PCSE Lecture 9

MPI Collectives

Cyrus Proctor
Victor Eijkhout

March 31, 2015

Introduction to Collectives

- Collective communication involves all the processes in a communicator
  - From one process to all
  - From all processes to one
  - From all processes to all processes
- Built upon point-to-point communication routines
- Could build your own collective communications
  - Would be tedious
  - Likely not as efficient
- Includes blocking routines
- As of MPI-3, introduced nonblocking routines of all its collective communication calls

Characteristics of Collective Communication Routines

MPI collective communication routines differ in many ways from MPI point-to-point communication routines:

- For blocking calls, must block until they have completed locally
- May, or may not, use synchronized communications (implementation dependent)
- Specify a root process to originate or receive all data, in some cases
- Must exactly match amounts of data specified by senders and receivers
- Do not use message tags or statuses
- Have many variations within the basic categories

Collective Communication Routines can be Divided into Three Subsets:

- Synchronization Operations – processes wait until all members of the group have reached the synchronization point
- Data Movement Operations – broadcast, scatter/gather, all to all
- Global Computation Operations – (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data

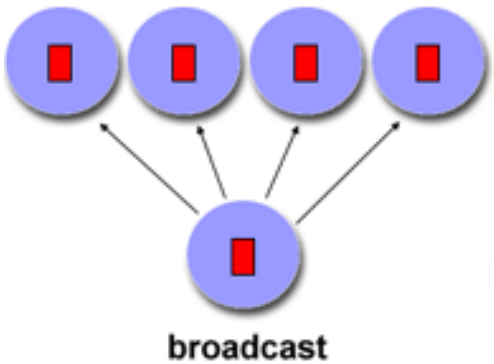
- A basic collective routine will fall into one or more subsets
- May have an "All" (all) variant
- May have a "variable" (v) variant
- With MPI-3, has "Initiate" or non-blocking (i) variant
- Variants can stack,e.g. (i) + (all) + (basic routine) + (v)
- As a result, this gives a large and robust library of routines to choose from

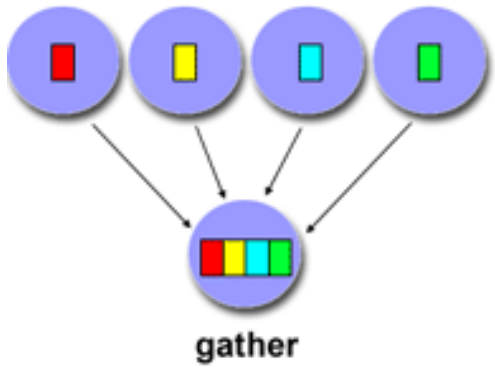
Synchronization – MPI_Barrier

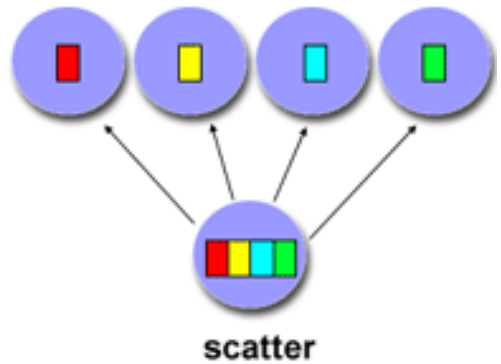
C	`int MPI_Barrier(MPI_Comm comm)`
Fortran	`MPI_BARRIER(comm, ierr)`

- Any process calling it will be blocked until all the processes within the group have called it
- Simple to use, looks very useful, most of the time, it is not
- Unsafe to use with non-blocking calls – use MPI_Wait (and friends)
- May be used implicitly in collective calls

