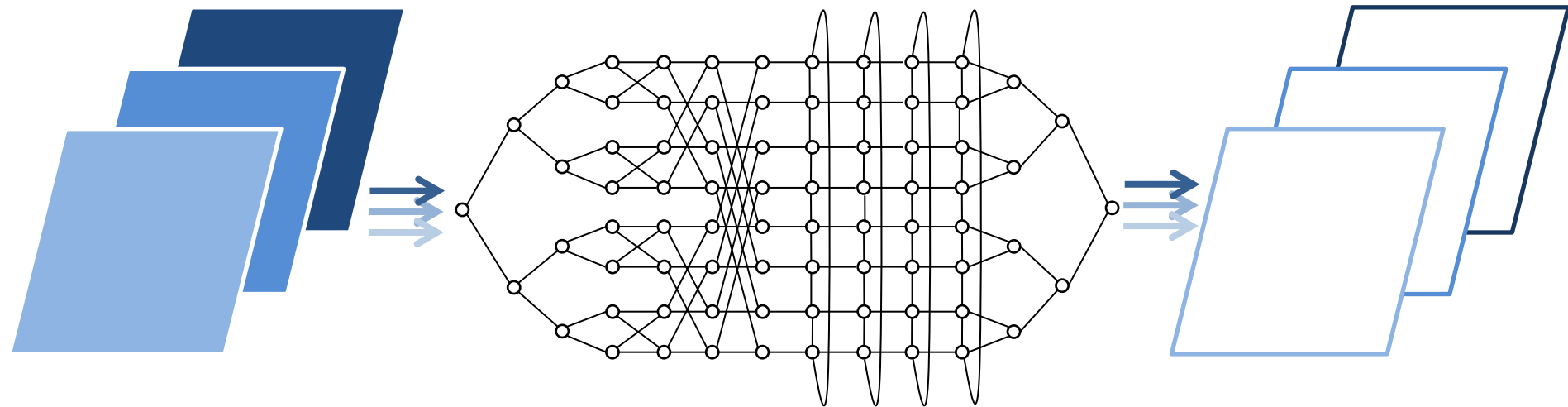


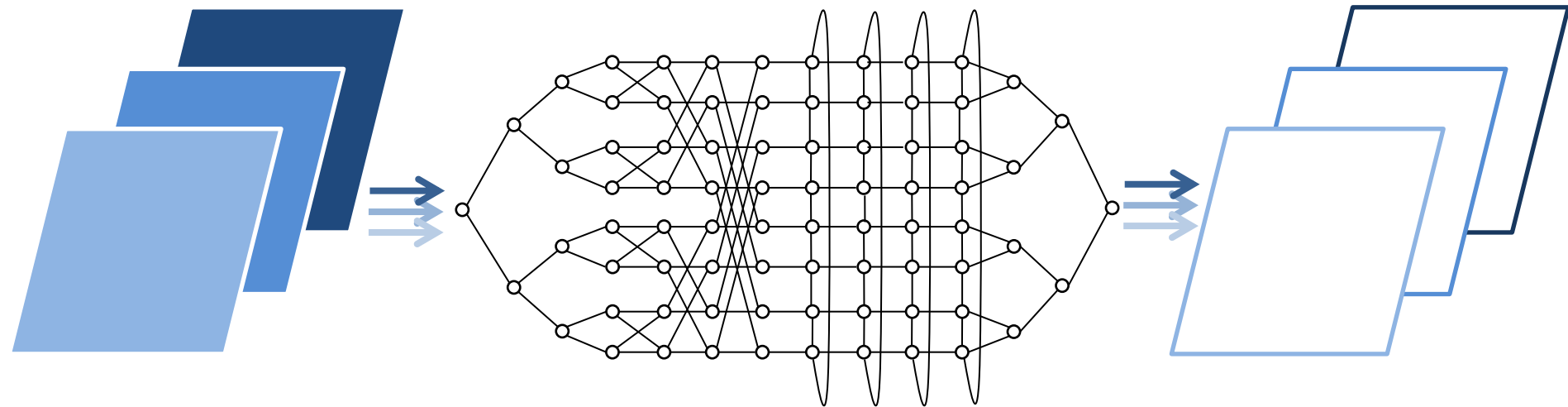
Programmation Parallèle

MPI: Message Passing Interface

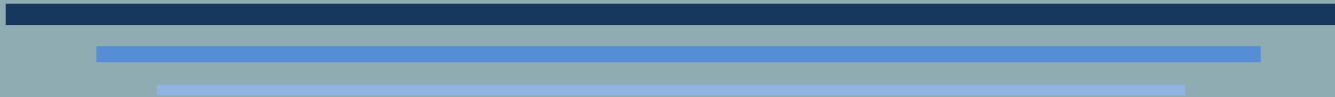


Manipulating MPI structures

ENSIIE-HPC/BigData-PP-IIP-Lecture 3



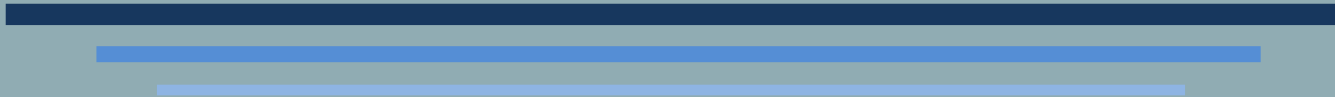
MPI PROCESS



MPI Process

- “MPI process” is a term defined and used in the MPI API
- In the MPI standard, an MPI Process **IS NOT** an OS process
 - You may implement it with a thread
- MPI process basically just means “MPI rank”
 - with a context local to the MPI rank

COMMUNICATORS



Communicators

- A communicator encapsulates the following concepts:
 - Contexts of communication,
 - Groups of processes,
 - Virtual topologies,
 - Attribute caching,
- Context provides the ability to have separate safe “universes” of message-passing in MPI.
 - Different libraries working on different communicators will not interact with each other, at MPI runtime level
 - They will still use the same network and may impact the other library message passing performance

Communicators

- Group of processes defines an ordered collection of processes, each with a rank
 - The group defines a scope for process names in communications
 - A rank number is only valid in the corresponding communicator
 - An MPI process may have different rank number per communicator
- Virtual topology
 - To be discussed in Lecture 4
