Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

MPI Collective Communication

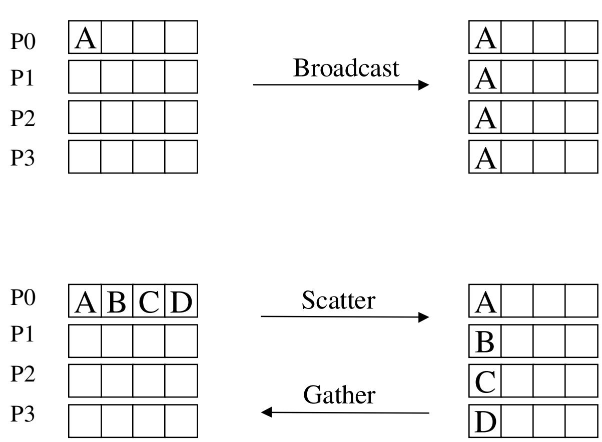
- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

Synchronization

- **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.

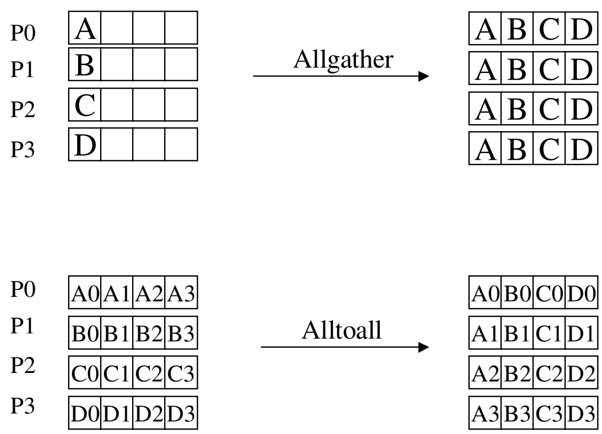
3

Collective Data Movement



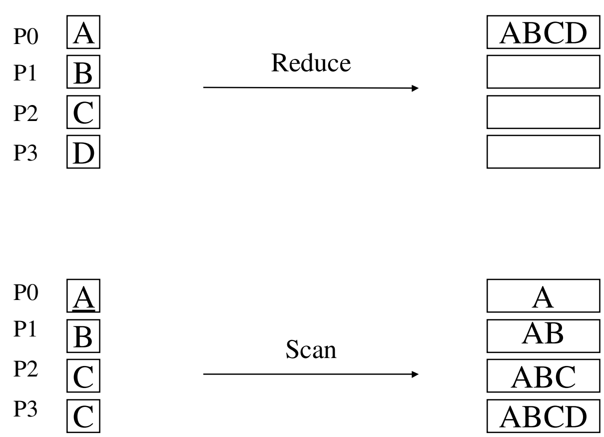
4

More Collective Data Movement



5

Collective Computation



6

3

MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, ReduceScatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`ReduceScatter`**, and **`Scan`** take both built-in and user-defined combiner functions.

7

MPI Built-in Collective Computation Operations

`MPI_Max`	Maximum
`MPI_Min`	Minimum
`MPI_Prod`	Product
`MPI_Sum`	Sum
`MPI_Land`	Logical and
`MPI_Lor`	Logical or
`MPI_Lxor`	Logical exclusive or
`MPI_Band`	Binary and
`MPI_Bor`	Binary or
`MPI_Bxor`	Binary exclusive or
`MPI_Maxloc`	Maximum and location
`MPI_Minloc`	Minimum and location

8

Defining your own Collective Operations

- Create your own collective computations with:

```
MPI_Op_create( user_fcn, commutes, &op );
MPI_Op_free( &op );

user_fcn( invec, inoutvec, len, datatype );
```

- The user function should perform:

```
inoutvec[i]  =  invec[i]  op  inoutvec[i];
```

  for i from 0 to len-1.
- The user function can be non-commutative.

9

When *not* to use Collective Operations

- Sequences of collective communication can be pipelined for better efficiency
- Example: Processor 0 reads data from a file and broadcasts it to all other processes.
  - Do i=1,m
    if (rank .eq. 0) read *, a
    call mpi_bcast(a, n, MPI_INTEGER, 0, comm, ierr)
    EndDo
  - Takes m n log p time.
- It can be done in (m+p) n time!

