Message Passing Interface

George Bosilca
bosilca@cs.utk.edu

MPI 1 & 2

- MPI 1
  - MPI Datatype
  - Intra/Inter Communicators
- MPI 2
  - Process management
  - Connect/Accept
  - MPI I/O

MPI Derived Datatypes

MPI Datatypes

- Abstract representation of underlying data
  - Handle type: MPI_Datatype
- Pre-defined handles for intrinsic types
  - E.g., C: MPI_INT, MPI_FLOAT, MPI_DOUBLE
  - E.g., Fortran: MPI_INTEGER, MPI_REAL
  - E.g., C++: MPI::BOOL
- User-defined datatypes
  - E.g., arbitrary / user-defined C structs

MPI Data Representation

- Multi platform interoperability
- Multi languages interoperability
  - Is MPI_INT the same as MPI_INTEGER?
  - How about MPI_INTEGER[1,2,4,8]?
- Handling datatypes in Fortran with MPI_SIZEOF and MPI_TYPE_MATCH_SIZE

Multi-Platform Interoperability

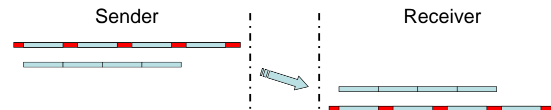
- Different data representations
  - Length 32 vs. 64 bits
  - Endianness conflict
- Problems
  - No standard about the data length in the programming languages (C/C++)
  - No standard floating point data representation
    - IEEE Standard 754 Floating Point Numbers
      - Subnormals, infinities, NANs …
    - Same representation but different lengths

How About Performance?

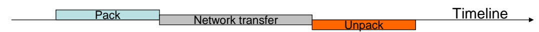
- Old way
  - Manually copy the data in a user pre-allocated buffer, or
  - Manually use MPI_PACK and MPI_UNPACK
- New way
  - Trust the [modern] MPI library
  - High performance MPI datatypes

How About Performance?

- Pack/Unpack Approach

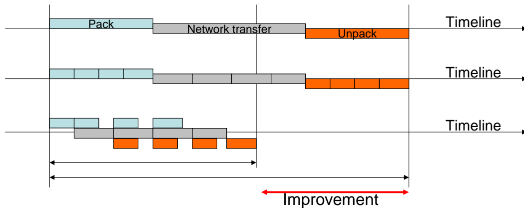


- How can we increase the performance?
  - 3 distinct steps: pack, network transfer, unpack
  - No computation / communication recovery



Improving Performance

- Increasing the computation / communication recovery
  - Split the computations in small slices



Improving Performance

- Raising others questions:
  - How to adapt to the network layer?
  - How to support RDMA operations?
  - How to handle heterogeneous communications?
  - How to split the data pack / unpack?
- Who handles all this?
  - MPI implementation can solve these problems
  - User-level applications cannot

MPI Datatypes

- MPI uses “datatypes” to:
  - Efficiently represent and transfer data
  - Minimize memory usage
- Even between heterogeneous systems
  - Used in most communication functions (MPI_SEND, MPI_RECV, etc.)
  - And file operations
- MPI contains a large number of pre-defined datatypes

Some of MPI’s Pre-Defined Datatypes

MPI_Datatype	C datatype	Fortran datatype
MPI_CHAR	signed char	CHARACTER
MPI_SHORT	signed short int	INTEGER*2
MPI_INT	signed int	INTEGER
MPI_LONG	signed long int	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	REAL
MPI_DOUBLE	double	DOUBLE PRECISION
MPI_LONG_DOUBLE	long double	DOUBLE PRECISION*8

Datatype Matching

- Two requirements for correctness:
  - Type of each data in the send / recv buffer matches the corresponding type specified in the sending / receiving operation
  - Type specified by the sending operation has to match the type specified for receiving operation
- Issues:
  - Matching of type of the host language
  - Match of types at sender and receiver

Datatype Conversion

- “Data sent = data received”
- 2 types of conversions:
  - Representation conversion: change the binary representation (e.g., hex floating point to IEEE floating point)
  - Type conversion: convert from different types (e.g., int to float)

➔ Only representation conversion is allowed

Datatype Conversion

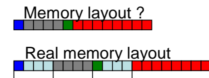
```
if( my_rank == root )
   MPI_Send( ai, 1, MPI_INT, … )
else
   MPI_Recv( ai, 1, MPI_INT, … )
```

```
if( my_rank == root )
   MPI_Send( ai, 1, MPI_INT, … )
else
   MPI_Recv( af, 1, MPI_FLOAT, … )
```

Memory Layout

- How to describe a memory layout ?

```
struct {
  char c1;
  int i;
  char c2;
  double d;
}
```



Using iovecs (list of addresses)
<pointer to memory, length>
<baseaddr c1, 1>,<addr_of_i, 4>, <addr_of_c2, 1>, <addr_of_d, 8>
- Waste of space
- Not portable …

Using displacements from base addr
<displacement, length>
<0, 1>, <4, 4>, <8, 1>, <12, 8>
- Sometimes more space efficient
- And nearly portable
- What are we missing ?

Datatype Specifications

- Type signature
  - Used for message matching
    $\{ type_0, type_1, \ldots, type_n \}$
- Type map
  - Used for local operations
    $\{ (type_0, disp_0), (type_1, disp_1), \ldots, (type_n, disp_n) \}$

➔ It’s all about the memory layout

User-Defined Datatypes

- Applications can define unique datatypes
  - Composition of other datatypes
  - MPI functions provided for common patterns
    - Contiguous
    - Vector
    - Indexed
    - …

➔ Always reduces to a type map of pre-defined datatypes

Handling datatypes

- MPI impose that all datatypes used in communications or file operations should be committed.
  - Allow MPI libraries to optimize the data representation

MPI_Type_commit(MPI_Datatype*)
MPI_Type_free(MPI_Datatype*)

- All datatypes used during intermediary steps, and never used to communicate does not need to be committed.

Contiguous Blocks

- Replication of the datatype into contiguous locations.

MPI_Type_contiguous(3, oldtype, newtype)

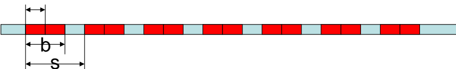
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count(positive integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Vectors

- Replication of a datatype into locations that consist of equally spaced blocks

MPI_Type_vector(7, 2, 3, oldtype, newtype)



MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	number of blocks (positive integer)
IN	blocklength	number of elements in each block (positive integer)
IN	stride	number of elements between start of each block (integer)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

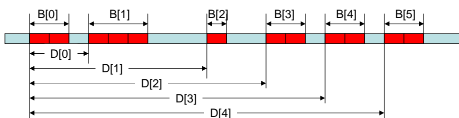
- Replication of an old datatype into a sequence of blocks, where each block can contain a different number of copies and have a different displacement

MPI_TYPE_INDEXED(count, array_of_blocks, array_of_displs, oldtype, newtype)

IN	count	number of blocks (positive integer)
IN	a_of_b	number of elements per block (array of positive integer)
IN	a_of_d	displacement of each block from the beginning in multiple multiple of oldtype (array of integers)
IN	oldtype	old datatype (MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Indexed Blocks

array_of_blocklengths[] = { 2, 3, 1, 2, 2, 2 }
array_of_displs[] = { 0, 3, 10, 13, 16, 19 }
MPI_Type_indexed(6, array_of_blocklengths,
array_of_displs, oldtype, newtype)



Datatype Composition

- Each of the previous functions are the super set of the previous

CONTIGUOUS < VECTOR < INDEXED

- Extend the description of the datatype by allowing more complex memory layout
  - Not all data structures fit in common patterns
  - Not all data structures can be described as compositions of others

“H” Functions

- Displacement is not in multiple of another datatype
- Instead, displacement is in bytes
  - MPI_TYPE_HVECTOR
  - MPI_TYPE_HINDEX
- Otherwise, similar to their non-”H” counterparts

Arbitrary Structures

- The most general datatype constructor
- Allows each block to consist of replication of different datatypes

