nbr_left, & tag_left, buf(1,n+1), 1, rowtype, nbr_down, & tag_left, comm2d, mpi_status_ignore, rc)

Note!

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

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

Two components of message-passing:
- Data movement
- Synchronization

One-sided communication:
- Only single process calls data movement functions
- Communication patterns specified by only a single process
- Remote memory access (RMA)

Why one-sided communication?
- Applications with dynamic communication patterns
  - Easier programming
- Less overheads
  - Synchronization
  - Matching of sender and receiver
- Hardware support for remote memory access
  - Cray XC40 Sisu at CSC

Origin and target
- Origin process: a process which calls data movement function
- Target process: a process whose memory is accessed
Remote memory access window

- *Window* is a region in process’s memory which is made available for remote operations
- *Windows* are created by collective calls
- *Windows* may be different in different processes

Data movement operations

- PUT data to the memory in target process
  - From local buffer in origin to the *window* in target
- GET data from the memory of target process
  - From the *window* in target to the local buffer in origin
- ACCUMULATE data in target process
  - Use local buffer in origin and update the data (e.g. add the data from origin) in the *window* in target
  - One-sided reduction

Synchronization

- RMA communication is non-blocking
- Communication takes place within *epochs*
  - Synchronization calls start and end an *epoch*
  - There can be multiple data movement calls within *epoch*
  - *Epoch* is specific to particular window
- Active synchronization:
  - Both origin and target perform synchronization calls
- Passive synchronization:
  - No MPI calls at target process

One-sided communication in a nutshell

- Define memory *window*
- Start epoch
  - Target: exposure epoch
  - Origin: access epoch
- GET, PUT, and ACCUMULATE data
- Complete the communications by ending the epoch
Creating an window

\begin{verbatim}
MPI_Win_create(base, size, disp_unit, info, comm, win)
\end{verbatim}

- \texttt{base} (pointer to) local memory to expose for RMA
- \texttt{size} size of a window in bytes
- \texttt{disp_unit} local unit size for displacements in bytes
- \texttt{info} hints for implementation
- \texttt{comm} communicator
- \texttt{win} handle to window

The window object is deallocated with

\begin{verbatim}
MPI_Win_free(win)
\end{verbatim}

Starting and ending an epoch

\begin{verbatim}
MPI_Win_fence(assert, win)
\end{verbatim}

- \texttt{assert} optimize for specific usage. Valid values are
  - "0", MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT,
  - MPI\_MODE\_NOPRECEDE, MPI\_MODE\_NOSUCCEED
- \texttt{win} window handle

- Used both for starting and ending an epoch
  - Should both precede and follow data movement calls
- Collective, barrier-like operation

Data movement: Put

\begin{verbatim}
MPI_Put(origin, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)
\end{verbatim}

- \texttt{origin} (pointer to) local data to be send to target
- \texttt{origin_count} number of elements to put
- \texttt{origin_datatype} MPI datatype for local data
- \texttt{target_rank} rank of the target task
- \texttt{target_disp} starting point in target window
- \texttt{target_count} number of elements in target
- \texttt{target_datatype} MPI datatype for remote data
- \texttt{win} RMA window
Data movement: Get

**MPI_Get**

```
MPI_Get(origin, origin_count, origin_datatype,
target_rank, target_disp, target_count,
target_datatype, win)
```

- `origin` (pointer to) local buffer in which to receive the data
- `origin_count` number of elements to get
- `origin_datatype` MPI datatype for local data
- `target_rank` rank of the target task
- `target_disp` starting point in target window
- `target_count` number of elements from target
- `target_datatype` MPI datatype for remote data
- `win` RMA window

Data movement: Accumulate

**MPI_Accumulate**

```
MPI_Accumulate(origin, origin_count, origin_datatype,
target_rank, target_disp, ...
```

- `origin` (pointer to) local data to be accumulated
- `origin_count` number of elements to put
- `origin_datatype` MPI datatype for local data
- `target_rank` rank of the target task
- `target_disp` starting point in target window
- `target_count` number of elements for target
- `target_datatype` MPI datatype for remote data
- `op` accumulation operation (as in MPI_Reduce)
- `win` RMA window