Data Movement – Basic Routines



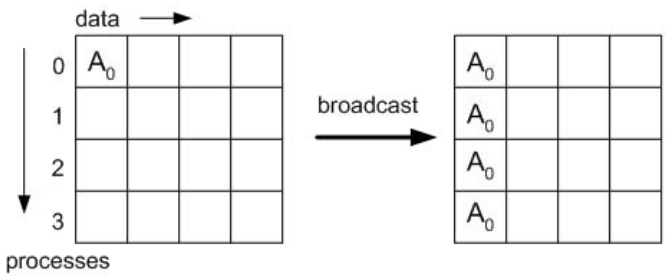




- MPI provides three categories of collective data-movement routines in which one process either sends to or receives from all processes: broadcast, gather, and scatter

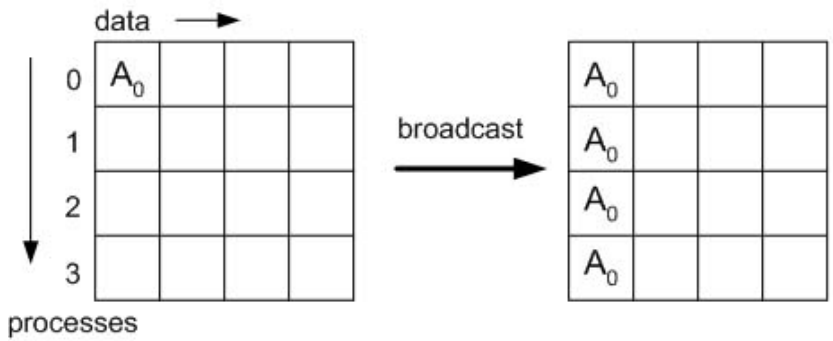
Basic Data Movement – Broadcast (MPI_Bcast)

C	`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
Fortran	`MPI_BCAST(buffer, count, datatype, root, comm, ierr)`



- All processes call MPI_Bcast
- One node (root) sends a message to all
  - All others receive the message

MPI_Bcast Example



- Create a message on the root process
- Send the message to all other processes
- Similar to the figure above

MPI_BCAST Example

Fortran Broadcast

```
program broadcast
  use mpi
  implicit none
  character(13) message
  integer rank, root, ierr
  data root/0/

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  if (rank .eq. root) then ! root can be any valid process
      message = "Hello, World!" ! len 13
  endif
  call MPI_BCAST(message, 13, MPI_CHARACTER, root, &
                 MPI_COMM_WORLD, ierr)
  write(*,*)"(Rank ", rank, "): Message received: ", message
  call MPI_FINALIZE(ierr)
end program broadcast
```

MPI_Bcast Example

C Broadcast

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char *argv[]){
  char message[14];
  int rank, size;
  int root = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == root){ // root can be any valid process
    strcpy(message, "Hello, world!"); // len 13
  } // don't forget the null terminated character
  MPI_Bcast(&message[0], 14, MPI_CHAR, root, MPI_COMM_WORLD);
  printf( "(Rank %i): Message received: %s\n", rank, message);

  MPI_Finalize();
}
```

MPI_Bcast Example Output

Run with ibrun -np 4 ./a.out

Fortran Output

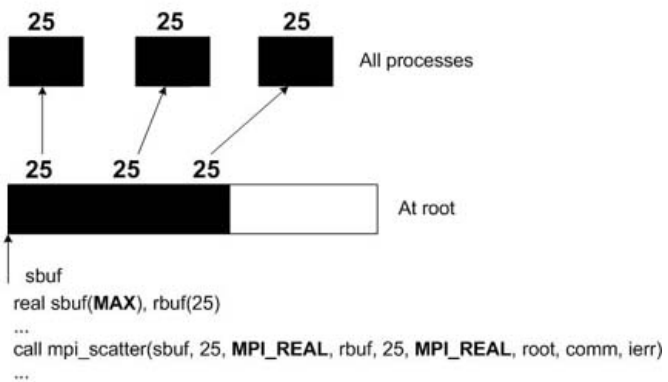
```
(Rank           0 ): Message received: Hello, World!
(Rank           1 ): Message received: Hello, World!
(Rank           2 ): Message received: Hello, World!
(Rank           3 ): Message received: Hello, World!
```

C Output