- Attributes define the local information that the user or library has added to a communicator for later reference.
 - Ex: hints for algorithm to use, not using user-defined data type...

Duplication

- First function to manipulate communicators → duplication

- Prototype:

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

- Useful when designing a library
 - Allow usage of same communicator
 - Avoid deadlock with pending communications

Split



- Possibility to split a communicator into multiple subgroups

- Prototype:

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

- Effect

- Create disjoint subgroups (one per color value)
- Within a subgroup, process are ranked according to key value

- Useful to adapt work and exploit different parallelism

Split

MPI_COMM_WORLD

r_1

r_0

r_3

r_2

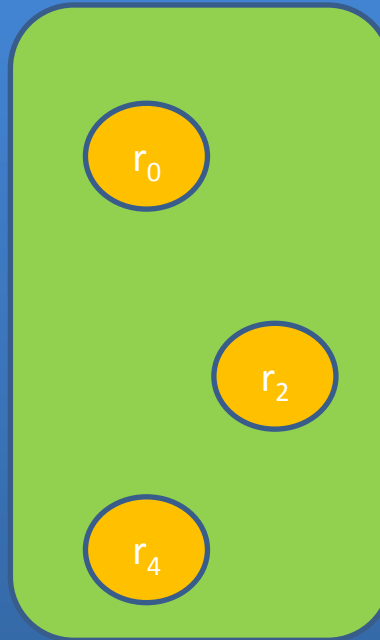
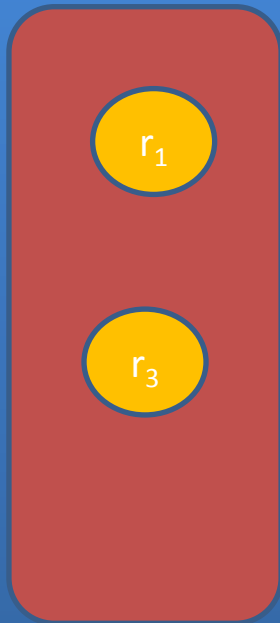
r_4

```
MPI_Comm_split(  
MPI_COMM_WORLD,  
r%2, r, &c)
```

Split



MPI_COMM_WORLD



```
MPI_Comm_split(  
MPI_COMM_WORLD,  
r%2, r, &c)
```

Creating Communicators



- Create a new communicator from an old communicator and a group
- Prototype:

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```
- Group argument must be a valid subset of the old *comm* group
- Each process in *comm* must call the function

Creating Communicators



- Exist a function to be called only from the processes to be included in the new comm *newcomm*

