Intermediate MPI

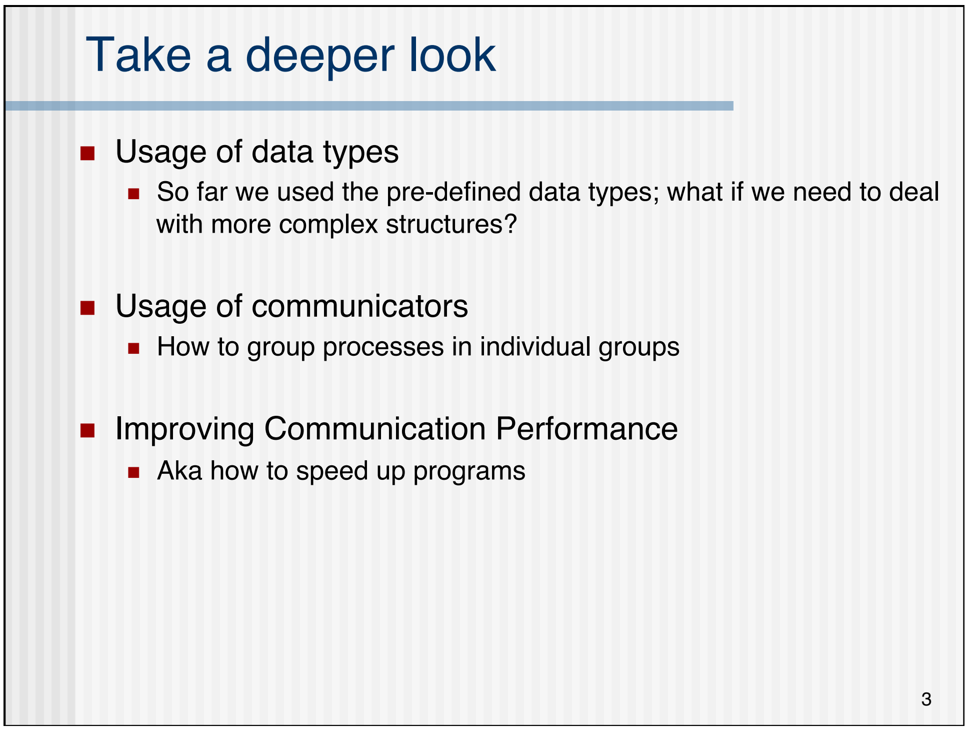
Erwin Laure
Director PDC-HPC

1

What we know already

- Everything to write MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication
  - Collective Communication

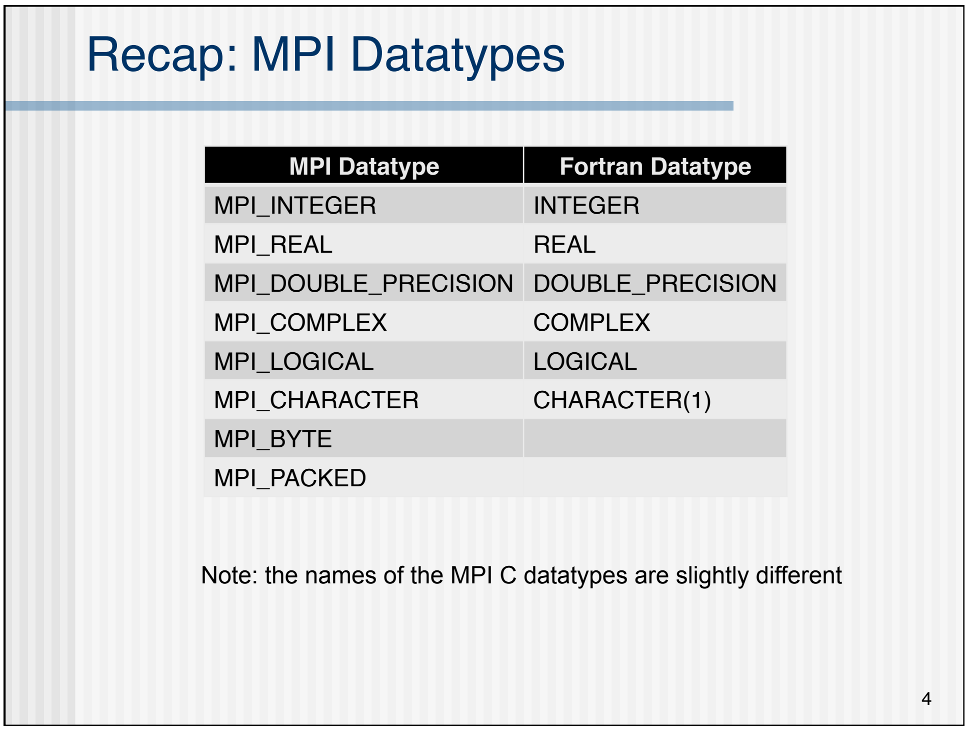
2



Take a deeper look

- Usage of data types
  - So far we used the pre-defined data types; what if we need to deal with more complex structures?
- Usage of communicators
  - How to group processes in individual groups
- Improving Communication Performance
  - Aka how to speed up programs

3



Recap: MPI Datatypes

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Note: the names of the MPI C datatypes are slightly different

4

Derived Datatypes

- Primitive datatypes are contiguous (basically arrays)
- Derived Datatypes allow you to define your own data structures based upon sequences of the MPI primitive data types.
- Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

5

Example

- Send one row of a matrix:
  - Data is contiguous in C; can simply send
  - But it is not contiguous in Fortran

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

- Send one column of a matrix:
  - Same as above but contiguous in Fortran
- How to solve non-contiguous case?
  - Send each element in separate message
    - Overhead and error prone

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

6

Send contiguous data

- Could be achieved simply with

```
MPI_Send(&a[i][0], 4, MPI_FLOAT, j, tag,
         MPI_COMM_WORLD);
```

- If you do this frequently, you might want to use a more descriptive datatype name (eg. coordinate point) and help MPI packing the data

Equivalent to above

```
MPI_Type_contiguous(4, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

MPI_Send(&a[i][0], 1, rowtype, j, tag,
         MPI_COMM_WORLD)
```

7

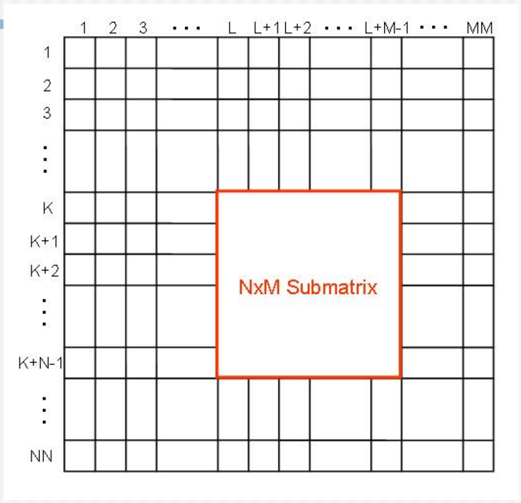
Example Cont'd

```
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
     }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD,
&stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
         rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);
```

- Note different type in send/recv
- Is the program safe?