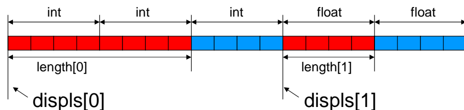
MPI_TYPE_CREATE_STRUCT(count, array_of_blocklength, array_of_displs, array_of_types, newtype)

IN	count	number of entries in each array (positive integer)
IN	a_of_b	number of elements in each block (array of integers)
IN	a_of_d	byte displacement in each block (array of Aint)
IN	a_of_t	type of elements in each block (array of MPI_Datatype handle)
OUT	newtype	new datatype (MPI_Datatype handle)

Arbitrary Structures

```
struct {
  int i[3];
  float f[2];
} array[100];
```

```
Array_of_lengths[] = { 2, 1 };
Array_of_displs[] = { 0, 2*sizeof(int) };
Array_of_types[] = { MPI_INT, MPI_FLOAT };
MPI_Type_struct( 2, array_of_lengths,
  array_of_displs, array_of_types, newtype );
```



displs[0] displs[1]

Portable Vs. non portable

- The portability refer to the architecture boundaries
- Non portable datatype constructors:
  - All constructors using byte displacements
  - All constructors with H<type>, MPI_Type_struct
- Limitations for non portable datatypes
  - One sided operations
  - Parallel I/O operations

MPI_GET_ADDRESS

- Allow all languages to compute displacements
  - Necessary in Fortran
  - *Usually* unnecessary in C (e.g., “&foo”)

MPI_GET_ADDRESS(location, address)

IN	location	location in the caller memory (choice)
OUT	address	address of location (address integer)

And Now the Dark Side…

- Sometimes more complex memory layout have to be expressed

Lower-Bound and Upper-Bound Markers

- Define datatypes with “holes” at the beginning or end
- 2 pseudo-types: MPI_LB and MPI_UB
  - Used with MPI_TYPE_STRUCT

Typemap = { ($type_0$, $disp_0$), …, ($type_n$, $disp_n$) }

$$lb(Typemap) = \begin{cases} \text{Min}_j \, disp_j & \text{if no entry has type lb} \\ \text{min}_j(\{disp_j \text{ such that } type_j = lb) & \text{otherwise} \end{cases}$$

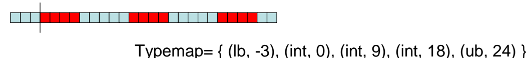
$$ub(Typemap) = \begin{cases} \text{Max}_j \, disp_j + sizeof(type_j) + align & \text{if no entry has type ub} \\ \text{Max}_j\{disp_j \text{ such that } type_j = ub\} & \text{otherwise} \end{cases}$$

MPI_LB and MPI_UB

```
displs = ( -3, 0, 6 )
blocklengths = ( 1, 1, 1 )
types = ( MPI_LB, MPI_INT, MPI_UB )
MPI_Type_struct( 3, displs, blocklengths,
  types, type1 )
```

Typemap= { (lb, -3), (int, 0), (ub, 6) }

```
MPI_Type_contiguous( 3, type1, type2 )
```



Typemap= { (lb, -3), (int, 0), (int, 9), (int, 18), (ub, 24) }

MPI 2 Solution

- Problem with the way MPI-1 treats this problem: upper and lower bound can become messy, if you have derived datatype consisting of derived dataype consisting of derived datatype consisting of… and each of them has MPI_UB and MPI_LB set
- There is no way to erase LB and UB markers once they are set !!!
- MPI-2 solution: reset the extent of the datatype

MPI_Type_create_resized (MPI_Datatype datatype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype*newtype);

- Erases all previous lb und ub markers

True Lower-Bound and True Upper-Bound Markers

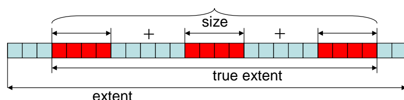
- Define the real extent of the datatype: the amount of memory needed to copy the datatype inside
- TRUE_LB define the lower-bound ignoring all the MPI_LB markers.

Typemap = { ($type_0$, $disp_0$), …, ($type_n$, $disp_n$) }

$true_lb(Typemap) = min_j \{ disp_j : type_j \text{ != } lb \}$

$true_ub(Typemap) = max_j \{ disp_j + sizeof(type_j) : type_j \text{ != } ub \}$

Information About Datatypes



MPI_TYPE_GET_{TRUE_}EXTENT(datatype, {true_}lb, {true_}extent)

IN	datatype	the datatype (MPI_Datatype handle)
OUT	{true_}lb	{true} lower-bound of datatype (MPI_AINT)
OUT	{true_}extent	{true} extent of datatype (MPI_AINT)

MPI_TYPE_SIZE(datatype, size)

IN	datatype	the datatype (MPI_Datatype handle)
OUT	size	datatype size (integer)

Decoding a datatype

- Sometimes is important to know how a datatype was created (eg. Libraries developers)
- Given a datatype can I determine how it was created ?
- Given a datatype can I determine what memory layout it describe ?

MPI_Type_get_enveloppe

```
MPI_Type_get_envelope ( MPI_Datatype datatype,
                        int *num_integers, int *num_addresses,
                        int *num_datatypes, int *combiner );
```

- The combiner field returns how the datatype was created, e.g.
  - MPI_COMBINER_NAMED: basic datatype
  - MPI_COMBINER_CONTIGUOS: MPI_Type_contiguous
  - MPI_COMBINER_VECTOR: MPI_Type_vector
  - MPI_COMBINER_INDEXED: MPI_Type_indexed
  - MPI_COMBINER_STRUCT: MPI_Type_struct
- The other fields indicate how large the integer-array, the datatype-array, and the address-array has to be for the following call to MPI_Type_get_contents

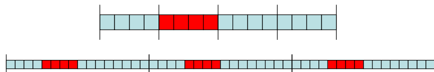
MPI_Type_get_contents

```
MPI_Type_get_contents ( MPI_Datatype datatype,
    int max_integer, int max_addresses, int max_datatypes,
    int *integers, int *addresses, MPI_Datatype *dts);
```

- Call is erroneous for a predefined datatypes
- If returned data types are derived datatypes, then objects are duplicates of the original derived datatypes. User has to free them using MPI_Type_free
- The values in the integers, addresses and datatype arrays are depending on the original datatype constructor

One Data By Cache Line

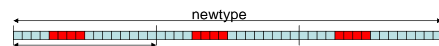
- Imagine the following architecture:
  - Integer size is 4 bytes
  - Cache line is 16 bytes
- We want to create a datatype containing the second integer from each cache line, repeated three times



- How many ways are there?

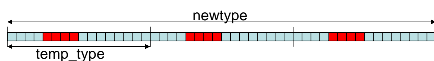
Solution 1

```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_INT, MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 0, 0, 0, 0 };
int array_of_lengths[] = { 1, 1, 1, 1 };
struct one_by_cacheline c[4];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1].int[1], &(array_of_displs[1]) );
MPI_Get_address( &c[2].int[1], &(array_of_displs[2]) );
MPI_Get_address( &c[3], &(array_of_displs[3]) );

for( i = 0; i < 4; i++ ) Array_of_displs[i] -= start;

MPI_Type_create_struct( 4, array_of_lengths,
                        array_of_displs, array_of_types, newtype )
```



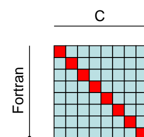
Solution 2

```
MPI_Datatype array_of_types[] = { MPI_INT, MPI_UB };
MPI_Aint start, array_of_displs[] = { 4, 16 };
int array_of_lengths[] = { 1, 1 };
struct one_by_cacheline c[2];

MPI_Get_address( &c[0], &(start) );
MPI_Get_address( &c[0].int[1], &(array_of_displs[0]) );
MPI_Get_address( &c[1], &(array_of_displs[1]) );

Array_of_displs[0] -= start;
Array_of_displs[1] -= start;
MPI_Type_create_struct( 2, array_of_lengths,
            array_of_displs, array_of_types, temp_type )
MPI_Type_contiguous( 3, temp_type, newtype )
```



Exercise

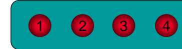
- Goals:
  - Create a datatype describing a matrix diagonal
  - What's different between C and Fortran ?