```
(Rank 0): Message received: Hello, world!
(Rank 1): Message received: Hello, world!
(Rank 2): Message received: Hello, world!
(Rank 3): Message received: Hello, world!
```

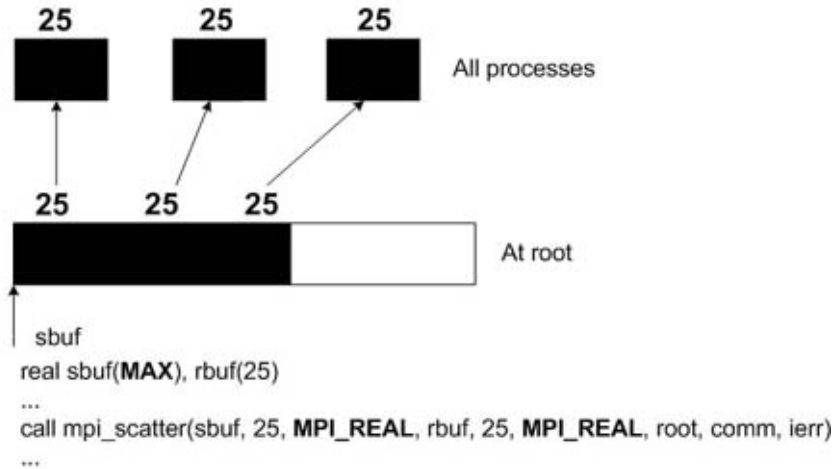
Basic Data Movement – Scatter (MPI_Scatter)

C	`int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Fortran	`MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)`



- Distribute the data into n segments
- The ith segment is sent to the ith process in the group which has n processes
- Splits data where MPI_Bcast does not

MPI_Scatter Example



- Create an array on the root process and populate it
- Split the array evenly amongst all processes
- Send the ith chunk of the array to the ith process
- Similar to the figure above

MPI_SCATTER Example

Fortran Scatter

```
program scatter
  use mpi
  implicit none
  integer, parameter :: SOME_MAX=8
  integer sendbuf(SOME_MAX), recvbuf(2)
  integer i, rank, numprocs, root, ierr
  data root/0/

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  if(rank .eq. root) then
    do i = 1, SOME_MAX
       sendbuf(i) = i-1
    end do
  endif

  ! All processes call MPI_SCATTER
  call MPI_SCATTER(sendbuf, 2, MPI_INTEGER, &
                   recvbuf, 2, MPI_INTEGER, &
                   root, MPI_COMM_WORLD, ierr)

  print *, "(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program scatter
```

MPI_Scatter Example

C Scatter

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#define SOME_MAX 8
int main(int argc, char *argv[]){
  int sendbuf[SOME_MAX], recvbuf[2];
  int i, rank, numprocs;
  int root = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == root){
    for(i = 0; i < SOME_MAX; i++){
      sendbuf[i] = i;
    }
  }
  // All processes call MPI_Scatter
  MPI_Scatter(&sendbuf[0], 2, MPI_INT,
              &recvbuf[0], 2, MPI_INT,
              root, MPI_COMM_WORLD);
  printf("(Rank %i): recvbuf: %i %i\n",
         rank, recvbuf[0], recvbuf[1]);

  MPI_Finalize();
}
```

MPI_Scatter Example Output

Run with ibrun -np 4 ./a.out

Fortran Output

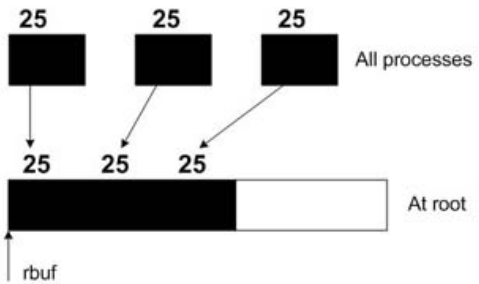
```
(Rank           0 ): recvbuf:           0           1
(Rank           1 ): recvbuf:           2           3
(Rank           2 ): recvbuf:           4           5
(Rank           3 ): recvbuf:           6           7
```