8

Example: submatrix



```
do j = 1, m
   call MPI_Send(a(k,l+j-1), n, MPI_DOUBLE,
        dest, tag, MPI_COMM_WORLD, ierr)
enddo
```

9

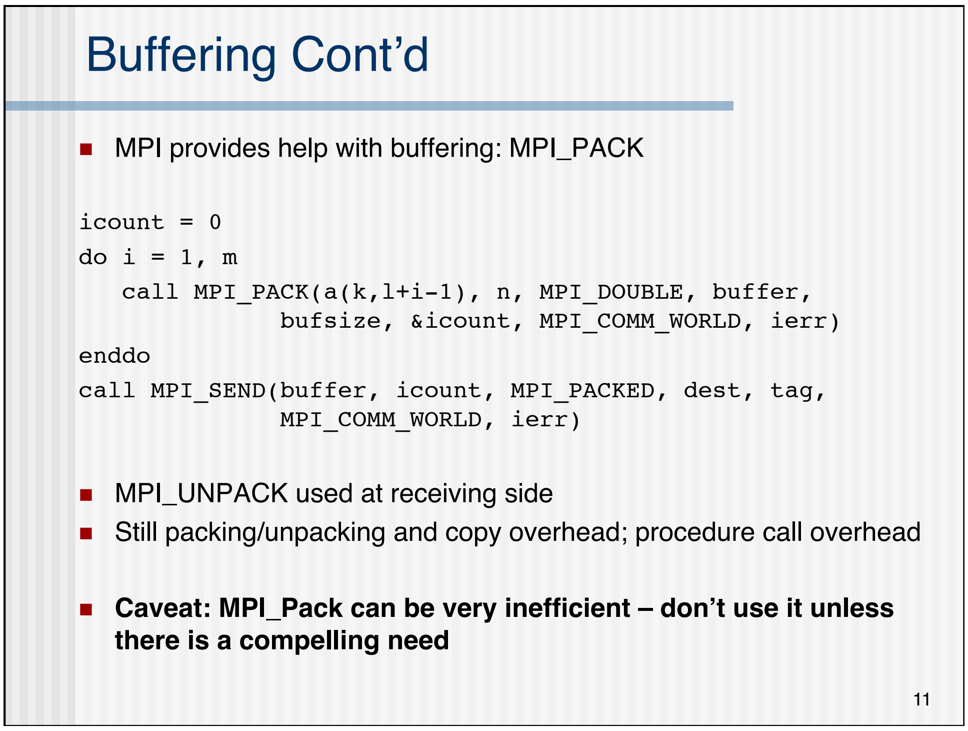
First Approach: Buffering

- Create a user-level buffer for the sub-matrix:

```
icount = 0
do j = l, l+m-1
   do i = k, k+n-1
      icount = icount + 1
      p(icount) = a(i,j)
   enddo
enddo

call MPI_Send(p, n*m, MPI_DOUBLE, dest, tag,
              MPI_COMM_WORLD, ierr)
```

- Limitations:
  - Usage of memory and CPU time to do buffering
  - Still can use only one datatype in the buffer
  - Need to interpret the buffer correctly on the receiving side

10



Buffering Cont'd

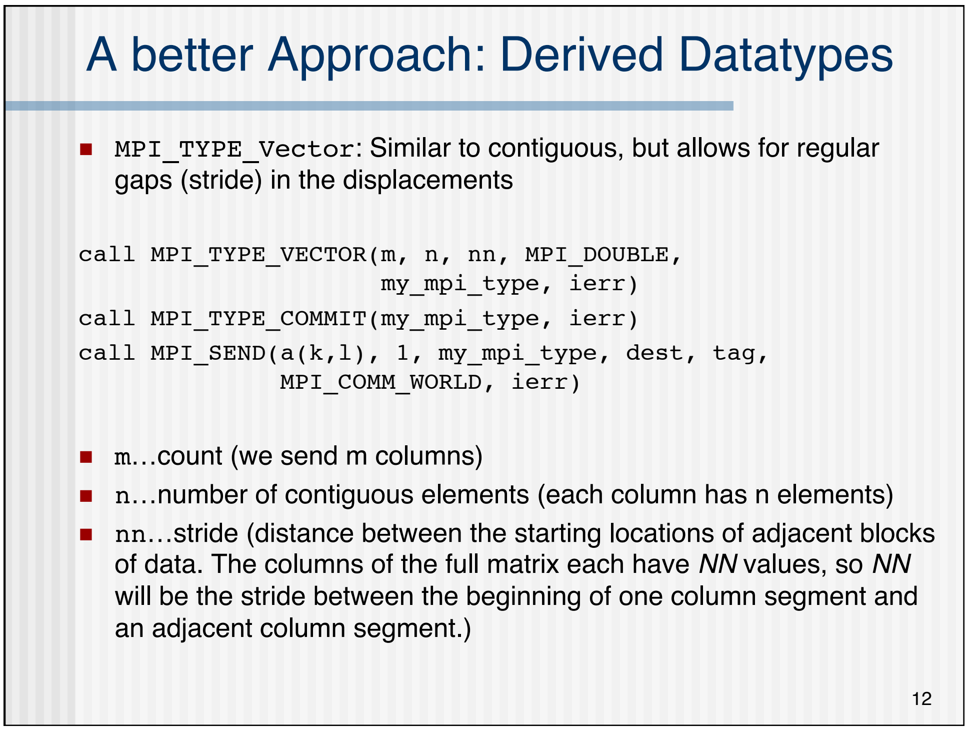
- MPI provides help with buffering: MPI_PACK

```
icount = 0
do i = 1, m
   call MPI_PACK(a(k,l+i-1), n, MPI_DOUBLE, buffer,
              bufsize, &icount, MPI_COMM_WORLD, ierr)
enddo
call MPI_SEND(buffer, icount, MPI_PACKED, dest, tag,
              MPI_COMM_WORLD, ierr)
```

- MPI_UNPACK used at receiving side
- Still packing/unpacking and copy overhead; procedure call overhead

- **Caveat: MPI_Pack can be very inefficient – don't use it unless there is a compelling need**

11



A better Approach: Derived Datatypes

- `MPI_TYPE_Vector`: Similar to contiguous, but allows for regular gaps (stride) in the displacements

```
call MPI_TYPE_VECTOR(m, n, nn, MPI_DOUBLE,
                     my_mpi_type, ierr)
call MPI_TYPE_COMMIT(my_mpi_type, ierr)
call MPI_SEND(a(k,l), 1, my_mpi_type, dest, tag,
              MPI_COMM_WORLD, ierr)
```

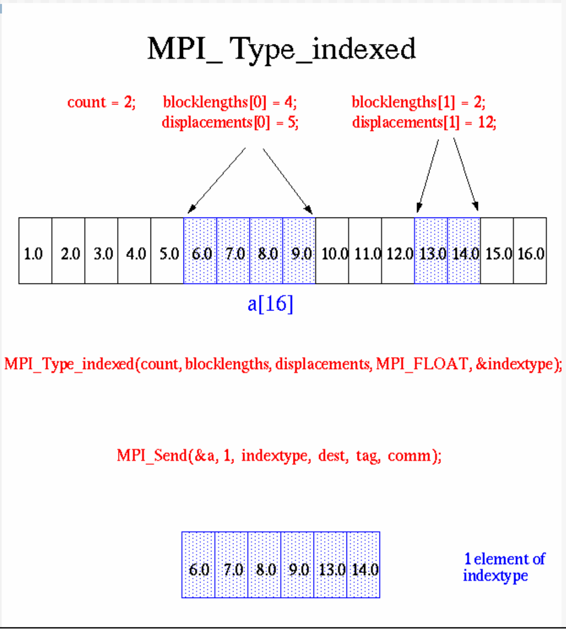
- `m`…count (we send m columns)
- `n`…number of contiguous elements (each column has n elements)
- `nn`…stride (distance between the starting locations of adjacent blocks of data. The columns of the full matrix each have *NN* values, so *NN* will be the stride between the beginning of one column segment and an adjacent column segment.)