Simple example: Put

```c
... double data;
MPI_Win window;

data = rank;
// Create window
MPI_Win_create(&data, sizeof(double), sizeof(double), MPI_INFO_NULL,
   MPI_COMM_WORLD, &window);

MPI_Win_fence(0, window);
if (rank == 0)
  MPI_Put(&data, 1, MPI_DOUBLE, 1, 0, 1, MPI_DOUBLE, window);
MPI_Win_fence(0, window);
MPI_Win_free(&window);
...```

Limitations for data access

- Compatibility of local and remote operations when multiple processes access a window during an epoch

<table>
<thead>
<tr>
<th>Operation</th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Get</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td></td>
</tr>
<tr>
<td>Put</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Acc</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- No limitations
- Operations on non-overlapping parts of window allowed
- Not allowed
Advanced synchronization

- Assert arguments in MPI_Win_fence:
  - `MPI_MODE_NOSTORE`
    The local window was not updated by local stores (or local get or receive calls) since last synchronization.
  - `MPI_MODE_NOPUT`
    The local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
  - `MPI_MODE_NOPRECEDE`
    The fence does not complete any sequence of locally issued RMA calls.
  - `MPI_MODE_NOSUCCEED`
    The fence does not start any sequence of locally issued RMA calls.

Enhancements in MPI 3

- New window creation function: `MPI_Win_allocate`,
  - Allocate memory and create window at the same time
- Dynamic windows: `MPI_Win_create_dynamic, MPI_Win_attach, MPI_Win_detach`
  - Non-collective exposure of memory
- New data movement operations: `MPI_Get_accumulate, MPI_Fetch_and_op, MPI_Compare_and_swap`
- New memory model
  `MPI_Win_allocate_shared`
  - Allocate memory which is shared between MPI tasks
- Enhancements for passive target synchronization

Advanced synchronization

- More control on epochs can be obtained by starting and ending the exposure and access epochs separately
- Target: Exposure epoch
  - Start: `MPI_Win_post`
  - End: `MPI_Win_wait`
- Origin: Access epoch
  - Start: `MPI_Win_start`
  - End: `MPI_Win_compete`
Summary

One-sided communication allows communication patterns to be specified from a single process.

- Can reduce synchronization overheads and provide better performance especially on recent hardware.

Basic concepts:
- Origin and target process
- Memory window
- Communication epoch
Parallel I/O – overview

Wednesday

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.00-9.45</td>
<td>Getting started with parallel I/O</td>
</tr>
<tr>
<td>9.45-10.00</td>
<td>Coffee</td>
</tr>
<tr>
<td>10.00-11.30</td>
<td>Exercises</td>
</tr>
<tr>
<td>11.30-12.00</td>
<td>Parallel I/O with MPI</td>
</tr>
<tr>
<td>12.00-13.00</td>
<td>Lunch</td>
</tr>
<tr>
<td>13.00-13.45</td>
<td>Parallel I/O with MPI (cont.)</td>
</tr>
<tr>
<td>13.45-14.30</td>
<td>Exercises</td>
</tr>
<tr>
<td>14.30-14.45</td>
<td>Coffee</td>
</tr>
<tr>
<td>14.45-15.30</td>
<td>HDF5</td>
</tr>
<tr>
<td>15.30-16.15</td>
<td>Exercises</td>
</tr>
<tr>
<td>16:15-16:30</td>
<td>Wrap-up</td>
</tr>
</tbody>
</table>

Parallel I/O

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

**Challenges**

- Number of tasks is rising rapidly
- Size of the data is also rapidly increasing