Intra and Inter Communicators

Groups

- A group is a set of processes
  - The group have a size
  - And each process have a rank
- Creating a group is a local operation
- Why we need groups
  - To make a clear distinction between processes
  - To allow communications in-between subsets of processes
  - To create intra and inter communicators …



Groups

- MPI_GROUP_*(group1, group2, newgroup)
  - Where * ∈ {UNION, INTERSECTION, DIFFERENCE}
  - Newgroup contain the processes satisfying the * operation ordered first depending on the order in group1 and then depending on the order in group2.
  - In the newgroup each process could be present only one time.
- There is a special group without any processes MPI_GROUP_EMPTY.

Groups

- group1 = {a, b, c, d, e}
- group2 = {e, f, g, b, a}
- Union
  - newgroup = {a, b, c, d, e, f, g}
- Difference
  - newgroup = {c, d}
- Intersection
  - newgroup = {a, b, e}

Groups

- MPI_GROUP_*(group, n, ranks, newgroup)
  - Where * ∈ {INCL, EXCL}
  - N is the number of valid indexes in the ranks array.
- For INCL the order in the result group depend on the ranks order
- For EXCL the order in the result group depend on the original order

Groups

- Group = {a, b, c, d, e, f, g, h, i, j}
- N = 4, ranks = {3, 4, 1, 5}
- INCL
  - Newgroup = {c, d, a, e}
- EXCL
  - Newgroup = {b, c, f, g, h, i, j}

Groups

- MPI_GROUP_RANGE_*(group, n, ranges, newgroup)
  - Where * ∈ {INCL, EXCL}
  - N is the number of valid entries in the ranges array
  - Ranges is a tuple (start, end, stride)
- For INCL the order in the new group depend on the order in ranges
- For EXCL the order in the new group depend on the original order

Groups

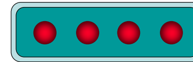
- Group = {a, b, c, d, e, f, g, h, i, j}
- N=3; ranges = ((6, 7, 1), (1, 6, 2), (0, 9, 4))
- Then the range
  - (6, 7, 1) => {g, h} (ranks (6, 7))
  - (1, 6, 2) => {b, d, f} (ranks (1, 3, 5))
  - (0, 9, 4) => {a, e, i} (ranks (0, 4, 8))
- INCL
  - Newgroup = {g, h, b, d, f, a, e, i}
- EXCL
  - Newgroup = {c, j}

Communicators

- A special channel between some processes used to exchange messages.
- Operations creating the communicators are collectives, but accessing the communicator information is a local operation.
- Special communicators: MPI_COMM_WORLD, MPI_COMM_NULL, MPI_COMM_SELF
- MPI_COMM_DUP(comm, newcomm) create an identical copy of the comm in newcomm.
  - Allow exchanging messages between the same set of nodes using identical tags (useful for developing libraries).

Intracommunicators

- What exactly is a intracommunicator ?

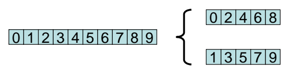


- some processes
- **ONE** group
- one communicator

- MPI_COMM_SIZE, MPI_COMM_RANK
- MPI_COMM_COMPARE(conn1, comm2, result)
  - MPI_IDENT: comm1 and comm2 represent the same communicator
  - MPI_CONGRUENT: same processes, same ranks
  - MPI_SIMILAR: same processes, different ranks
  - MPI_UNEQUAL: otherwise

Intracommunicators

- MPI_COMM_CREATE(comm, group, newcomm)
  - Create a new communicator on all processes from the communicator comm who are defined on the group.
  - All others processes get MPI_COMM_NULL




```
MPI_Group_range_excl( group, 1, (0, 9, 2), odd_group );
MPI_Group_range_excl( group, 1, (1, 9, 2), even_group );
MPI_Comm_create( comm, odd_group, odd_comm );
MPI_Comm_create( comm, even_group, even_comm );
```

Intracommunicators

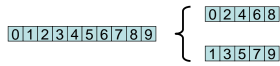
- MPI_COMM_SPLIT(comm, color, key, newcomm)
  - Color : control of subset assignment
  - Key : control of rank assignement

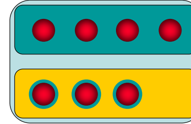
rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
color	0	⊥	3	0	3	0	0	5	3	⊥
key	3	1	2	5	1	1	1	2	1	0

3 different colors => 3 communicators
1. {A, D, F, G} with ranks {3, 5, 1, 1} => {F, G, A, D}
2. {C, E, I} with ranks {2, 1, 3} => {E, C, I}
3. {H} with ranks {1} => {H}

B and J get MPI_COMM_NULL as they provide an undefined color (MPI_UNDEFINED)

Intracommunicators





Rank	0	1	2	3	4	5	6	7	8	9
process	A	B	C	D	E	F	G	H	I	J
Color	0	1	0	1	0	1	0	1	0	1
Key	1	1	1	1	1	1	1	1	1	1

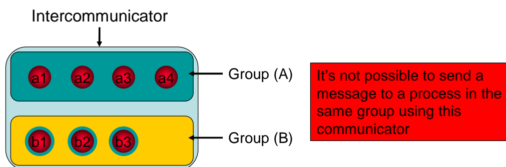
Intercommunicators

- And what's a intercommunicator ?

- some more processes
- **TWO** groups
- one communicator

- MPI_COMM_REMOTE_SIZE(comm, size)
  MPI_COMM_REMOTE_GROUP(comm, group)
- MPI_COMM_TEST_INTER(comm, flag)
- MPI_COMM_SIZE, MPI_COMM_RANK return the local size respectively rank

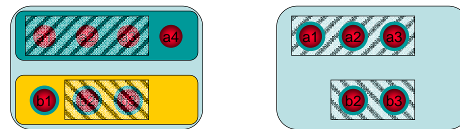
Anatomy of a Intercommunicator



For any processes from group (A)	For any processes from group (B)
• (A) is the local group • (B) is the remote group	• (A) is the remote group • (B) is the local group

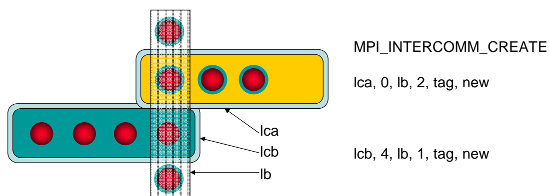
Intercommunicators

- MPI_COMM_CREATE(comm, group, newcomm)
  - All processes on the left group should execute the call with the same subgroup of processes, when all processes from the right side should execute the call with the same subgroup of processes. Each of the subgroup is related to a different side.



Intercommunicators

- MPI_INTERCOMM_CREATE(local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)
  Local_comm : local intracommunicator
  Local_leader : rank of root in the local_comm
  Bridge_comm : "bridge" communicator …
  Remote_leader : rank of remote leader in bridge_comm



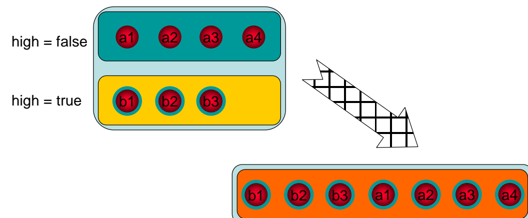
MPI_INTERCOMM_CREATE

lca, 0, lb, 2, tag, new

lcb, 4, lb, 1, tag, new

Intercommunicators

- MPI_INTERCOMM_MERGE(intercomm, high, intracomm)
  - Create an intracomm from the union of the two groups
  - The order of processes in the union respect the original one
  - The high argument is used to decide which group will be first (rank 0)



Example

```
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
int rank = 0;

if( /* left side (ie. a*) */ ) {
  MPI_Comm_group( inter_comm, &local_group);
  MPI_Group_incl( local_group, 1, &rank, &group);
  MPI_Group_free( &local_group );
} else
  MPI_Comm_group( inter_comm, &group );

MPI_Comm_create( inter_comm, group,
                 &new_inter_comm );
MPI_Group_free( &group );
```

a1 a2 a3 a4
b1 b2 b3
a1
b1 b2 b3

Exercice

0 1 2 3 4 5 6 7 8 9 a b

0 3 6 9
1 4 7 a
2 5 8 b

Intercommunicators – P2P

On process 0:
MPI_Send(buf, MPI_INT, 1, n, tag, intercomm)

- Intracommunicator
- Intercommunicator

N = 3

N = 3

Intercommunicators– P2P

On process 0:
MPI_Send(buf, MPI_INT, 1, 0, tag, intercomm)

- Intracommunicator
- Intercommunicator

Not MPI safe if the receive was not posted before.