12

Different Derived Datatypes

- **Contiguous**: This is the simplest constructor. It produces a new datatype by making count copies of an existing one.
- **Vector**: This is a slight generalization of the contiguous type that allows for regular gaps in the displacements. Elements are separated by multiples of the extent of the input datatype.
- **Hvector**: This is like vector, but elements are separated by a specified number of bytes.
- **Indexed** and **Hindexed**: An array of displacements of the input datatype is provided; the displacements are measured in terms of the extent of the input datatype or in bytes.
- **Struct**: This provides a fully general description.

Example: MPI_TYPE_INDEXED

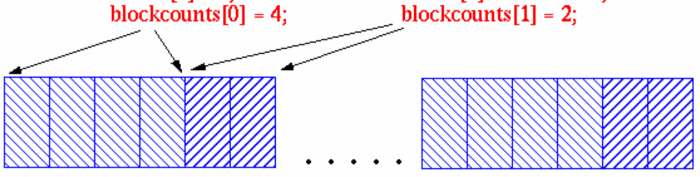


```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);

MPI_Send(&a, 1, indextype, dest, tag, comm);
```

Example: MPI_TYPE_Struct

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;    oldtypes[1] = MPI_INT
           offsets[0] = 0;             offsets[1] = 4 * extent;
           blockcounts[0] = 4;         blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

15

A word of offsets

```
typedef struct {
   float  x, y, z,
          velocity
   int    n, type}
Particle;

Particle particles;
```

- & operator might not return the correct address on all systems
- To find the exact offset one can use the `MPI_Get_address` function

```
MPI_Get_address(
&particles, &p_address);
MPI_Get_address(
&particles.x,&x_address);
…
x_offset = x_address -
p_address;
…
```

16

Other Derived Datatype Commands

- `MPI_Type_extent` returns the size in bytes of the specified data type. Useful for the MPI subroutines that require specification of offsets in bytes.
- `MPI_Type_commit` commits new datatype to the system. Required for all user constructed (derived) datatypes.
- `MPI_TYPE_free` deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.

17

Derived Datatypes Summary

- MPI allows to create user defined datatypes
- Useful if non-contiguous memory locations need to be communicated
- The created derived datatype should be used frequently in a program – otherwise overhead might be too large

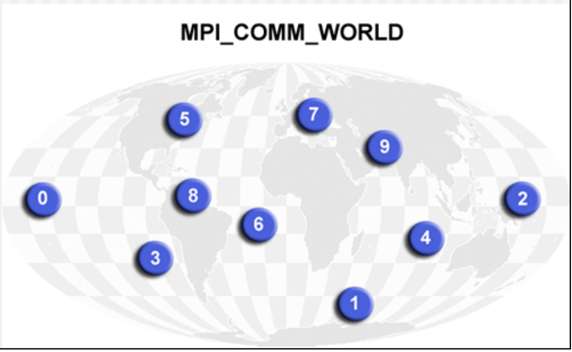
18

Groups and Communicators

19

Recap

- Processes belong to **groups**
- Processes within a group are identified with their **rank**
  - A group of n processes has ranks 0 … n-1
- MPI uses objects called **communicators** and groups to define which collection of processes may communicate with each other
  - `MPI_COMM_WORLD` is the default communicator covering all of the original MPI processes



Communicator Basics

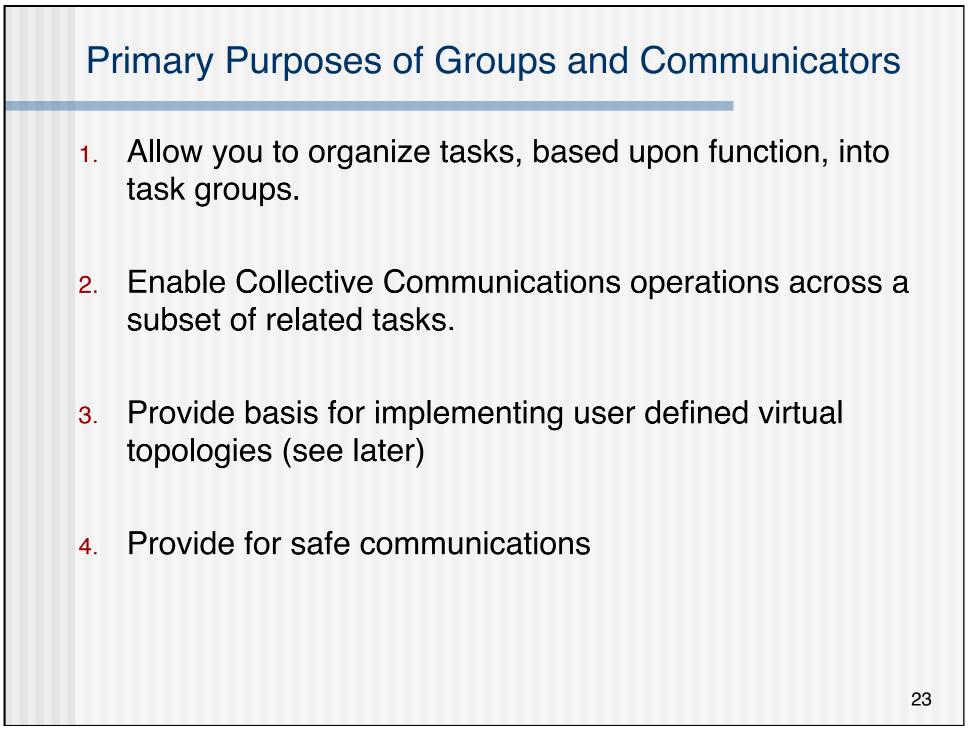
- So far we used MPI_COMM_WORLD
  - Allows any process to communicate with any other process
  - Very useful for many tasks
- Sometimes it is advantageous to restrict the number of processes in a communicator (group)
  - E.g. Matrix-Matrix multiplication:
    - Communication along rows and columns
    - Can have individual communicators for rows and columns
  - E.g. Master/Worker:
    - Restrict certain communications only to workers

21

Groups vs. Communicators

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. A group is always associated with a communicator object.
- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. The communicator that comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