- Disparity of computing power vs. I/O performance is getting worse and worse
- The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!

**I/O layers**

- **High-level**
  - Applications
    - High level I/O
      - Libraries
        - HDF5, NetCDF,...

- **Intermediate level**
  - POSIX syscalls
  - MPI I/O

- **Low-level**
  - Parallel file system
    - Lustre, GPFS,...

**PARALLEL FILE SYSTEMS**
## File systems

- Practically all large parallel computer systems provide a parallel file system area
  - Files can be accessed from all tasks
  - Large systems often have dedicated I/O nodes
- Some systems also provide a local disk area for temporary storage
  - Only visible to tasks on the same node
  - Results have to be copied after simulation

## Lustre

- Lustre is a popular parallel file system that is used in many large systems
  - Also at CSC (Sisu, Taito)
- Separated storage of data and metadata
  - Single metadata server
  - Clustered data storage
  - Supports e.g. striping of large datafiles for higher performance

### Lustre architecture

- Metadata Server (MDS)
- Metadata Target (MDT)
- Object Storage Servers (OSSs)
- Object Storage Targets (OSTs)
- Lustre Clients

### Lustre file striping

- Data file
- Striped data file
- I/O tasks
- I/O tasks
Lustre file striping

- Striping pattern of a file/directory can be queried or set with the lfs command
- `lfs getstripe <dir|file>`
- `lfs setstripe -c count dir`
  - Set the default stripe count for directory `dir` to `count`
  - All the new files within the directory will have the specified striping
  - Also stripe size can be specified, see `man lfs` for details
- Proper striping can enhance I/O performance a lot

Performance with striping

- Parallel write in Sisu

  ![Graph showing performance with different stripe counts](image)

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)
Special case: stdout and stderr

- Standard Output and Error streams are effectively serial I/O and will be a severe bottleneck for application scaling
- Disable debugging messages when running in production mode
  - “Hello, I’m task 32,000!”
- Ensure only the very minimum is written to the stdout/err!
  - Interim results, timings,…

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

Example: spokesman strategy

if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_recv(full_data(i*n), n, &
      MPI_REAL, i, tag, &
      MPI_COMM_WORLD, rc)
  end do
  open(funit, file=fname, access="stream")
  write(funit) full_data
  close(funit)
else
  call mpi_send(data, n, &
    MPI_REAL, 0, tag, &
    MPI_COMM_WORLD, status, rc)
end if
Summary

- Parallel file system is needed for efficient parallel I/O
  - Striping of files
- Multiple processes can perform I/O with Posix calls
  - Spokesman strategy
  - Every man for himself
  - Subset of writers/readers

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 to implementing the same patterns on your own)
- Portable programming interface
  - By default, binary files are not portable

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 or provided as an explicit offset from the beginning of the file
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, every man for himself

Opening & Closing files

All processes in a communicator open a file using