Communicators - Collectives

- Simple classification by operation class
- **One-To-All** (simplex mode)
  - One process contributes to the result. All processes receive the result.
    - MPI_Bcast
    - MPI_Scatter, MPI_Scatterv
- **All-To-One** (simplex mode)
  - All processes contribute to the result. One process receives the result.
    - MPI_Gather, MPI_Gatherv
    - MPI_Reduce
- **All-To-All** (duplex mode)
  - All processes contribute to the result. All processes receive the result.
    - MPI_Allgather, MPI_Allgatherv
    - MPI_Alltoall, MPI_Alltoallv
    - MPI_Allreduce, MPI_Reduce_scatter
- **Other**
  - Collective operations that do not fit into one of the above categories.
    - MPI_Scan
    - MPI_Barrier

Collectives

	Who generate the result	Who receive the result
One-to-all	One in the local group	All in the local group
All-to-one	All in the local group	One in the local group
All-to-all	All in the local group	All in the local group
Others	?	?

Extended Collectives

From each process point of view

	Who generate the result	Who receive the result
One-to-all	One in the local group	All in the remote group
All-to-one	All in the local group	One in the remote group
All-to-all	All in the local group	All in the remote group
Others	?	?

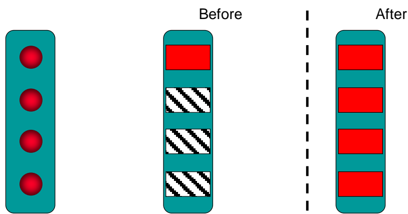
Extended Collectives

- Simplex mode (ie. rooted operations)
  - A root group
    - The root use MPI_ROOT as root process
    - All others use MPI_PROC_NULL
  - A second group
    - All use the real rank of the root in the remote group
- Duplex mode (ie. non rooted operations)
  - Data send by the process in one group is received by the process in the other group and vice-versa.

Broadcast

One-to-all	One in the local group	All in the local group

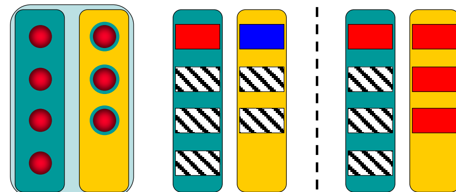
MPI_Bcast(buf, 1, MPI_INT, 0, intracomm)



Extended Broadcast

One-to-all	One in the local group	All in the remote group

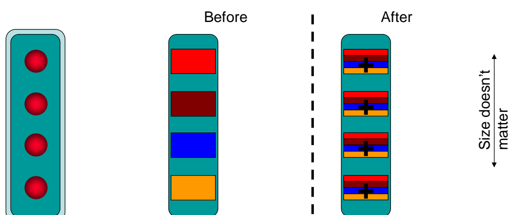
Root group root process: MPI_Bcast(buf, 1, MPI_INT, MPI_ROOT, intercomm)
Root group other processes: MPI_Bcast(buf, 1, MPI_INT, MPI_PROC_NULL, intercomm)
Other group MPI_Bcast(buf, 1, MPI_INT, root_rank, intercomm)



Allreduce

All-to-all	All in the local group	All in the local group

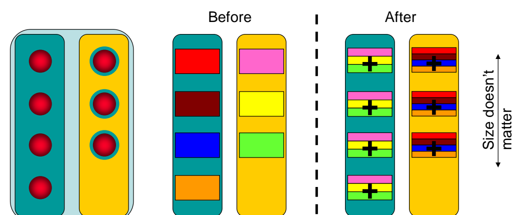
MPI_Allreduce(sbuf, rbuf, 1, MPI_INT, +, intracomm)

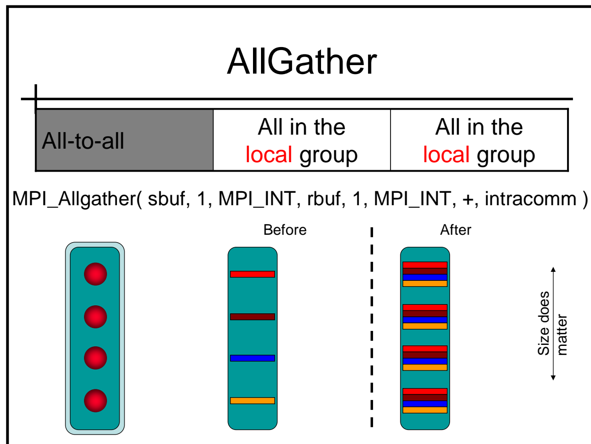


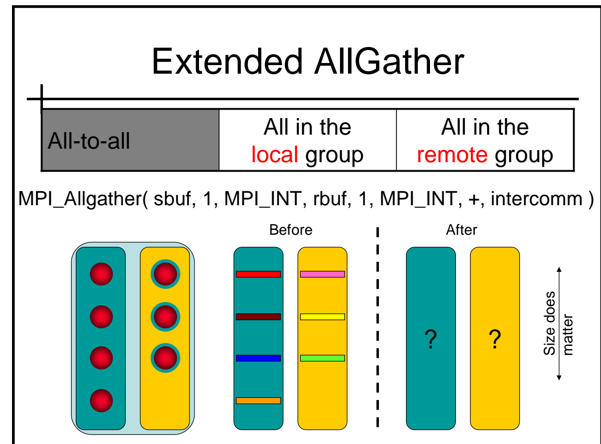
Extended Allreduce

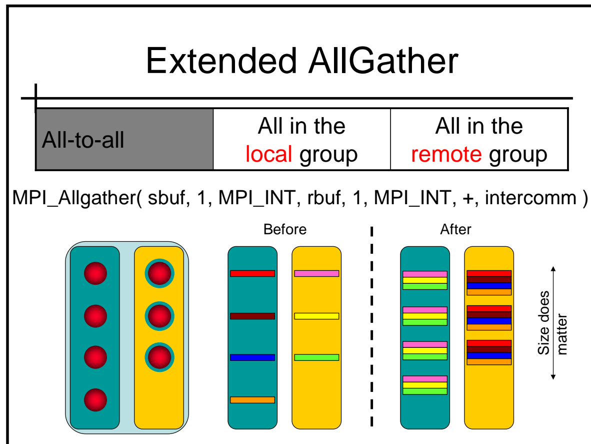
All-to-one	All in the local group	All in the remote group

MPI_Allreduce(sbuf, rbuf, 1, MPI_INT, +, intercomm)









Scan/Exscan and Barrier

- Scan and Exscan are illegal on intercommunicators
- For MPI_Barrier all processes in a group may exit the barrier when all processes on the other group have entered in the barrier.