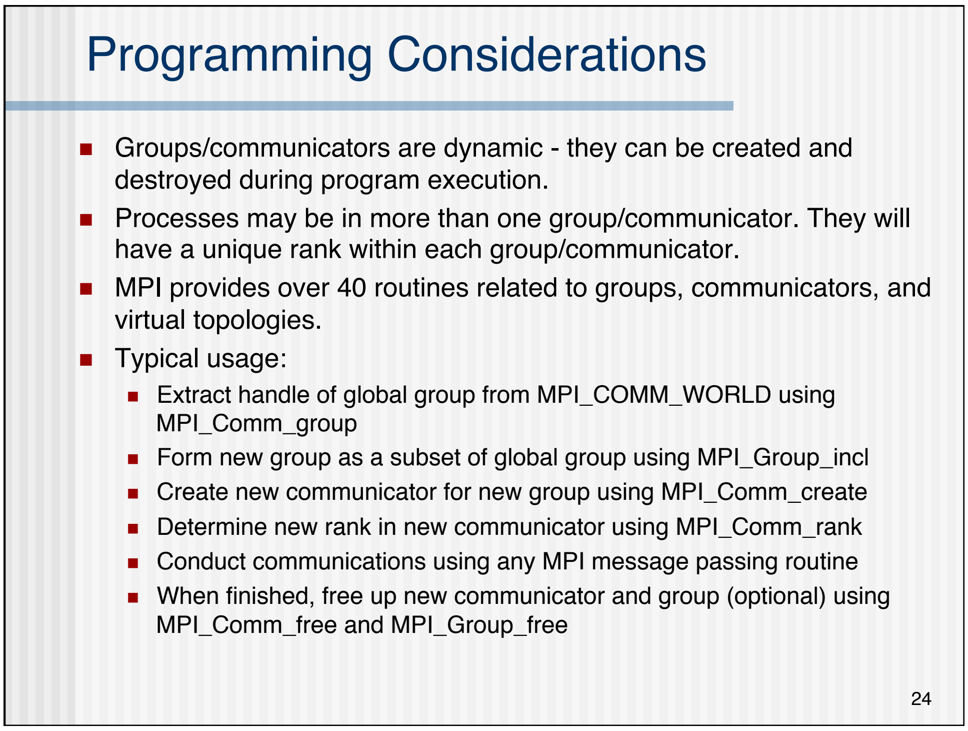
22



Primary Purposes of Groups and Communicators

1. Allow you to organize tasks, based upon function, into task groups.
2. Enable Collective Communications operations across a subset of related tasks.
3. Provide basis for implementing user defined virtual topologies (see later)
4. Provide for safe communications

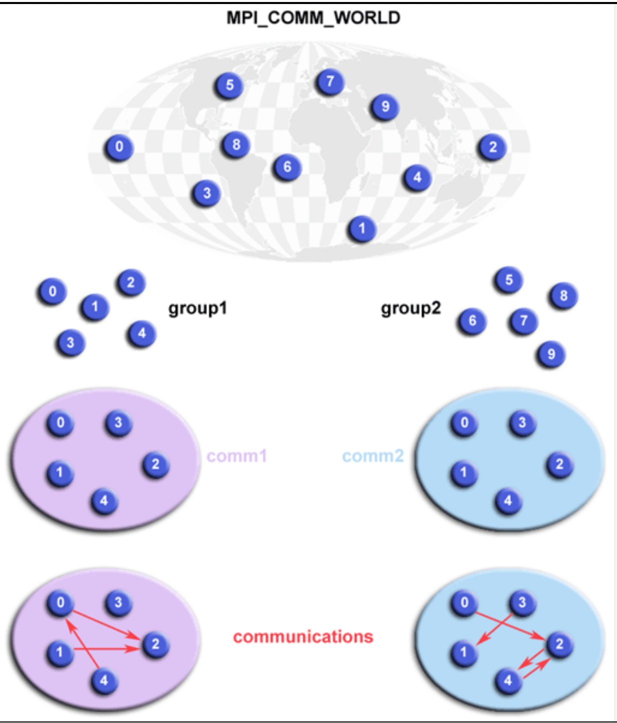
23



Programming Considerations

- Groups/communicators are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
- MPI provides over 40 routines related to groups, communicators, and virtual topologies.
- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

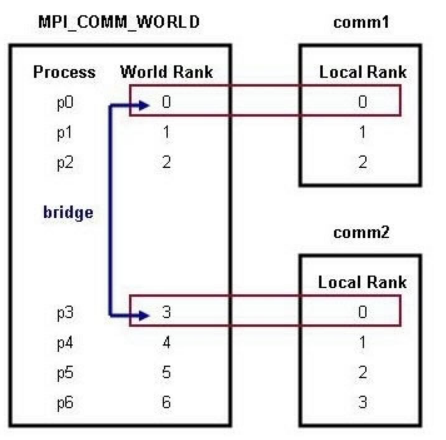
24



25

Intra- and Intercommunicators

- Intracommunicators refer to a process group
  - E.g. comm1 from the example below
  - Allow communication within the group
- Intercommunicators refer to two groups of processes
  - Allow communication between disjoint groups



Creation of Intracommunicators

- Split an existing intracommunicator into two or more sub-communicators
- Duplicate an existing intracommunicator
- Modify a group of processes from an existing intracommunicator, and create a new communicator based on this modified group

Communicator Split

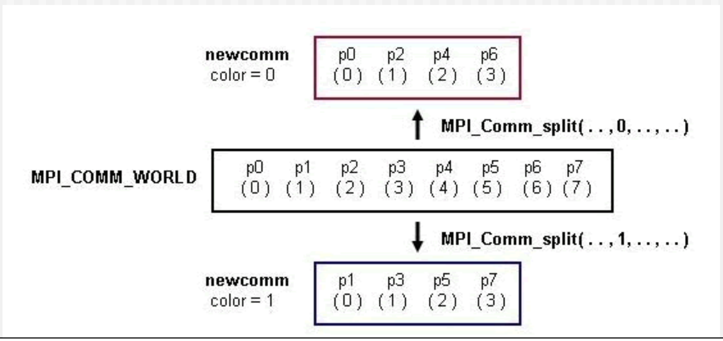
```
MPI_Comm_split(MPI_Comm comm, int color, int key,
               MPI_Comm *newcomm);
MPI_COMM_SPLIT(int comm, int color, int key, int
               newcomm, int IERR)
```

- Color denotes the group a process should be part of
- Key denotes the ranking in the new group

Example

- Split MPI_COMM_WORLD into two groups for even-ranked and odd-ranked process and keep the relative ranking

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
color = rank%2;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &newcomm);
```



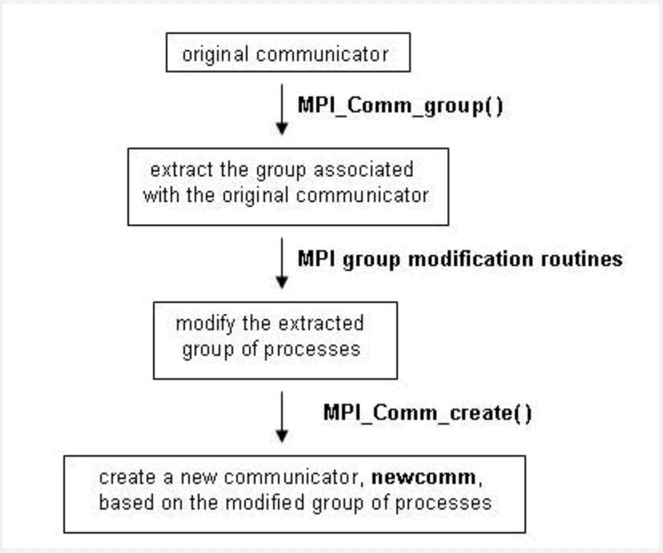
29

Duplication of existing Communicator

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

MPI_COMM_DUP(int comm, int newcomm, int IERR)
```

30

Modifying a Group of Processes


original communicator
MPI_Comm_group()
extract the group associated with the original communicator
MPI group modification routines
modify the extracted group of processes
MPI_Comm_create()
create a new communicator, **newcomm**, based on the modified group of processes

31

Group Modifications

- **`MPI_Group_incl`** creates a new group by reordering a specified number of the processes from an existing group
- **`MPI_Group_excl`** creates a new group from an original group that contains all processes left after deleting those with specified ranks.
- **`MPI_Group_union`** creates a new group that contains all processes in the first group followed by all processes in the second group with no duplication of processes.
- **`MPI_Group_intersection`** creates a new group from two groups that contains all processes that are in both of the groups with rank order the same as that in the first group1.
- **`MPI_Group_difference`** creates a new group from two groups that contains all processes in the first group that are not in the second group with rank order the same as that in the first group.