- Prototype:

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group  
group, int tag, MPI_Comm *newcomm)
```

- Tag allows to identify MPI_Comm_create_group calls in a multithreaded environment
 - Does not interfere with communication tags

Comparing Communicators



- Possibility to compare communicators
- Prototype:

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```
- Result
 - MPI_IDENT: comm1 and comm2 are the same object (group, context)
 - MPI_CONGRUENT: same group with same rank order (not context)
 - MPI_SIMILAR: same groupe (not same rank order, not same context)
 - MPI_UNEQUAL

Inter-communicators



- What we discussed so far are intra-communicators (one group, one context)
- However, when an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module.
- Inter-communicators are defined to exchange messages between multiple groups

Inter-communicators



- **Prototype:**

```
int MPI_Intercomm_create(MPI_Comm local_comm, int  
local_leader, MPI_Comm peer_comm, int  
remote_leader, int tag, MPI_Comm *newintercomm)
```

- Use two intra-comms to create an inter-comm
- Processes should provide identical local_comm and local_leader arguments within each group.
- Collective call over the union of both groups

Inter-communicators

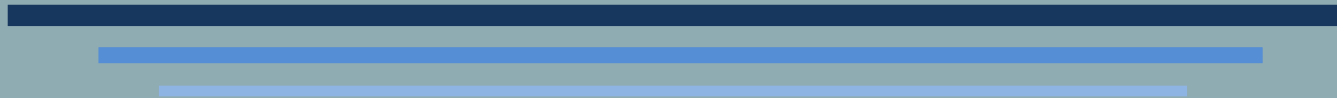


- Possible to merge the groups of an inter-communicator to create an intra-communicator:

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,  
MPI_Comm *newintracomm)
```

- High argument: fix the order in the new intra-comm of the ranks of the two inter-comms
 - If high=true for all processes, arbitrary order
 - If high=false for group1 and high=true for group2, then ranks order will be ranks from group1 then ranks from group2
- Useful because some operations are not possible with inter-communications

USER-DEFINED DATATYPES



Homogeneous blocks

Derived Datatypes



- Communication mechanisms studied to this point allow send/recv of a *contiguous buffer of identical elements* of predefined datatypes.
- Often want to send *non-homogenous* elements (structure) or chunks that are not *contiguous in memory*
- MPI allows *derived datatypes* for this purpose.

Homogeneous block



- MPI Type contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
 - IN count (replication count)
 - IN oldtype (base data type)
 - OUT newtype (handle to new data type)
- Creates a new type which is simply a replication of oldtype into contiguous locations

Committing a derived datatype



- Every datatype constructor returns an *uncommitted* datatype. Think of commit process as a compilation of datatype description into efficient internal form.
- Must call *MPI Type commit*(&datatype).
- Once committed, a datatype can be repeatedly reused.
- If called more than once, subsequent call has no effect.

Freeing a derived datatype



- Call to *MPI Type free* (&datatype) sets the value of datatype to `MPI_DATATYPE_NULL`.
- Not possible to use the derived datatype anymore
- Datatypes that were derived from the defined datatype are unaffected.

MPI_Type_contiguous example



```
/* create a type which describes a line of ghost cells */  
/* buf[1..nxl] set to ghost cells */  
int nxl;  
MPI_Datatype ghosts;  
  
MPI_Type_contiguous (nxl, MPI_DOUBLE, &ghosts);  
MPI_Type_commit(&ghosts)  
MPI_Send (buf, 1, ghosts, dest, tag, MPI_COMM_WORLD);  
..  
..  
MPI_Type_free(&ghosts);
```






Spaced homogeneous blocks



- MPI Type vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
 - IN count (number of blocks)
 - IN blocklength (number of elements per block)
 - IN stride (spacing between start of each block, measured in # elements)
 - IN oldtype (base datatype)
 - OUT newtype (handle to new type)
- Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of oldtype with a stride that is multiple of extent of old type.

MPI_Type_vector example

```
MPI_Datatype mytype;  
MPI_Type_vector(3, 4, 5, MPI_DOUBLE, &mytype);  
MPI_Type_commit(&mytype);
```

- oldtype = MPI_DOUBLE 
- blocklength = 4 
- count = 3 
- stride = 5 
- **mytype** : 

Spaced homogeneous blocks



- MPI_Type_create_hvector (int count, int blocklength, MPI_Aint stride, MPI_Datatype old, MPI_Datatype *new)
 - IN count (number of blocks)
 - IN blocklength (number of elements/block)
 - IN stride (number of bytes between start of each block)
 - IN old (old datatype)
 - OUT new (new datatype)
- Same as MPI_Type_vector, except that stride is given in bytes rather than in elements
 - h stands for heterogeneous.