Dynamic Processes: Spawn

Innovative Computing Laboratory

Dynamic Processes

- Adding processes to a running job
  - As part of the algorithm i.e. branch and bound
  - When additional resources become available
  - Some master-slave codes where the master is started first and asks the environment how many processes it can create
- Joining separately started applications
  - Client-server or peer-to-peer
- Handling faults/failures

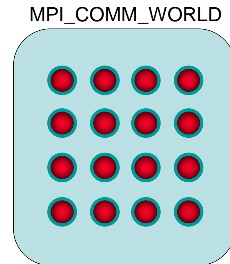
MPI-1 Processes

- All process groups are derived from the membership of the MPI_COMM_WORLD
  - No external processes
- Process membership static (vs. PVM)
  - Simplified consistency reasoning
  - Fast communication (fixed addressing) even across complex topologies
  - Interfaces well to many parallel run-time systems

Static MPI-1 Job

- MPI_COMM_WORLD
- Contains 16 processes

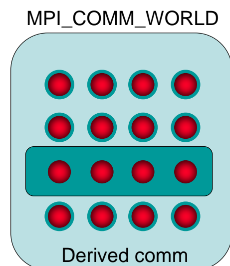
MPI_COMM_WORLD



Static MPI-1 Job

- MPI_COMM_WORLD
- Contains 16 processes
- Can only subset the original MPI_COMM_WORLD
  - No external processes

MPI_COMM_WORLD



Derived comm

Disadvantages of Static Model

- Cannot add processes
- Cannot remove processes
  - If a process fails or otherwise disappears, all communicators it belongs to become invalid

➔ Fault tolerance undefined

MPI-2

- Added support for dynamic processes
  - Creation of new processes on the fly
  - Connecting previously existing processes
- Does not standardize inter-implementation communication
  - Interoperable MPI (IMPI) created for this

Open Questions

How do you add more processes to an already-running MPI-1 job?

- How would you handle a process failure?
- How could you establish MPI communication between two independently initiated, simultaneously running MPI jobs?

MPI-2 Process Management

- MPI-2 provides "spawn" functionality
  - Launches a child MPI job from a parent MPI job
- Some MPI implementations support this
  - Open MPI
  - LAM/MPI
  - NEC MPI
  - Sun MPI
- High complexity: how to start the new MPI applications ?

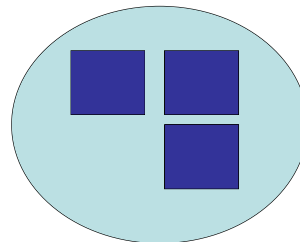
MPI-2 Spawn Functions

- MPI_COMM_SPAWN
  - Starts a set of new processes with the same command line
  - Single Process Multiple Data
- MPI_COMM_SPAWN_MULTIPLE
  - Starts a set of new processes with potentially different command lines
  - Different executables and / or different arguments
  - Multiple Processes Multiple Data

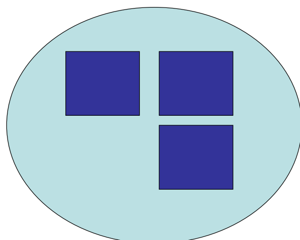
Spawn Semantics

- Group of parents collectively call spawn
  - Launches a new set of children processes
  - Children processes become an MPI job
  - An **inter**communicator is created between parents and children
- Parents and children can then use MPI functions to pass messages
- MPI_UNIVERSE_SIZE

Spawn Example

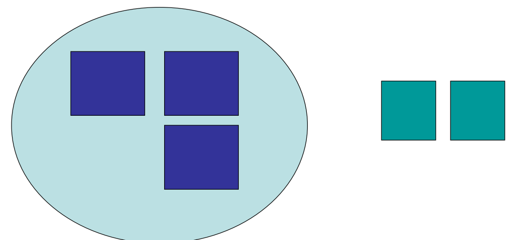


Spawn Example



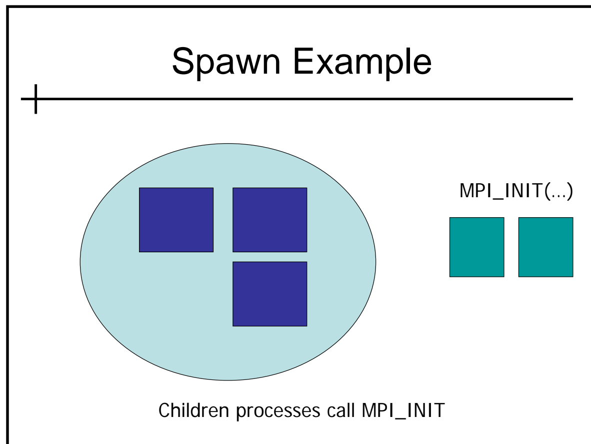
Parents call MPI_COMM_SPAWN

Spawn Example

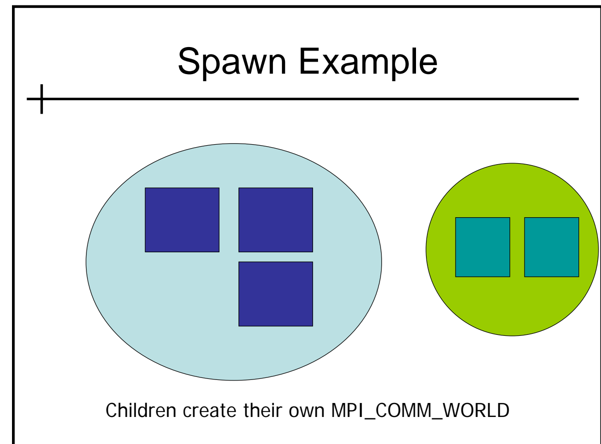


Two processes are launched

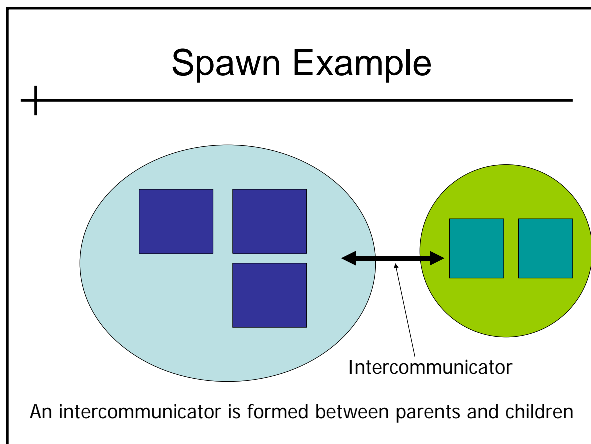
Spawn Example



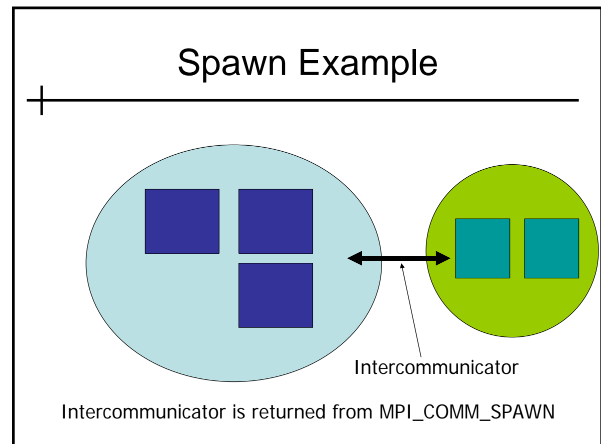
Spawn Example



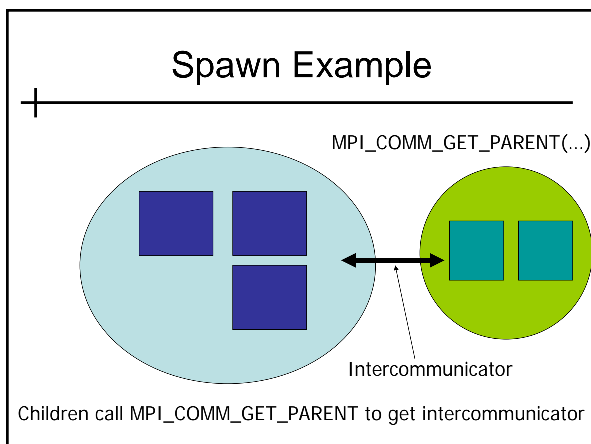
Spawn Example



Spawn Example



Spawn Example



Master / Slave Demonstration

- Simple ‘PVM’ style example
  - User starts singleton master process
  - Master process spawns slaves
  - Master and slaves exchange data, do work
  - Master gathers results
  - Master displays results
  - All processed shut down