```c
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

File is closed using

```c
MPI_File_close(fhandle)
```

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

File reading

- Read file at explicitly defined offsets
  - Thread-safe
  - The file pointer is neither referred or incremented

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

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

File writing

- Read file at explicitly defined offsets
  - Thread-safe
  - The file pointer is neither used or incremented

```c
MPI_File_write_at(fhandle, disp, buf, count, datatype, status)
```
Example: parallel write

```plaintext
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
  call mpi_comm_rank(mpi_comm_world, myid, &
    err)
  do i = 1, count
    buf(i) = myid * count + i
  end do
end program output
```

File pointer

- The file pointer can be also updated with
  
  ```plaintext
  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
  ```

- There are routines `MPI_File_write` and `MPI_File_read` that use the updated file handle (and not explicit offset)

File reading

- Read file at individual file pointer
  
  ```plaintext
  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

- 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 opened with MPI_MODE_WRONLY or MPI_MODE_CREATE

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)

Non-blocking MPI I/O

- Non-blocking independent I/O is similar to non-blocking send/recv routines
  - `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:
**Non-contiguous data access with MPI I/O**

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.

**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.

### File view

- **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.

### Decomposition for a 2D array

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.
File view for non-contiguous data

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

Collective I/O can enable the actual disk access to remain contiguous

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 and passed as a variable of type integer(MPI_OFFSET_KIND)

Summary

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

I/O LIBRARIES: HDF5
I/O libraries

- How should HPC data be stored?
  - Large, complex, heterogeneous, esoteric, metadata ...
  - Parallel and random access
- Traditional relational databases poor fit
  - Cannot handle large objects
  - Many unnecessary features for HPC data
- Need a better solution
  - HDF5
  - NetCDF

HDF5

- A data model, library, and file format for storing and managing multidimensional data
- Can store complex data objects and meta-data
- File format and files are portable
- Possibility for parallel I/O on top of MPI-IO
- Fortran, C, C++, and Java interfaces
- The HDF5 data model and library are complex

Key concepts

- **File** – contiguous string of bytes (in memory or disc)
- **Group** – collection of objects (including other groups)
- **Dataset** – multidimensional array of data elements
- **Datatype** – description of data element
- **Dataspace** – description of dimensions of multidimensional array
- **Attribute** – named value associated with group or dataset
- **Property list** – parameters controlling options in library. H5P_DEFAULT refers to default options

HDF5 library and programming model

- HDF5 library is written in C, but uses object model
  - Implements *identifiers* to refer objects
  - Identifiers are of type hid_t in C and integer(kind=hid_t) in Fortran
- Functions are grouped using prefixes, e.g. all dataset related functions start with H5D prefix
  - Fortran routines and constants have _f suffix
HDF5 file organisation

A HDF5 file is a container for basic group and dataset objects.
Objects are organized in a tree and referred like in Unix filesystem:
- root group:  /
- group under the root group:  /group_0
- dataset in the group group_1:  /group_1/dataset1

Root group is created automatically, other groups are created by the user.

Creating and opening a HDF5 file

In order to open or create file user must define:
- File access mode
  - H5F_ACC_EXCL or H5F_ACC_TRUNC in create
  - H5F_ACC_RDONLY or H5F_ACC_RDWR in open
- File creation property list (only in create)
- File access property list
- H5P_DEFAULT can be used for the default property list

Creating / opening a file

Creating a file

```c
#include <hdf5.h>
hid_t file_id; /* file identifier */
/* Create a new file using default properties. */
file_id = H5Fcreate("myfile.hdf5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

Opening a file

```c
/* Open a file using default properties. */
file_id = H5Fopen("myfile.hdf5", H5F_ACC_RDWR, H5P_DEFAULT, H5P_DEFAULT);
```

Closing a file

```c
H5Fclose(file_id);
```

HDF5 dataset

A HDF5 dataset stores multidimensional array of data elements.

When creating a dataset, user must specify:
- Name of dataset
- Data type
- Data space
- Storage properties

Basic data types
- H5T_NATIVE_INT, H5T_NATIVE_FLOAT, ...
**HDF5 dataspace**

- A HDF5 dataspace is an object describing dimensionality and shape of multiarray.
- Dataspace is required component of dataset definition.
- Dataspaces are used to control data transfer during read or write.
  - Layout of data in the source (memory during write, dataset during read).
  - Layout of data in the destination (dataset during write, memory during read).

**Creating a dataset**

```c
int ndims = 2;
hsize_t dims[2] = {3, 4};
hid_t filespace_id, dset_id, plist_id;

// Dataspace definition is needed for dataset creation
filespace_id = H5Screate_simple(ndims, dims, NULL);

// Create a dataset of ints with given dataspace
dset_id = H5Dcreate(file_id, "board", H5T_NATIVE_INT,
                      filespace_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
```

**Writing data**

- During I/O operation the raw data in memory can have different type and/or size than in the HDF5 file and therefore one has to specify:
  - The dataset and it’s datatype and dataspace in memory.
  - The dataset’s dataspace in the file.
  - The dataset transfer property list.
  - The data buffer.
- When all these specifications have been done the data can be written with function `H5Dwrite`.