32

Example

- In a master/worker scheme create communicator for workers
  - Master has rank 0

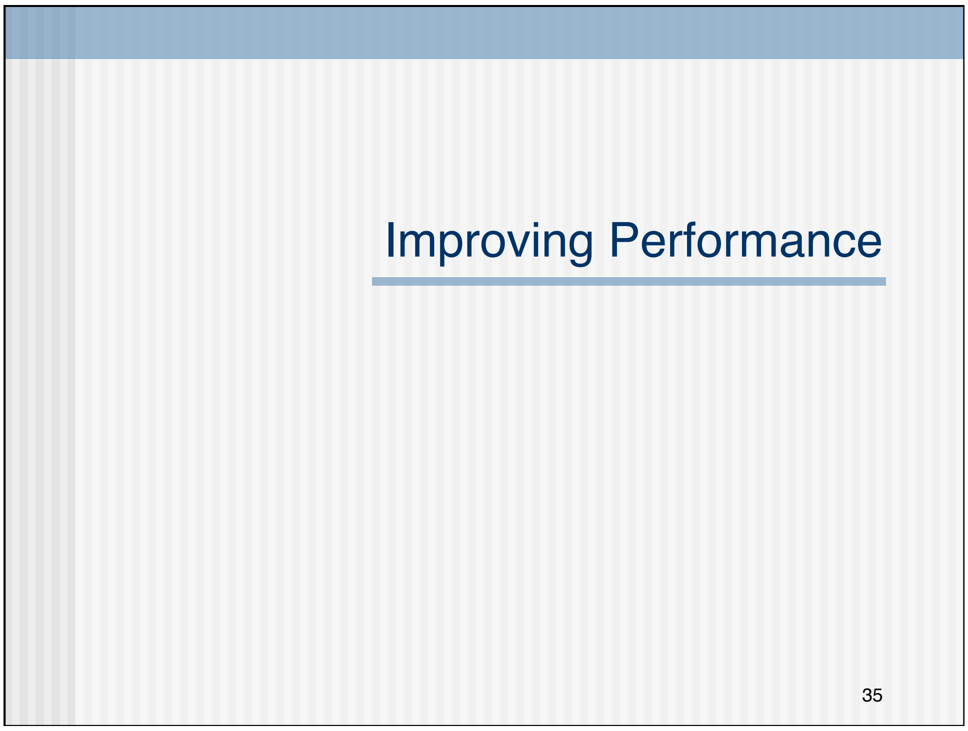
```
comm_world = MPI_COMM_WORLD;

MPI_Comm_group(comm_world, &group_world);

MPI_Group_excl(group_world, 1, &ranks,
               &group_worker);
              /* process 0 not member */

MPI_Comm_create(comm_world, group_worker, &comm_worker);
…
MPI_Comm_free(&comm_worker);
```

33

Communicators Summary

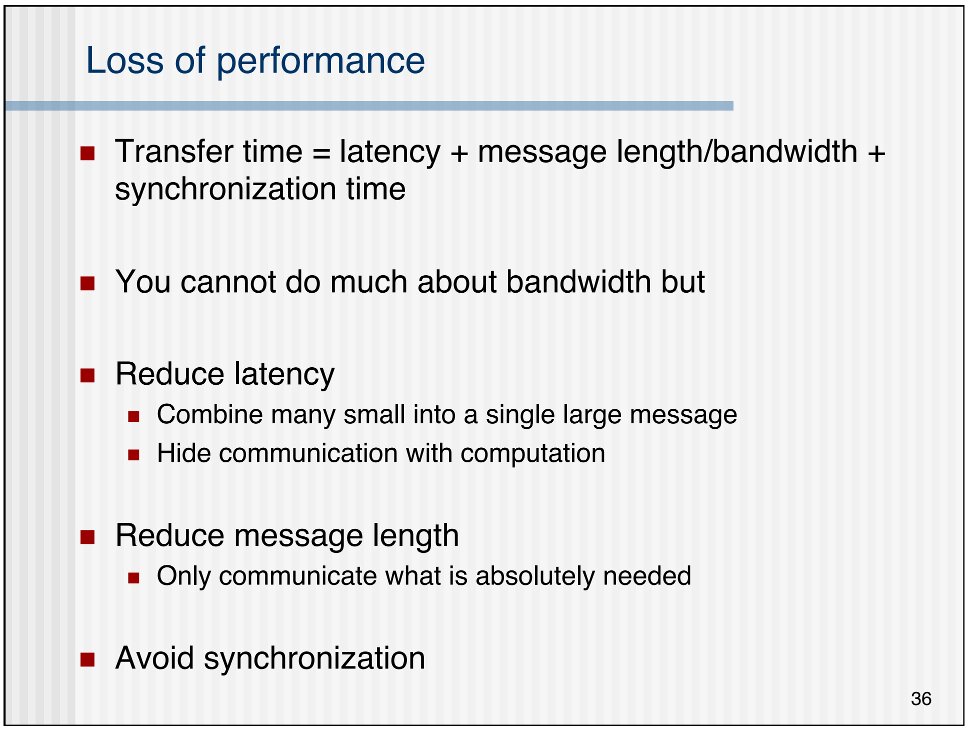
- Communicators provide a powerful tool to restrict communication to subsets of processes
- Useful for certain programming styles
  - E.g. Master/Worker
  - Virtual Topologies (see later)

34



Improving Performance

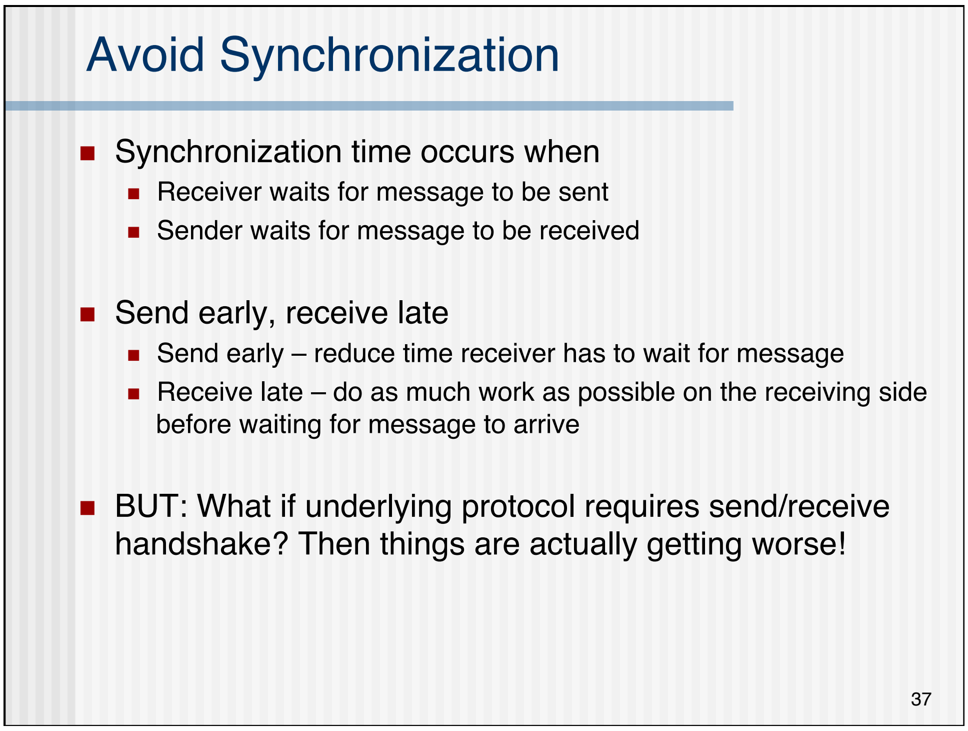
35



Loss of performance

- Transfer time = latency + message length/bandwidth + synchronization time
- You cannot do much about bandwidth but
- Reduce latency
  - Combine many small into a single large message
  - Hide communication with computation
- Reduce message length
  - Only communicate what is absolutely needed
- Avoid synchronization