C Output

```
(Rank 0): recvbuf: 0 1
(Rank 1): recvbuf: 2 3
(Rank 2): recvbuf: 4 5
(Rank 3): recvbuf: 6 7
```

Basic Data Movement – Gather (MPI_Gather)

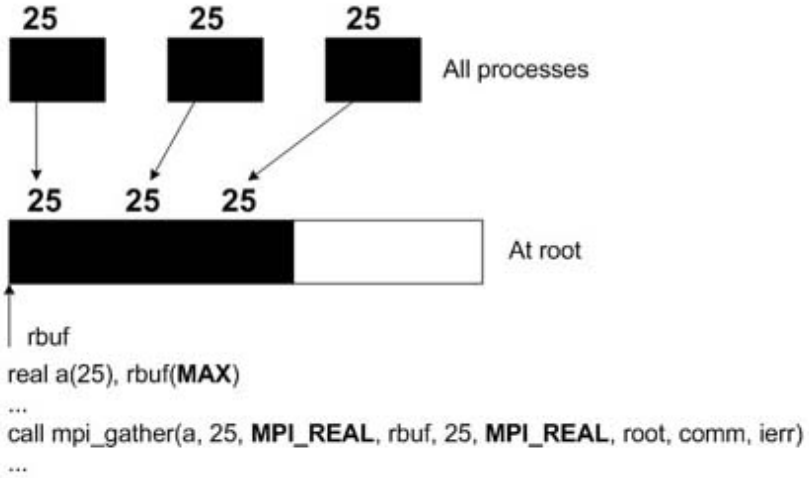
C	`int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )`
Fortran	`MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)`




```
real a(25), rbuf(MAX)
...
call mpi_gather(a, 25, MPI_REAL, rbuf, 25, MPI_REAL, root, comm, ierr)
...
```

- Inverse of MPI_Scatter
- Array is scattered across n processes
- Gather all pieces of array onto root process

MPI_Gather Example



- Create a local array on all processes and populate them
- Send the local array from all processes to the root process
- Similar to the figure above

MPI_GATHER Example

Fortran Gather

```
program gather
  use mpi
  implicit none
  integer, parameter :: SOME_MAX = 8
  integer sendbuf(2), recvbuf(SOME_MAX)
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  do i = 1, 2
    sendbuf(i) = (100 * rank) + rank + 1
  end do

  ! All processes call MPI_GATHER
  call MPI_GATHER(sendbuf, 2, MPI_INTEGER, &
                  recvbuf, 2, MPI_INTEGER, &
                  root, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program gather
```

MPI_Gather Example

C Gather

```c
#include <stdio.h>
#include "mpi.h"
#define SOME_MAX 8
int main(int argc, char *argv[]){
  int sendbuf[2], recvbuf[SOME_MAX] = { 0 };
  int i, rank, numprocs;
  int root = 3;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for(i = 0; i < 2; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }

  // All processes call MPI_Scatter
  MPI_Gather(&sendbuf[0], 2, MPI_INT,
             &recvbuf[0], 2, MPI_INT,
             root, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");

  MPI_Finalize();
}
```

MPI_Gather Example Output

Run with ibrun -np 4 ./a.out

Fortran Output – Only root process receives data

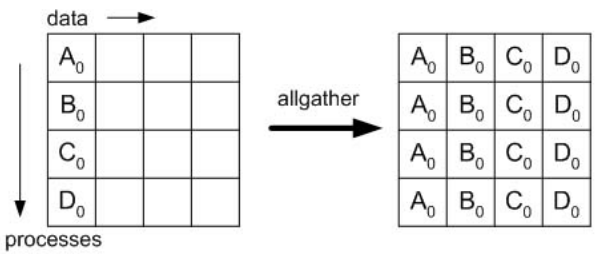
```
(Rank 1 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3 ): recvbuf: 1 1 102 102 203 203 304 304
```