USER-DEFINED DATATYPES



Heterogeneous blocks and types

Heterogeneous space and block length



- `MPI_Type_indexed` (`int count`, `int *array_of_blocklengths`, `int *array_of_displacements`, `MPI_Datatype oldtype`, `MPI_Datatype *newtype`);
 - IN `count` (number of blocks)
 - IN `array_of_blocklengths` (number of elements/block)
 - IN `array_of_displacements` (displacement for each block, measured as number of elements)
 - IN `oldtype`
 - OUT `newtype`
- Displacements between successive blocks may not be equal.
- Block lengths may not be equal.

MPI_Type_indexed example

```
int blocklength[3] = {2,3,1}
```

```
int displacement[3] = {0,3,8}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_vector(3, &blocklength, &displacement, MPI_DOUBLE_2,  
&mytype);
```

```
MPI_Type_commit(&mytype);
```

■ oldtype = MPI_DOUBLE_2



■ blocklength = 2,3,1





■ stride = 0,3,8



■ **mytype** :





Heterogeneous space and block length



- `MPI_Type_create_hindexed` (int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype);
 - IN count (number of blocks)
 - IN array_of_blocklengths (number of elements/block)
 - IN array_of_displacements (displacement for each block, measured as number of elements)
 - IN oldtype
 - OUT newtype
- Same as `MPI_Type_indexed`, except that stride is given in bytes rather than in elements

Heterogeneous space

- 
- 
- `int MPI_Type_create_indexed_block(int count, int blocklength, const int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)`
 - `int MPI_Type_create_hindexed_block(int count, int blocklength, const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)`
 - Same as `MPI_Type_indexed` and `MPI_Type_create_hindexed` but with same size for all blocks
 - Still possible to have different spacing between blocks

Upper-triangular transfer



```
double a[100][100];
Int disp[100], blocklen[100], i, dest, tag;
MPI_Datatype upper;

/* compute start and size of each row */
for (i = 0; i < 100; ++i){
    disp[i] = 100*i + i;
    blocklen[i] = 100 - i;
}

MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
```

Fully heterogeneous type



- `MPI_Type_create_struct` (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);
 - IN count (number of blocks)
 - IN array_of_blocklengths (number of elements in each block)
 - IN array_of_displacements (byte displacement of each block)
 - IN array_of_types (type of elements in each block)
 - OUT newtype
- Most general type constructor.
- Further generalizes `MPI_Type_create_hindexed`
- Allows each block to consist of replications of different datatypes.

MPI_Type_create_struct example

```
MPI_Datatype types[3] = {DOUBLE, INT, SHORT}
```

```
int blocklength[3] = {2,2,5}
```

```
int displacement[3] = {0,14,26}
```

```
MPI_Datatype mytype;
```

```
MPI_Type_vector(3, &blocklength, &displacement, &types, &mytype);
```

```
MPI_Type_commit(&mytype);
```

- oldtype = double, int, short



- blocklength = 2,2,5



- stride (bytes) = 0,14,26



- mytype :**



Example: struct of basic types



```
Struct Partstruct{  
    char class;  
    double d[6];  
    char b[7];  
}
```

```
Struct Partstruct  particle[1000];  
Int                dest, tag;  
MP_Comm           comm;
```

```
MPI_Datatype particletype;  
MPI_Datatype type[3]    = {MPI_CHAR, MPI_DOUBLE, MPI_CHAR};  
int                blocklen[3] = {1, 6, 7};  
MPI_Aint          disp[3]    = {0, sizeof(double), 7*sizeof(double)};
```


```
MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);  
MPI_Type_commit(&Particletype);  
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
```