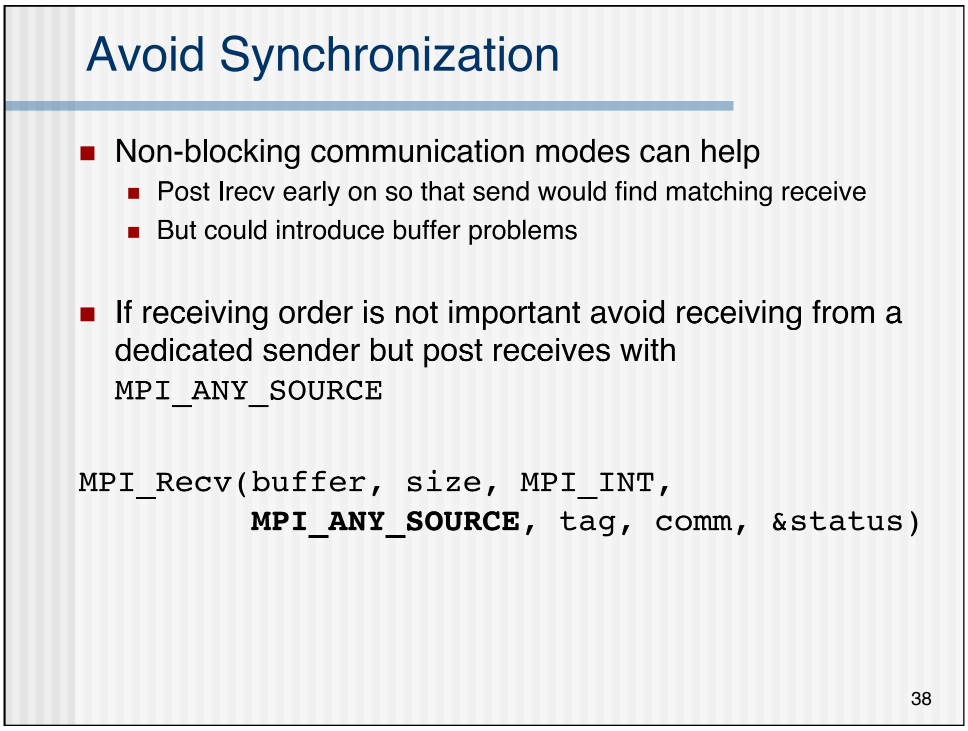
36



Avoid Synchronization

- Synchronization time occurs when
  - Receiver waits for message to be sent
  - Sender waits for message to be received
- Send early, receive late
  - Send early – reduce time receiver has to wait for message
  - Receive late – do as much work as possible on the receiving side before waiting for message to arrive
- BUT: What if underlying protocol requires send/receive handshake? Then things are actually getting worse!

37

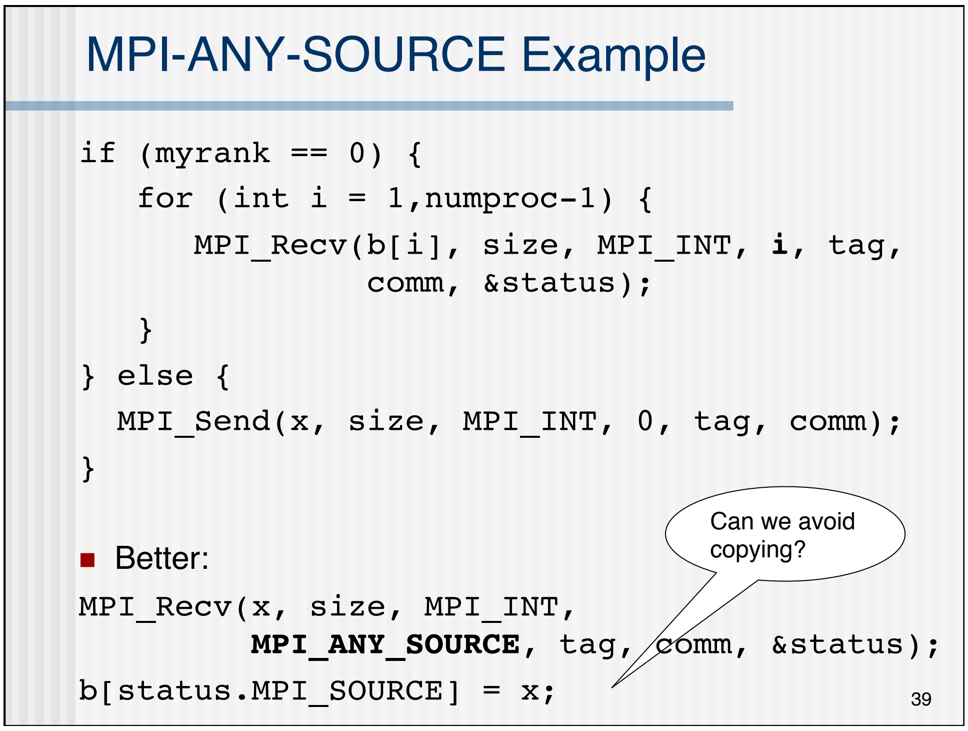


Avoid Synchronization

- Non-blocking communication modes can help
  - Post Irecv early on so that send would find matching receive
  - But could introduce buffer problems
- If receiving order is not important avoid receiving from a dedicated sender but post receives with `MPI_ANY_SOURCE`

```
MPI_Recv(buffer, size, MPI_INT,
         MPI_ANY_SOURCE, tag, comm, &status)
```