Master / Slave Demonstration

Master program

```
MPI_Init(…)
MPI_Spawn(…, slave, …);

for (i=0; i < size; i++)
   MPI_Send(work, …,i, …);
for (i=0; i < size; i++)
   MPI_Recv(presults, …);
calc_and_display_result(…)
MPI_Finalize()
```

Slave program

```
MPI_Init(…)
MPI_Comm_get_parent
   (&intercomm)
MPI_Recv(work,…, intercomm)
result = do_something(work)
MPI_Send(result,…,
   intercomm)
MPI_Finalize()
```

MPI "Connected"

- "Two processes are connected if there is a communication path directly or indirectly between them."
  - E.g., belong to the same communicator
  - Parents and children from SPAWN are connected
- Connectivity is transitive
  - If A is connected to B, and B is connected to C
  - A is connected to C

MPI "Connected"

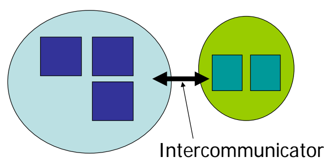
- Why does "connected" matter?
  - MPI_FINALIZE is collective over set of connected processes
  - MPI_ABORT *may* abort all connected processes
- How to disconnect?
  - …stay tuned

Multi-Stage Spawning

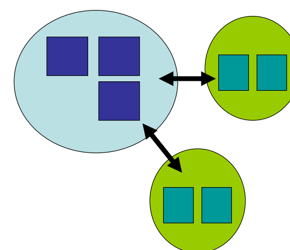
- What about multiple spawns?
  - Can sibling children jobs communicate directly?
  - Or do they have to communicate through a common parent?

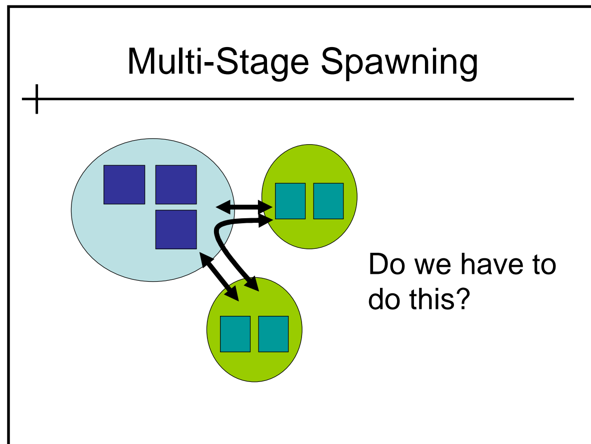
➔ Is all MPI dynamic process communication hierarchical in nature?

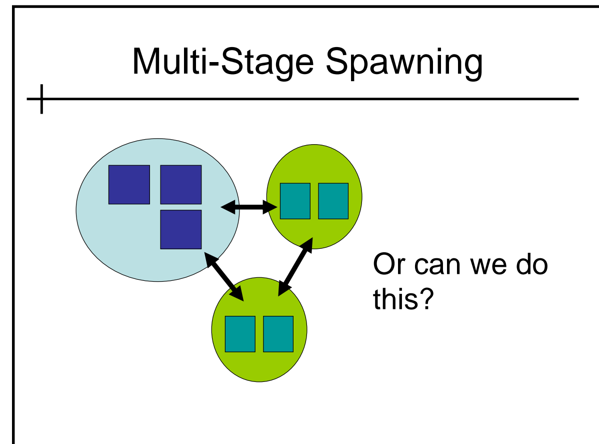
Multi-Stage Spawning



Multi-Stage Spawning







Dynamic Processes: Connect / Accept

Establishing Communications

- MPI-2 has a TCP socket style abstraction
  - Process can accept and connect connections from other processes
  - Client-server interface
- MPI_COMM_CONNECT
- MPI_COMM_ACCEPT

Establishing Communications

- How does the client find the server?
  - With TCP sockets, use IP address and port
  - What to use with MPI?
- Use the MPI name service
  - Server opens an MPI “port”
  - Server assigns a public “name” to that port
  - Client looks up the public name
  - Client gets port from the public name
  - Client connects to the port

Server Side

- Open and close a port
  - MPI_OPEN_PORT(info, port_name)
  - MPI_CLOSE_PORT(port_name)
- Publish the port name
  - MPI_PUBLISH_NAME(service_name, info, port_name)
  - MPI_UNPUBLISH_NAME(service_name, info, port_name)

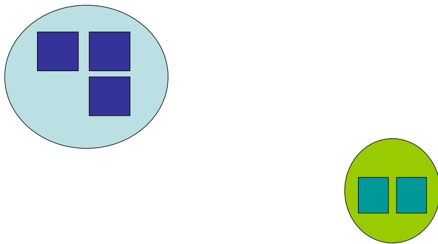
Server Side

- Accept an incoming connection
  - MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
  - comm is a **intra**communicator; local group
  - newcomm is an **inter**communicator; both groups

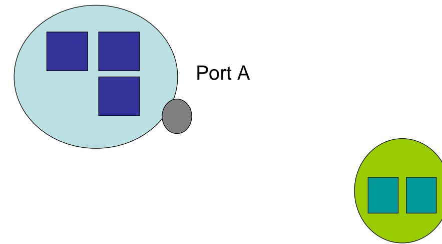
Client Side

- Lookup port name
  - MPI_LOOKUP_NAME(service_name, info, port_name)
- Connect to the port
  - MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
  - comm is a **intra**communicator; local group
  - newcomm is an **inter**communicator; both groups

Connect / Accept Example

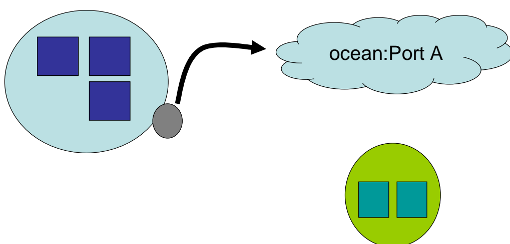


Connect / Accept Example



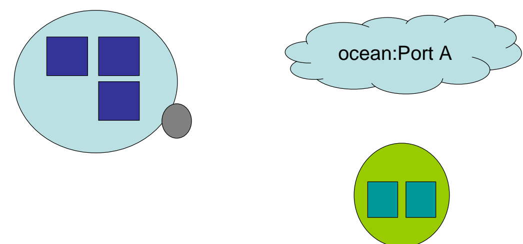
Server calls MPI_OPEN_PORT

Connect / Accept Example

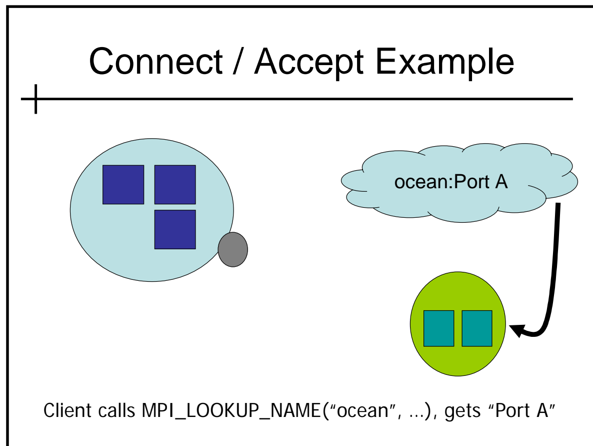


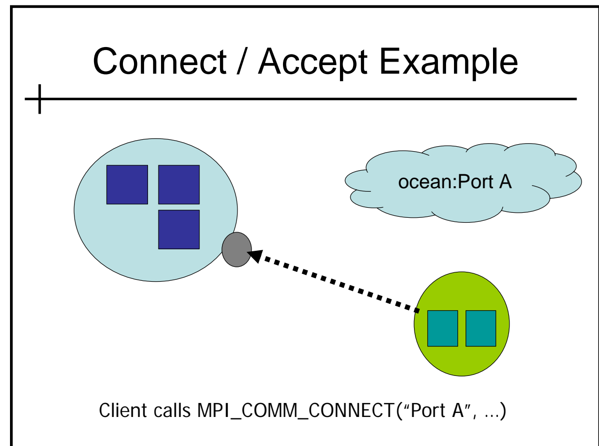
Server calls MPI_PUBLISH_NAME("ocean", info, port_name)