Subarray

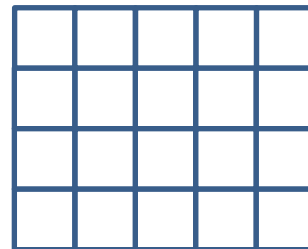
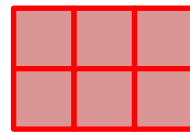
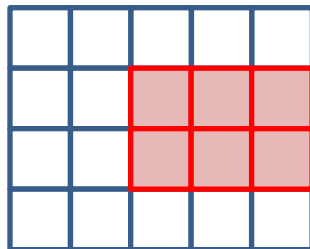
- `MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype);`
 - IN ndims (number of dimensions of the array)
 - IN array_of_sizes (sizes for each dimension of original array)
 - IN array_of_subsizes (sizes for each dimension of subarray)
 - IN order (type of order, C or FORTRAN, row major or column major)
 - IN oldtype (type of element of the original array)
 - OUT newtype
- Exists other derived datatypes creatin function targeting arrays
- Ex: `MPI_Type_create_subarray`
- Build a datatype to capture subarray(s) in a linearized multi-dimensionnal array

MPI_Type_create_subarray example

```
MPI_Datatype subarray3x2;  
int array_of_sizes[2] = {5,4};  
int array_of_subsizes[2] = {3,2};  
int arrays_of_starts[2] = {2,1};  
MPI_type_create_subarray(NDIMS, array_of_sizes, array_of_subsizes,  
array_of_starts, MPI_ORDER_C, MPI_FLOAT, &subarray3x2);  
MPI_TYPE_COMMIT(&subarray3x2);
```

- oldtype = MPI_FLOAT 
- array_of_subsizes[2] = {3,2};
- array_of_sizes[2] = {5,4};
- arrays_of_starts[2] = {2,1};

■ **mytype :**



MPI_Type_create_subarray example



```
#define NDIMS 2
MPI_Datatype subarray3x2;
int array_of_sizes[NDIMS], array_of_subsizes[NDIMS], arrays_of_starts[NDIMS];



array_of_sizes[0] = 5; array_of_sizes[1] = 4;
array_of_subsizes[0] = 3; array_of_subsizes[1] = 2;
array_of_starts[0] = 2; array_of_starts[1] = 1;
order = MPI_ORDER_C;

MPI_type_create_subarray(NDIMS, array_of_sizes, array_of_subsizes, array_of_starts,
order, MPI_FLOAT, &subarray3x2);

MPI_TYPE_COMMIT(&subarray3x2);

MPI_Send(&x[0][0], 1, subarray3x2, ...);
```

Alignment

- Be very careful about data alignment
- Data alignment may change the extent and offsets of a derived datatypes
- Ex: struct with one double and two ints
 - If ints are aligned on 4B and double on 8B
- Struct1 {int a; int b; double d;} 
 - Extent: 16B, array of 10 struct1: 160B
- Struct2 {int a; double d; int b;} 
 - Extent: 20B, array of 10 struct2: 200B
 - Necessary to add the 4B displacement to build a valid datatype

Back to create_struct example





```
Struct Partstruct{  
    char class;  
    double d[6];  
    char b[7];  
}
```

```
Struct Partstruct  particle[1000];  
Int                dest, tag;  
MP_Comm           comm;
```

```
MPI_Datatype particletype;  
MPI_Datatype type[3]    = {MPI_CHAR, MPI_DOUBLE, MPI_CHAR};  
int                blocklen[3] = {1, 6, 7};  
MPI_Aint          disp[3]    = {0, sizeof(double), 7*sizeof(double)};
```

```
MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);  
MPI_Type_commit(&Particletype);  
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
```

Alignment

- 
- 
- Note, this example assumes that a double is double-word aligned. If double's are single-word aligned, then **disp** would be initialized as

(0, sizeof(int), sizeof(int) + 6*sizeof(double))

- *MPI_Get_address* allows us to write more generally correct code.
- *MPI_Get_address* (void *location, MPI_Aint *address);
 - IN location (location in caller memory)
 - OUT address (address of location)

Size of a datatype



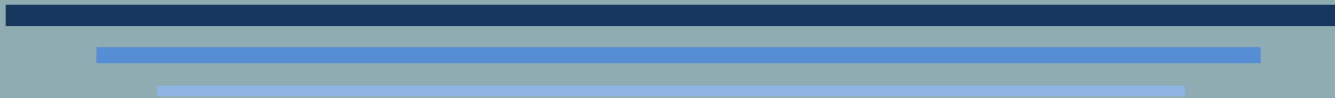
- MPI_Type_size(MPI_Datatype datatype, int *size)
 - IN datatype (datatype)
 - OUT size (datatype size)
- Returns number of bytes actually occupied by datatype, excluding strided areas.