38



MPI-ANY-SOURCE Example

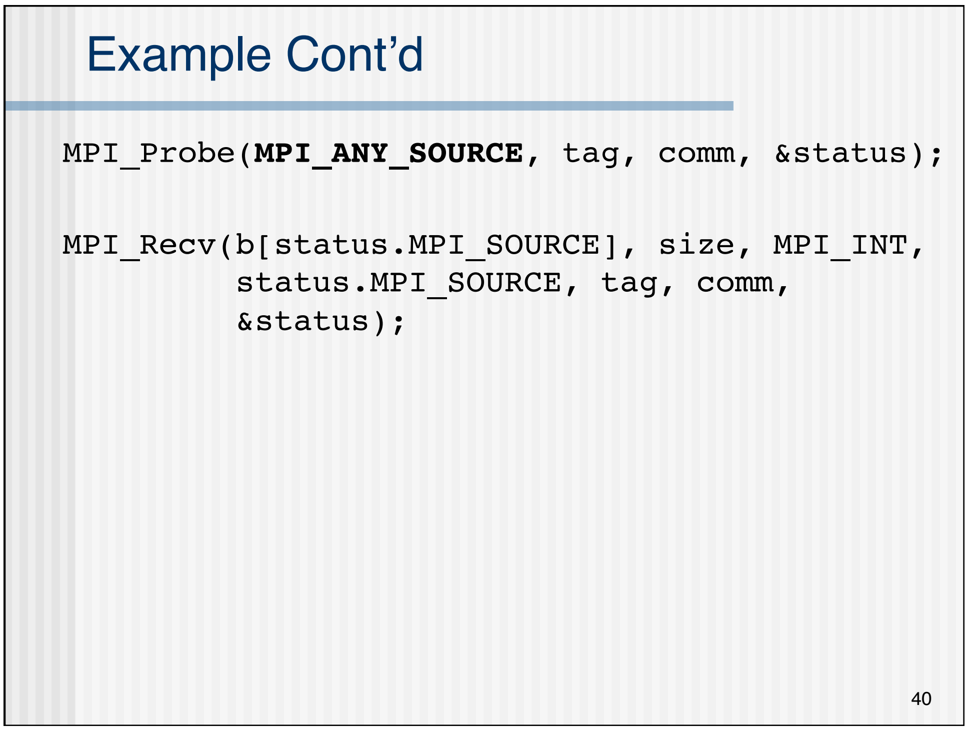
```
if (myrank == 0) {
   for (int i = 1,numproc-1) {
      MPI_Recv(b[i], size, MPI_INT, i, tag,
               comm, &status);
   }
} else {
  MPI_Send(x, size, MPI_INT, 0, tag, comm);
}
```

- Better:

```
MPI_Recv(x, size, MPI_INT,
         MPI_ANY_SOURCE, tag, comm, &status);
b[status.MPI_SOURCE] = x;
```

Can we avoid copying?

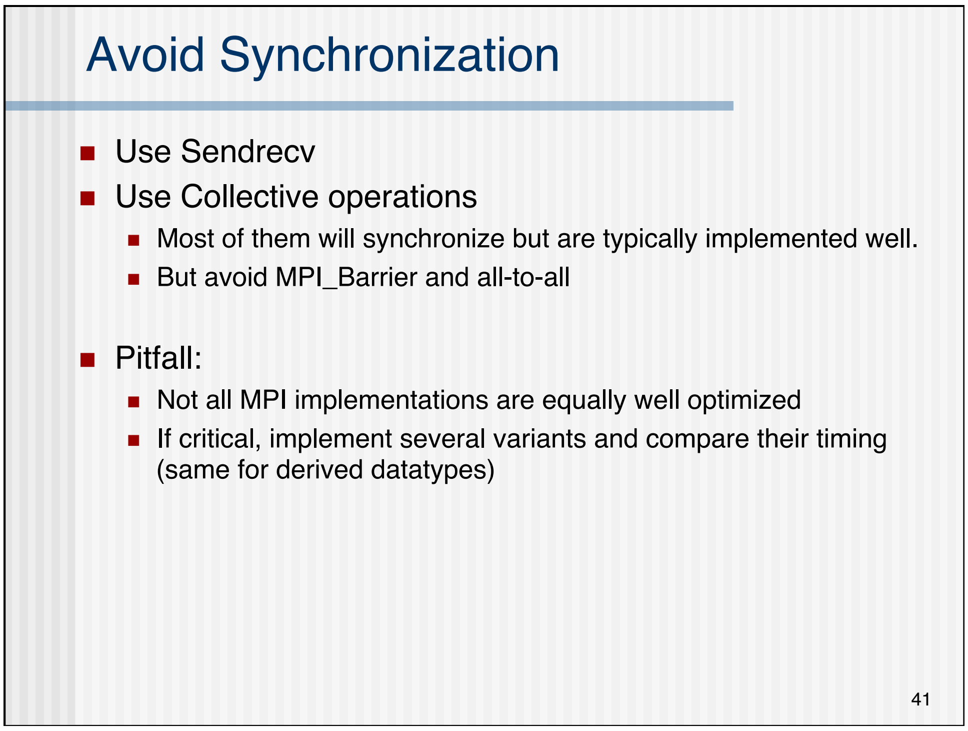
39



Example Cont’d

```
MPI_Probe(MPI_ANY_SOURCE, tag, comm, &status);

MPI_Recv(b[status.MPI_SOURCE], size, MPI_INT,
         status.MPI_SOURCE, tag, comm,
         &status);
```

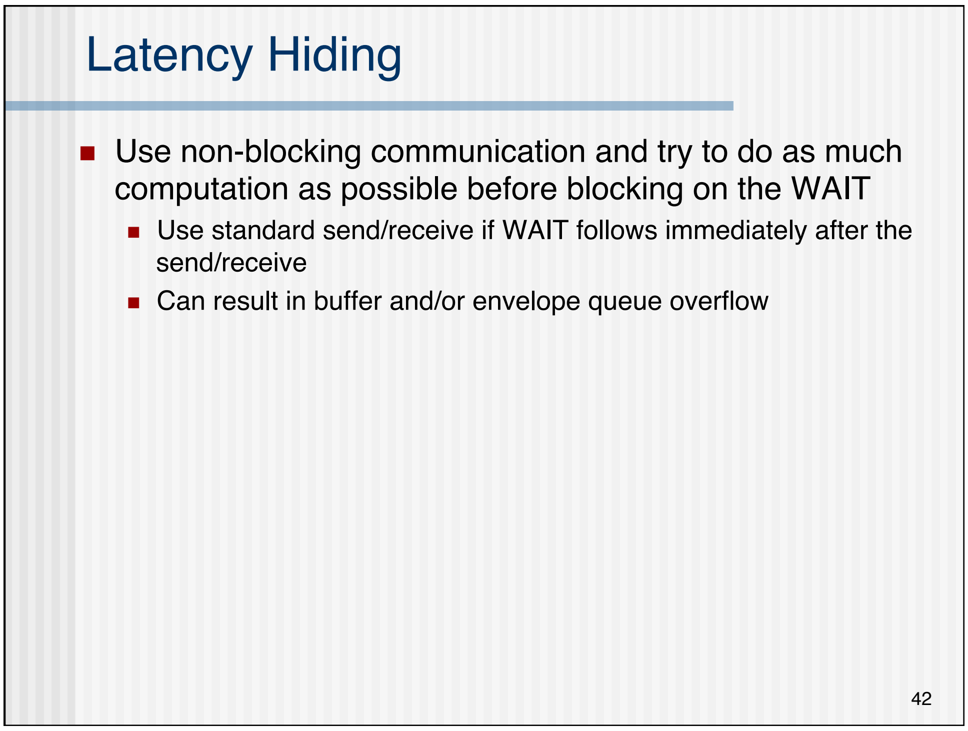
40



Avoid Synchronization

- Use Sendrecv
- Use Collective operations
  - Most of them will synchronize but are typically implemented well.
  - But avoid MPI_Barrier and all-to-all
- Pitfall:
  - Not all MPI implementations are equally well optimized
  - If critical, implement several variants and compare their timing (same for derived datatypes)