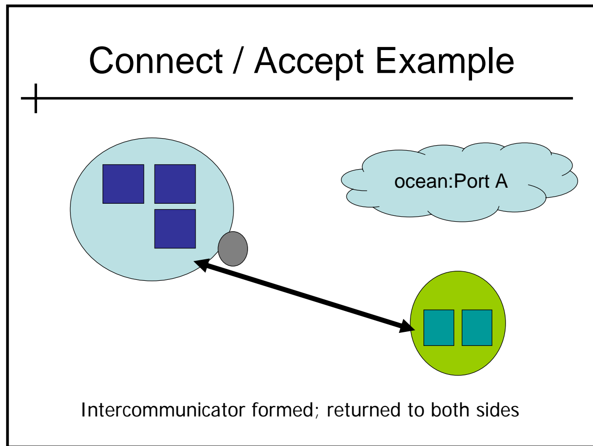
Connect / Accept Example

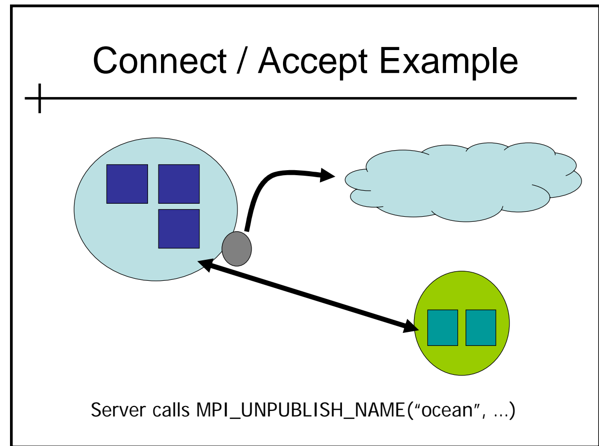


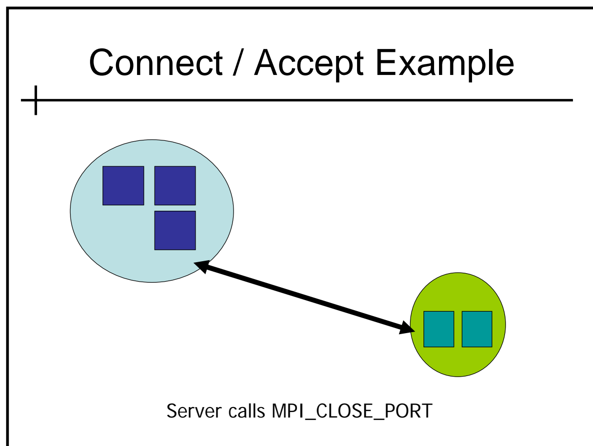
Server blocks in MPI_COMM_ACCEPT("Port A", ...)

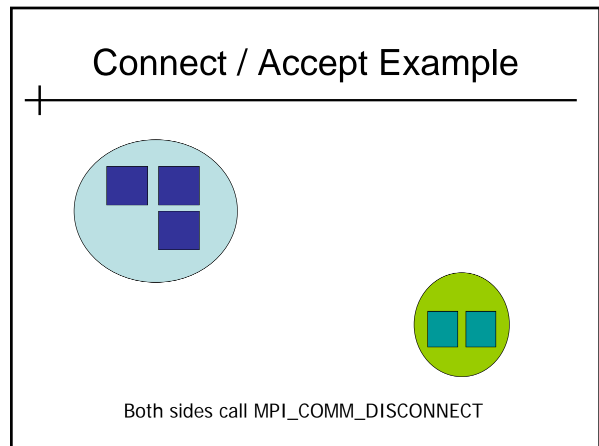

Connect / Accept Example
ocean:Port A
Client calls MPI_LOOKUP_NAME("ocean", ...), gets "Port A"


Connect / Accept Example
ocean:Port A
Client calls MPI_COMM_CONNECT("Port A", ...)


Connect / Accept Example
ocean:Port A
Intercommunicator formed; returned to both sides


Connect / Accept Example
Server calls MPI_UNPUBLISH_NAME("ocean", ...)


Connect / Accept Example
Server calls MPI_CLOSE_PORT


Connect / Accept Example
Both sides call MPI_COMM_DISCONNECT

Summary

- Summary
  - Server opens a port
  - Server publishes public “name”
  - Client looks up public name
  - Client connects to port
  - Server unpublishes name
  - Server closes port
  - Both sides disconnect

➔ Similar to TCP sockets / DNS lookups

MPI_COMM_JOIN

- A third way to connect MPI processes
  - User provides a socket between two MPI processes
  - MPI creates an intercommunicator between the two processes

➔ Will not be covered in detail here

Disconnecting

- Once communication is no longer required
  - MPI_COMM_DISCONNECT
  - Waits for all pending communication to complete
  - Then formally disconnects groups of processes -- no longer “connected”
- Cannot disconnect MPI_COMM_WORLD

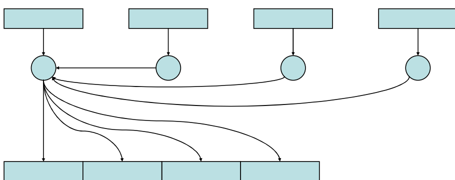
MPI I/O

Based on a presentation given by

Rajeev Thakur

I/O in Parallel Applications (1)

- Sequential I/O:
  - All processes send data to rank 0, and 0 writes it to the file

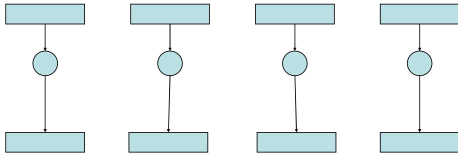


I/O in Parallel Applications (1)

- Pros:
  - parallel machine may support I/O from only one process (e.g., no common file system)
  - Some I/O libraries (e.g. HDF-4, NetCDF) not parallel
  - resulting single file is handy for **ftp**, **mv**
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance (single node bottleneck)

I/O in Parallel Applications (2)

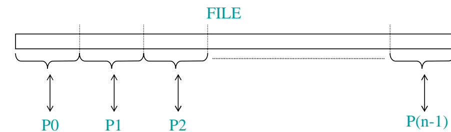
- Each process writes to a separate file



- Pros:
  - parallelism, high performance
- Cons:
  - lots of small files to manage
  - difficult to read back data from different number of processes

What is Parallel I/O ?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



Why Parallel I/O ?

- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

What's the link with MPI ?

- Writing is like sending a message and reading is like receiving.
- Any parallel I/O system will need a mechanism to
  - define collective operations (*MPI communicators*)
  - define noncontiguous data layout in memory and file (*MPI datatypes*)
  - Test completion of nonblocking operations (*MPI request objects*)
- I.e., lots of MPI-like machinery

Basic I/O operations

- int MPI_File_open(MPI_Comm comm, char* filename, int amode, MPI_Info info, MPI_File* fh);
- int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence);
- int MPI_File_read(MPI_File fh, void* buf, int count, MPI_Datatype dtype, MPI_Status* status);
- Write similar to read
- int MPI_File_close(MPI_File* fh);

The simplest MPI IO Application

```
MPI_Comm_rank( comm, &rank );
MPI_File_open( comm, “scratch”, MPI_MODE_RDONLY,
               MPI_INFO_NULL, &fh );
MPI_File_seek( fh, rank * 1024, MPI_SEEK_SET );
MPI_File_read( fh, buf, 1024, MPI_CHAR, &status );
MPI_File_close( fh );
```

- Is this example correct ? Why ?