“Real” size of a datatype



- *MPI Type get extent* (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
 - IN datatype (datatype you are querying)
 - OUT lb (lower bound of datatype)
 - OUT extent (extent of datatype)
- Returns the lower bound and extent of datatype.
- Upper bound is lower_bound + extent

USER-DEFINED OPERATORS



User-Defined Operations

```
int MPI_Op_create (  
    MPI_User_function *user_fn(in),  
    int commute(in),  
    MPI_Op *op(out),  
);
```

Function pointer **user_fn**.

This function should allow **associative** reduction of an element vector (the number of elements and data type are given as arguments).

The following prototype has to be followed:
`void (*f)(void* invec, void* inoutvec, int *len, MPI_Datatype *datatype);`

User-Defined Operations



```
int MPI_Op_create (  
    MPI_User_function *user_fn(in) ,  
    int commute(in) ,  
    MPI_Op *op(out) ,  
);
```

Does reduction commute?

If yes, runtime may optimize
reduction performance

User-Defined Operations

- Example of user-defined operation

```
void user_add( int *invec, int *inoutvec, int *len,
  MPI_Datatype *dtype ) {
  int i;
  for ( i = 0 ; i < *len ; i++ )
    inoutvec[i] += invec[i];
}
```

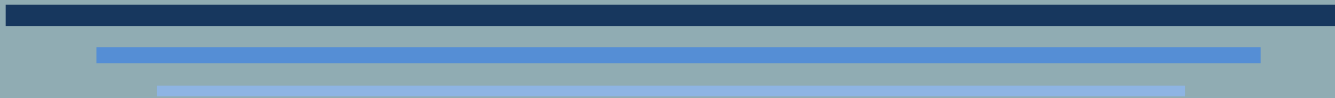
- Creation of this operation

```
MPI_Op_create( (MPI_User_function *)user_add, 1,
  &op );
```



- Free operation

```
MPI_Op_free(op);
```


HYBRID PROGRAMMING WITH THREADS AND SHARED MEMORY



MPI and Threads

- 
- 
- MPI describes parallelism between MPI processes, with separate address spaces
 - *Thread* parallelism provides a shared-memory model within a process
 - OpenMP and Pthreads are common models
 - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
 - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

Programming for Multicore



- Almost all chips are multicore these days
- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network
- Common options for programming such clusters
 - All MPI
 - MPI between processes both within a node and across nodes
 - MPI internally uses shared memory to communicate within a node
 - MPI + OpenMP
 - Use OpenMP within a node and MPI across nodes
 - MPI + Pthreads
 - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as “hybrid programming”

MPI's Four Levels of Thread Safety



- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
 - `MPI_THREAD_SINGLE`: only one thread exists in the application
 - `MPI_THREAD_FUNNELED`: multithreaded, but only the main thread makes MPI calls (the one that called `MPI_Init_thread`)
 - `MPI_THREAD_SERIALIZED`: multithreaded, but only one thread *at a time* makes MPI calls
 - `MPI_THREAD_MULTIPLE`: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- MPI defines an alternative to `MPI_Init`
 - `MPI_Init_thread(requested, provided)`
 - *Application indicates what level it needs; MPI implementation returns the level it supports*

MPI+OpenMP



- **MPI_THREAD_SINGLE**

- There is no OpenMP multithreading in the program.

- **MPI_THREAD_FUNNELED**

- All of the MPI calls are made by the master thread. i.e. all MPI calls are
 - *Outside OpenMP parallel regions, or*
 - *Inside OpenMP master regions, or*
 - *Guarded by call to MPI_Is_thread_main MPI call.*
 - (same thread that called MPI_Init_thread)

- **MPI_THREAD_SERIALIZED**

```
#pragma omp parallel
```

```
...
```

```
#pragma omp critical
```

```
{
```

```
...MPI calls allowed here...
```

```
}
```