**Reading data**

- Reading of data from HDF5 file proceeds in opposite way to writing:
  - Open the file.
  - Open the dataset.
  - Determine the dataspace of dataset.
  - Determine dataset transfer property list.
  - Read the data from dataset to memory.
- HDF5 provides several `H5xGet_yyyy` routines for obtaining information from datasets and dataspaces.
Data selection

- Dataspace can be used to describe just selected data points instead of the whole array
  - Hyperslabs
  - Selected points

Hyperslab is defined by four arrays
- offset starting point of the hyperslab
- stride distance between successive elements
- count number of blocks to select
- block size of the block

Hyperslabs
Scatter and gather operations by defining different source and destination dataspaces

Parallel I/O with HDF5

- So far we have used the default properties when opening the files
- For parallel I/O, special parameters are needed for file creation and dataset writing
  - HDF5 uses MPI I/O file routines for file operations
  - HDF5 library has to be compiled with parallel IO support (not turned on by default)

File access properties for parallel I/O

```c
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, MPI_COMM_WORLD, MPI_INFO_NULL);
file_id = H5Fcreate("test.h5", H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);
```

Dataset transfer property for parallel I/O

```c
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace, plist_id, data);
H5Pclose(plist_id);
```

Default write mode for datasets is individual
HDF5 example

```c
#include <hdf5.h>
int main(int argc, char *argv[]) {
  hid_t file_id, filespace_id, memoryspace_id, dset_id; // identifiers
  hsize_t dims[2] = {10, 5}; // two dimensional 10x5 dataspace
  hsize_t *maxdims = NULL; // fixed size dataspace
  int data[10][5]; // data to write
  // Calculate some data
  // ...
  // Create a file
  file_id = H5Fcreate("test.hdf5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
  // Create a dataspace
  filespace_id = H5Screate_simple(2, dims, maxdims);
  dset_id = H5Dcreate(file_id, "set1", H5T_NATIVE_INT, filespace_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
  // Write data to file
  memoryspace_id = H5Screate_simple(2, dims, maxdims);
  H5Dwrite(dset_id, H5T_NATIVE_INT, memoryspace_id, filespace_id, H5P_DEFAULT, data);
  // Clean up
  H5Sclose(filespace_id); H5Sclose(memoryspace_id); H5Dclose(dset_id); H5Fclose(file_id);
```

High level API

For simple, serial pre- and postprocessing tasks one can use the high level API of HDF5

```c
int ndims = 2;
  hsize_t dims[2] = {2, 3};
int data[6] = {0,1,2,3,4,5};
hid_t file_id;
file_id = H5Fcreate("data.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
H5LTmake_dataset(file_id, "/data", 2, dims, H5T_NATIVE_INT, data);
```

Summary

- HDF5 provides hierarchical data format for multidimensional array data
  - Files, datasets, dataspaces, selections
- Parallel IO can be performed by defining proper property lists for file creation/opening and dataset transfer
- Each process can work on part of the dataset
- HDF5 files are portable and can be analyzed also with several external programs
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_INITIALIZATION_ERROR(ierr)
MPI_COMM_SIZE(comm, size, ierr)
MPI_COMM_RANK(comm, rank, ierr)
MPI_BARRIER(comm, ierr)
MPI_FINALIZE(ierr)

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, ierr)

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, ierr)
### MPI datatypes

<table>
<thead>
<tr>
<th>MPI type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_INT</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
</tbody>
</table>

### MPI datatypes

<table>
<thead>
<tr>
<th>MPI type</th>
<th>Fortran type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td>REAL*8 (nonstandard)</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
</tbody>
</table>

### Combined send & receive

#### C/C++ binding

```c
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

```fortran
MPI_Bcast(buffer, count, datatype, root, comm, status)
```

### MPI_Bcast

#### C & Fortran bindings

```c
int MPI_Bcast(void* buffer, int count, MPI_data type datatype, 
               int root, MPI_Comm comm)
```

```fortran
MPI_BCAST(buffer, count, datatype, root, comm, ierror)
```

### MPI_Bcast

#### C/C++ binding

```c
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 )
```
MPI_Scatter

C & Fortran bindings

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

```fortran
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
```

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

MPI_Gather

C and Fortran bindings

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

```fortran
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
```

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

MPI_Gatherv

C and Fortran bindings

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