Using explicit offset

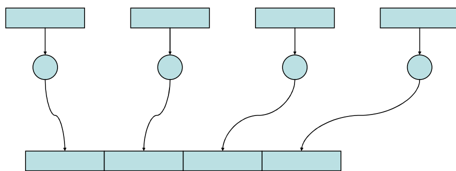
```
call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
        MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_FILE_READ_AT( fh, offset, buf, nints,
                       MPI_INTEGER, status, ierr )
call MPI_GET_COUNT( status, MPI_INTEGER, count, ierr )
print *, 'process ', rank, 'read ', count, 'integers'

call MPI_FILE_CLOSE( fh, ierr )
```

Writing to a file

- Use **MPI_File_write** or **MPI_File_write_at**
- Use **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** as the flags to **MPI_File_open**
- If the file doesn't exist previously, the flag **MPI_MODE_CREATE** must also be passed to **MPI_File_open**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Views

- Processes write to shared file



- **`MPI_File_set_view`** assigns regions of the file to separate processes

Defining a view

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **MPI_File_set_view**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

Example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                  MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```

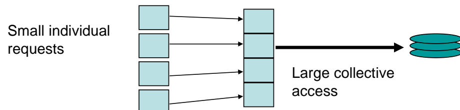
Using files

- **MPI_File_seek** } like Unix seek
- **MPI_File_read_at**
- **MPI_File_write_at** } combine seek and I/O for thread safety
- **MPI_File_read_shared**
- **MPI_File_write_shared** } use shared file pointer

- Collective operations

Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of “big picture” to file system
- Framework for 2-phase I/O, in which communication precedes I/O (can use MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large



Noncontiguous Accesses

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory and file within a single function call by using derived datatypes
- Allows implementation to optimize the access
- Collective IO combined with noncontiguous accesses yields the highest performance.

Non Blocking I/O

```
MPI_Request request;
MPI_Status status;

MPI_File_iwrite_at(fh, offset, buf, count, datatype,
                   &request);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_Wait(&request, &status);
```

Split Collective I/O

- A restricted form of nonblocking collective I/O
- Only one active nonblocking collective operation allowed at a time on a file handle
- Therefore, no request object necessary

```
MPI_File_write_all_begin(fh, buf, count, datatype);

for (i=0; i<1000; i++) {
    /* perform computation */
}

MPI_File_write_all_end(fh, buf, &status);
```

I/O Consistency Semantics

- The consistency semantics specify the results when multiple processes access a common file and one or more processes write to the file
- MPI guarantees stronger consistency semantics if the communicator used to open the file accurately specifies all the processes that are accessing the file, and weaker semantics if not
- The user can take steps to ensure consistency when MPI does not automatically do so

Example 1

- File opened with **MPI_COMM_WORLD**. Each process writes to a *separate* region of the file and reads back only what it wrote.

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	`MPI_File_write_at(off=100,cnt=100)`
`MPI_File_read_at(off=0,cnt=100)`	`MPI_File_read_at(off=100,cnt=100)`

- MPI guarantees that the data will be read correctly

Example 2

- Same as example 1, except that each process wants to read what the *other* process wrote (overlapping accesses)
- In this case, MPI does *not* guarantee that the data will automatically be read correctly

Process 0	Process 1
`/* incorrect program */`	`/* incorrect program */`
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	`MPI_File_write_at(off=100,cnt=100)`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_read_at(off=100,cnt=100)`	`MPI_File_read_at(off=0,cnt=100)`

- **In the above program, the read on each process is not guaranteed to get the data written by the other process!**

Solutions …

- The user must take extra steps to ensure correctness
- There are three choices:
  - set atomicity to true
  - close the file and reopen it
  - ensure that no write sequence on any process is concurrent with any sequence (read or write) on another process

Example 2 – solution 1

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_set_atomicity(fh1,1)`	`MPI_File_set_atomicity(fh2,1)`
`MPI_File_write_at(off=0,cnt=100)`	`MPI_File_write_at(off=100,cnt=100)`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_read_at(off=100,cnt=100)`	`MPI_File_read_at(off=0,cnt=100)`

Example 2 – Solution 2

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	`MPI_File_write_at(off=100,cnt=100)`
`MPI_File_close`	`MPI_File_close`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_read_at(off=100,cnt=100)`	`MPI_File_read_at(off=0,cnt=100)`

Example 2 – Solution 3

- *Ensure that no write sequence on any process is concurrent with any sequence (read or write) on another process*
- a sequence is a set of operations between any pair of open, close, or file_sync functions
- a write sequence is a sequence in which any of the functions is a write operation

Example 2 – Solution 3

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	
`MPI_File_sync`	`MPI_File_sync /*collective*/`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_sync /*collective*/`	`MPI_File_sync`
	`MPI_File_write_at(off=100,cnt=100)`
`MPI_File_sync /*collective*/`	`MPI_File_sync`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_sync`	`MPI_File_sync /*collective*/`
`MPI_File_read_at(off=100,cnt=100)`	`MPI_File_read_at(off=0,cnt=100)`
`MPI_File_close`	`MPI_File_close`

File Interoperability

- Users can optionally create files with a portable binary data representation
- “datarep” parameter to MPI_File_set_view
- native - default, same as in memory, not portable
- internal - impl. defined representation providing an impl. defined level of portability
- external32 - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations

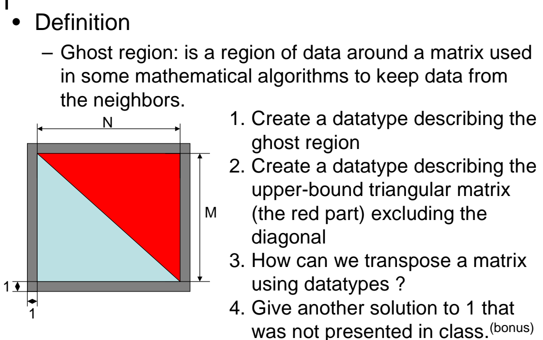
Homework

Homework

- What: 3 Parts
  - Datatypes, Everything else …
- When: Before 2PM on March 2nd 2005
- How: by email at bosilca@cs.utk.edu
  - A MD5 and a tar (gzipped) file
  - Inside a README if something special have to be done.
  - Functional Makefile inside the archive !!!
  - 1 < Ratio comment/code < 2

Homework – Part 1

- Definition
  - Ghost region: is a region of data around a matrix used in some mathematical algorithms to keep data from the neighbors.



1. Create a datatype describing the ghost region
2. Create a datatype describing the upper-bound triangular matrix (the red part) excluding the diagonal
3. How can we transpose a matrix using datatypes ?
4. Give another solution to 1 that was not presented in class.(bonus)

Homework – Part 2

- Do not forget :
  - the exercise on slide 42
  - the exercise on slide 62
  - the question on slide 103-104.
- Provide the simplest solution, and if you have to write a paragraph about, keep it small and concise (not more than 10 lines).

Homework – Part 3

- Create the master / slave framework described on the next slides, respecting the following constraints:
  - Each entity (master, slave, status checker) is a separate MPI application.
  - Do not use MPI_Spawn.
  - The work distributed by the master consist on a unique random number between 1 and 10, and represent the amount of seconds the worker have to sleep before requesting another piece of work.
  - The status checker print the total number of tasks completed as well as the number of tasks pending.
  - Do it in the simplest possible way !!!!

Advanced Master / Slave

- Use all of the MPI topics covered
  - Accept / connect
  - Multi-threaded application
- To create a generalized master / slave framework
  - Adapt the messages exchanged between processes to distribute work and return results
  - Fill in slave functions to do the work

Framework Architecture

A classical (PVM-like) approach
- Workers connect / disconnect at run time

New feature: master is multi-threaded to minimize the response latency

Framework Architecture Master

Connect / Accept thread:
- Accept new connections
- Add the new workers to the work pool
- Handle the status report

Work distribution thread:
- Distribute the available work between the workers
- Manage the workload between clients

Results Management thread:
- Asynchronously handle the result (e.g., save to file)
- Manage memory used for the result

Framework Architecture