- **MPI_THREAD_MULTIPLE**

- Any thread may make an MPI call at any time

MPI_THREAD_MULTIPLE

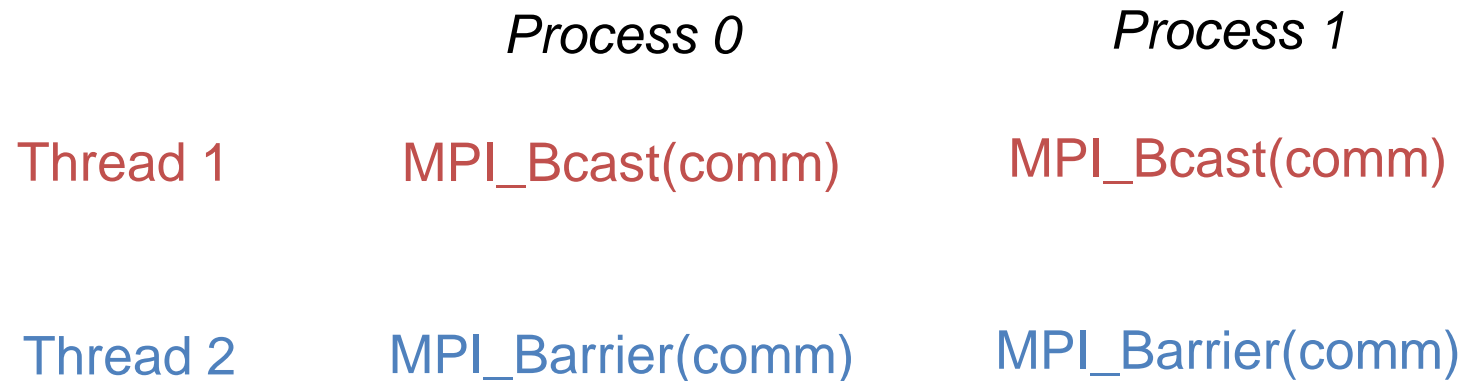


- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
 - e.g., accessing an info object from one thread and freeing it from another thread
- User must ensure that collective operations on the same communicator are correctly ordered among threads
 - e.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator

Threads and MPI

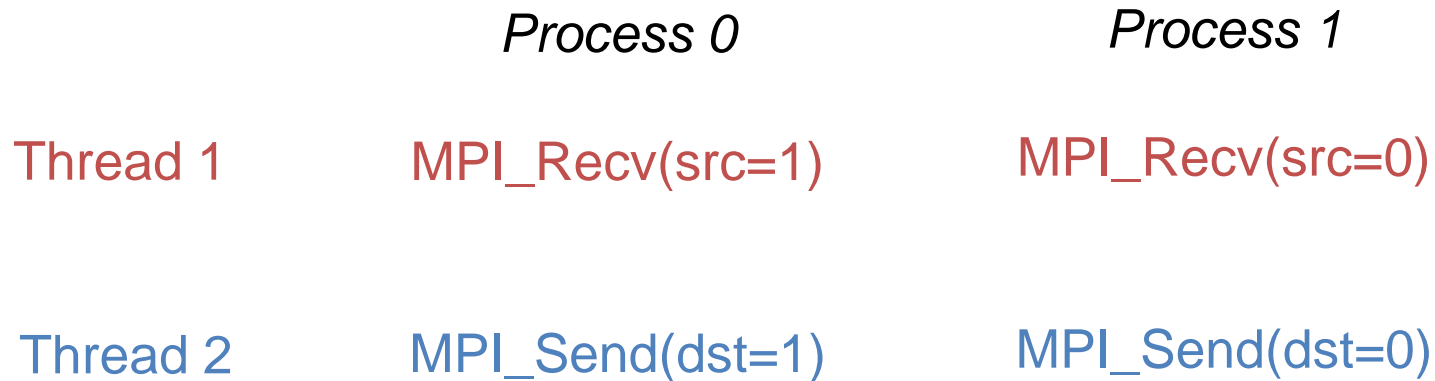
- An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support `MPI_THREAD_MULTIPLE`
- A program that calls `MPI_Init` (instead of `MPI_Init_thread`) should assume that only `MPI_THREAD_SINGLE` is supported
- *A threaded MPI program that does not call `MPI_Init_thread` is an incorrect program (common user error we see)*

An Incorrect Program



- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

A Correct Example



- An implementation must ensure that the above example never deadlocks for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

The Current Situation



- All MPI implementations support `MPI_THREAD_SINGLE` .
- They probably support `MPI_THREAD_FUNNELED` even if they don't admit it.
 - Does require thread-safe malloc
 - Probably OK in OpenMP programs
- Many (but not all) implementations support `THREAD_MULTIPLE`
 - Hard to implement efficiently though (lock granularity issue)
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need `FUNNELED`
 - So don't need “thread-safe” MPI for many hybrid programs
 - But watch out for Amdahl's Law!

Performance with Thread Multiple



- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- Synchronization: bad for performances