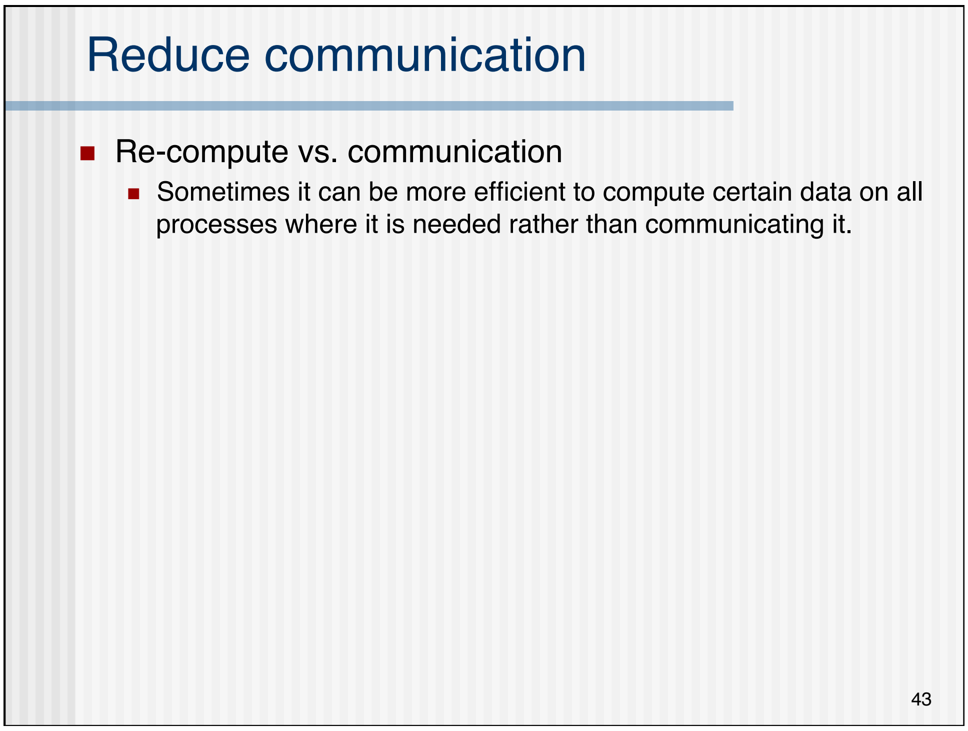
41



Latency Hiding

- Use non-blocking communication and try to do as much computation as possible before blocking on the WAIT
  - Use standard send/receive if WAIT follows immediately after the send/receive
  - Can result in buffer and/or envelope queue overflow

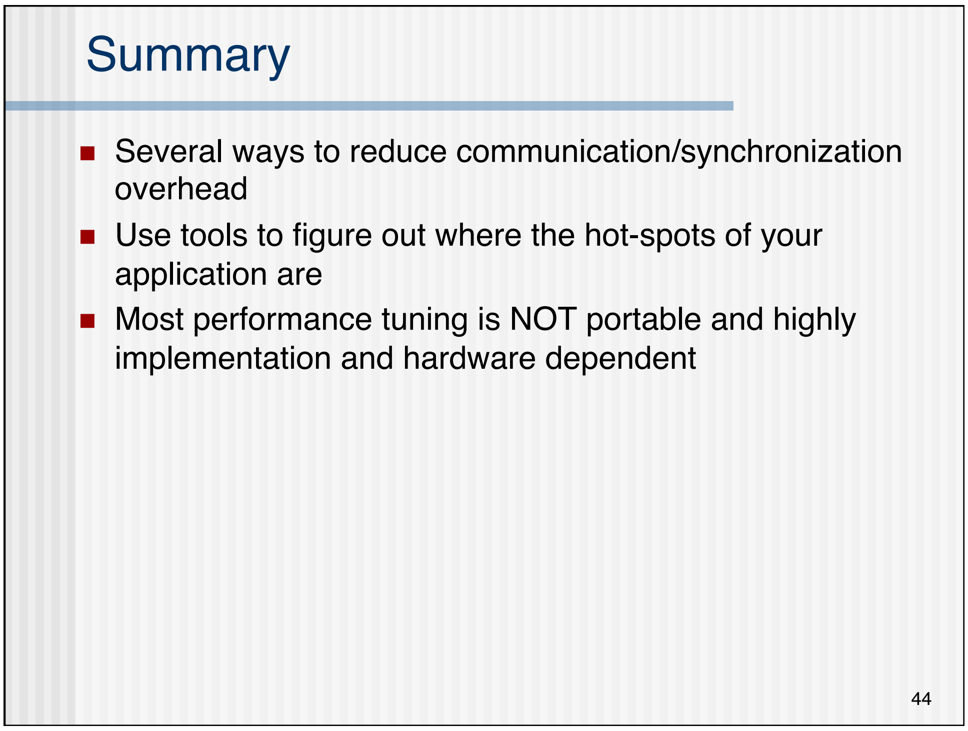
42



Reduce communication

- Re-compute vs. communication
  - Sometimes it can be more efficient to compute certain data on all processes where it is needed rather than communicating it.

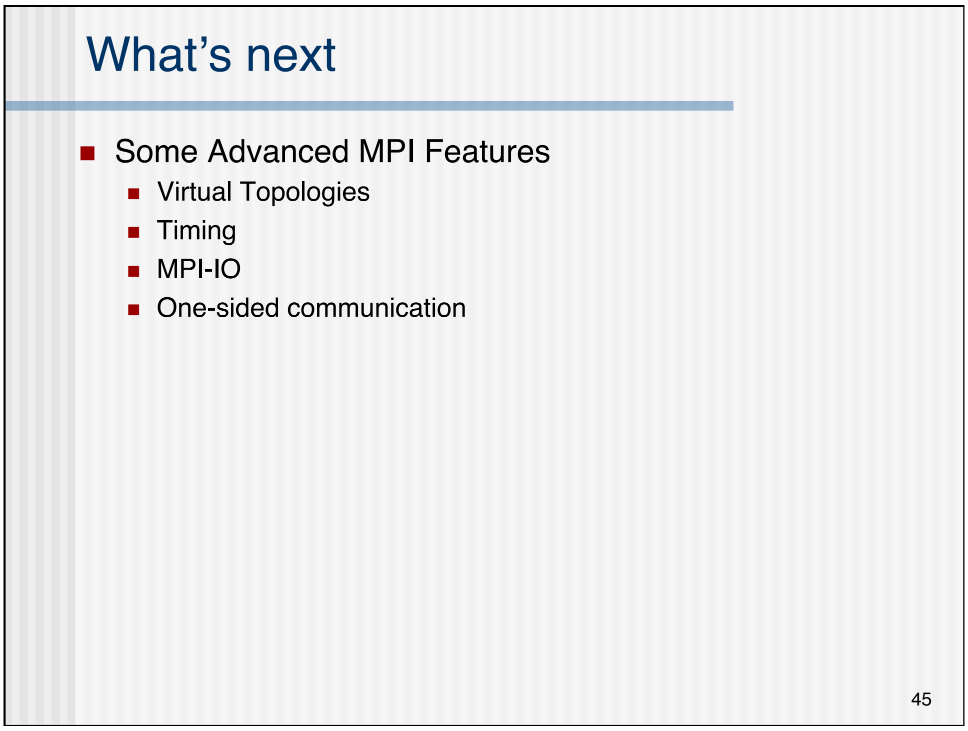
43



Summary

- Several ways to reduce communication/synchronization overhead
- Use tools to figure out where the hot-spots of your application are
- Most performance tuning is NOT portable and highly implementation and hardware dependent

44



What’s next

- Some Advanced MPI Features
  - Virtual Topologies
  - Timing
  - MPI-IO
  - One-sided communication

45