```fortran
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm, ierror)
```

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

C and Fortran bindings

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

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

type sendbuf(*), recvbuf(*)

integer count, datatype, op, root, comm, ierror

---

Available reduction operations (argument op)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Max value</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Min value</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Max value + location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Min value + location</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bytewise AND</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bytewise OR</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bytewise XOR</td>
</tr>
</tbody>
</table>

---

MPI_Allreduce

C & Fortran bindings

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

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

type :: sendbuf(*), recvbuf(*)

integer :: count, datatype, op, comm, ierror

---

MPI_Allgather

C & Fortran bindings

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

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

type :: sendbuf(*), recvbuf(*)

integer :: sendcount, recvcount, sendtype, recvtype, comm, ierror
MPI_Reduce_scatter

C & Fortran bindings

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

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

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

MPI_Alltoall

C & Fortran bindings

```c
int MPI_Alltoall(void* sendbuf, void* recvbuf, 
int sendcount, MPI_datatype sendtype, void* ... comm, ierror)
```

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

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

MPI_Alltoallv

C & Fortran bindings

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

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

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

Non-blocking send

C/C++ binding

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

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

C/C++ binding

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

```fortran
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
```
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
Non-blocking receive

C/C++ binding

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

Fortran binding

```fortran
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
```

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

Wait for non-blocking operations

C/C++ binding

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

Fortran binding

```fortran
MPI_WAIT(REQUEST, STATUS, IERROR)
```

INTEGER :: REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Creating a communicator

C and Fortran bindings

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

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

integer :: comm, color, key, newcomm, rc

Return code values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>No error; MPI routine completed successfully.</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>Invalid communicator. A common error is to use a null communicator in a call</td>
</tr>
<tr>
<td>MPI_ERR_INTERN</td>
<td>This error is returned when some part of the implementation is unable to acquire memory.</td>
</tr>
</tbody>
</table>
Communicator manipulation

C and Fortran bindings

```c
int MPI_Comm_compare ( MPI_Comm comm1, MPI_Comm comm2, int result )
```

```fortran
integer :: comm1, comm2, result, rc
```

```c
int MPI_Comm_dup ( MPI_Comm comm, MPI_Comm newcomm )
```

```fortran
integer :: comm, newcomm, rc
```

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

```fortran
integer :: comm, rc
```

Creating a communication topology

C and Fortran bindings

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

```fortran
integer :: old_comm, ndims, dims(:), comm_cart, rc
```

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

```fortran
integer :: comm, direction, displ, source, dest, rc
```

Ranks and coordinates

C and Fortran bindings

```c
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdim, int *coords)
```

```fortran
integer :: comm, rank, maxdim, coords(:), rc
```

```c
int MPI_Cart_rank(MPI_Comm comm, int *coords, int rank)
```

```fortran
integer :: comm, coords(:), rank, rc
```

Communication in a topology

C and Fortran bindings

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

```fortran
integer :: comm, direction, displ, source, dest, rc
```
C interfaces for datatype routines

```c
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)
```

Fortran interfaces for datatype routines

```fortran
integer :: type, rc
mpi_type_commit(type, rc)
integer :: count, oldtype, newtype, rc
mpi_type_free(type, ... newtype, rc)
integer :: ndims, order, oldtype, newtype, rc
integer, dimension(ndims) :: sizes, subsizes, starts
```

One-sided communication: window

C and Fortran bindings

```c
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win, rc)
    <type> :: base(*)
    integer(kind=MPI_ADDRESS_KIND) :: size
    integer :: disp_unit, info, comm, win, rc
```

```fortran
MPI_Win_fence(int assert, MPI_Win win)
MPI_WIN_FENCE(assert, win, rc)
```

Assert values

```
0, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED
```
One-sided communication: put

C and Fortran bindings

```c
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win *win)
```

```fortran
MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, rc)
```

One-sided communication: accumulate

C and Fortran bindings

```c
int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win *win)
```

```fortran
MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, rc)
```

One-sided communication: get

C and Fortran bindings

```c
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win *win)
```

```fortran
MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, rc)
```

C interfaces to MPI I/O routines

```c
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
```

```c
int MPI_File_close(MPI_File *fh)
```

```c
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
```

```c
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

```c
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

```c
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```

```c
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```
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_set_view(fh, disp, etype, filetype, datarep, info)

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)