10

5

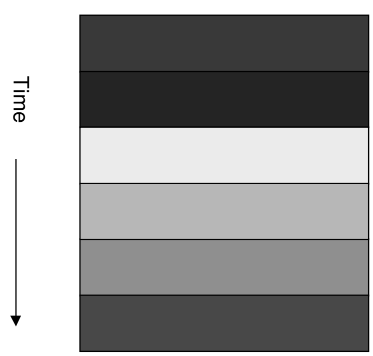
Pipeline the Messages

- Processor 0 reads data from a file and sends it to the next process. Other forward the data.
  - »
```
Do i=1,m
  if (rank .eq. 0) then
      read *, a
      call mpi_send(a, n, type, 1, 0, comm,ierr)
  else
      call mpi_recv(a,n,type,rank-1, 0,
                         comm,status,ierr)
      call mpi_send(a,n,type,next, 0, comm,ierr)
  endif
EndDo
```

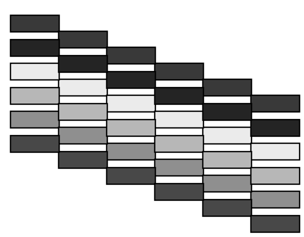
11

Concurrency between Steps

- Broadcast:



- Pipeline



Each broadcast takes less time then pipeline version, but total time is longer

Another example of deferring synchronization

12

Notes on Pipelining Example

- Use MPI_File_read_all
  - » Even more optimizations possible
    - – Multiple disk reads
    - – Pipeline the individual reads
    - – Block transfers
- Sometimes called “digital orrery”
  - » Circular particles in n-body problem
  - » Even better performance if pipeline never stops
- “Elegance” of collective routines can lead to fine-grain synchronization
  - » performance penalty

Implementation Variations

- Implementations vary in goals and quality
  - » Short messages (minimize separate communication steps)
  - » Long messages (pipelining, network topology)
- MPI’s general datatype rules make some algorithms more difficult to implement
  - » Datatypes can be different on different processes; only the type signature must match

Using Datatypes in Collective Operations

- Datatypes allow noncontiguous data to be moved (or computed with)
- As for all MPI communications, only the *type signature* (basic, language defined types) must match
  - » Layout in memory can differ on each process

15

Example of Datatypes in Collective Operations

- Distribute a matrix from one processor to four
  - » Processor 0 gets A(0:n/2,0:n/2), Processor 1 gets A(n/2+1:n,0:n/2), Processor 2 gets A(0:n/2,n/2+1:n), Processor 3 get A(n/2+1:n,n/2+1:n)
- Scatter (One to all, different data to each)
  - » Data at source is not contiguous (n/2 numbers, separated by n/2 numbers)
  - » Use vector type to represent submatrix

16

Matrix Datatype

- MPI_Type_vector(n/2 per block, n/2 blocks, dist from beginning of one block to next = n, MPI_DOUBLE_PRECISION, &subarray_type)
- Can use this to send
  - » Do j=0,1
    ```
    Do j=0,1
      Do i=0,1
       call MPI_Send( a(1+i*n/2:i*n/2+n/2,
                        1+j*n/2:j*n/2+n/2),1,
                       subarray_type, … )
    ```
  - » Note sending **ONE** type contain multiple basic elements

Scatter with Datatypes

- Scatter is like
  - » Do i=0,p-1
    ```
    Do i=0,p-1
      call mpi_send(a(1+i*extent(datatype)),….)
    ```
    – "1+" is from 1-origin indexing in Fortran
  - » Extent is the distance from the beginning of the first to the end of the last data element
  - » For subarray_type, it is
    ((n/2-1)n+n/2) * extent(double)

Layout of Matrix in Memory

N = 8 example

Process 0

0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27

Process 2

32	40	48	56
33	41	49	57
34	42	50	58
35	43	51	59

Process 1

4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

Process 3

36	44	52	60
37	45	53	61
38	46	54	62
39	47	55	63

Using MPI_UB

- Set Extent of each datatype to n/2
  - » Size of contiguous block all are built from
- Use Scatterv (independent multiples of extent)
- Location (beginning location) of blocks
  - » Processor 0: 0 * 4
  - » Processor 1: 1 * 4
  - » Processor 2: 8 * 4
  - » Processor 3: 9 * 4
- MPI-2: Use MPI_Type_create_resized instead

1(

Changing Extent

- MPI_Type_struct
  » types(1) = subarray_type
    types(2) = MPI_UB
    displac(1) = 0
    displac(2) = (n/2) * 8   ! Bytes!
    blklens(1) = 1
    blklens(2) = 1
    call MPI_Type_struct(2, blklens, displac, types,
        newtype, ierr)
- `newtype` contains all of the data of `subarray_type`.
  » Only change is "extent," which is used *only* when computing where in a buffer to get or put data relative to other data

Scattering A Matrix

- sdisplace(1) = 0
  sdisplace(2) = 1
  sdisplace(3) = n
  sdisplace(4) = n + 1
  scounts(1,2,3,4)=1
  call MPI_Scatterv(a, scounts, sdispls, newtype,&
      alocal, n*n/4, MPI_DOUBLE_PRECISION,&
      0, comm, ierr)
  » Note that process 0 sends 1 item of newtype but all processes receive $n^2/4$ double precision elements
- Exercise: Work this out and convince yourself that it is correct