C Output – Only root process receives data

```
(Rank 1): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3): recvbuf: 1 1 102 102 203 203 304 304
```

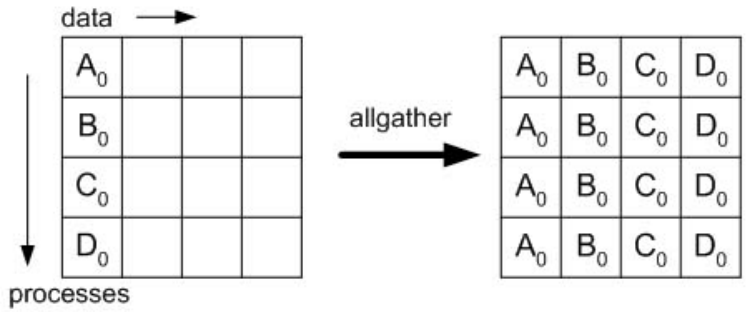
Data Movement – AllGather (MPI_Allgather)

C	`int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
Fortran	`MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)`



- MPI_Gather except all processes receive result
- No root process
- Gather all pieces of array onto all processes involved

MPI_Allgather Example



- Create a local array on all processes and populate them
- Send the local array from all processes to every process in turn
- Conceptually, picture a gather followed by a broadcast
- Similar to the figure above

MPI_ALLGATHER Example

Fortran Allgather

```
program allgather
  use mpi
  implicit none
  integer, parameter :: SOME_MAX = 8
  integer sendbuf(2), recvbuf(SOME_MAX)
  integer i, rank, numprocs, ierr
  recvbuf = 0

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  do i = 1, 2
    sendbuf(i) = (100 * rank) + rank + 1
  end do

  ! No root process for MPI_ALLGATHER
  call MPI_ALLGATHER(sendbuf, 2, MPI_INTEGER, &
                     recvbuf, 2, MPI_INTEGER, &
                     MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program allgather
```

MPI_Allgather Example

C Allgather

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#define SOME_MAX 8
int main(int argc, char *argv[]){
  int sendbuf[2], recvbuf[SOME_MAX] = { 0 };
  int i, rank, numprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for(i = 0; i < 2; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }

  // No root process for MPI_Allgather
  MPI_Allgather(&sendbuf[0], 2, MPI_INT,
                &recvbuf[0], 2, MPI_INT,
                MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");

  MPI_Finalize();
}
```

MPI_Allgather Example Output

Run with ibrun -np 4 ./a.out

Fortran Output – All processes receive data

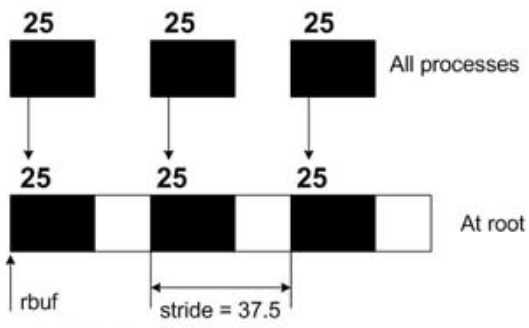
```
(Rank 0 ): recvbuf: 1 1 102 102 203 203 304 304
(Rank 1 ): recvbuf: 1 1 102 102 203 203 304 304
(Rank 2 ): recvbuf: 1 1 102 102 203 203 304 304
(Rank 3 ): recvbuf: 1 1 102 102 203 203 304 304
```

C Output – All processes receive data

```
(Rank 0): recvbuf: 1 1 102 102 203 203 304 304
(Rank 1): recvbuf: 1 1 102 102 203 203 304 304
(Rank 2): recvbuf: 1 1 102 102 203 203 304 304
(Rank 3): recvbuf: 1 1 102 102 203 203 304 304
```

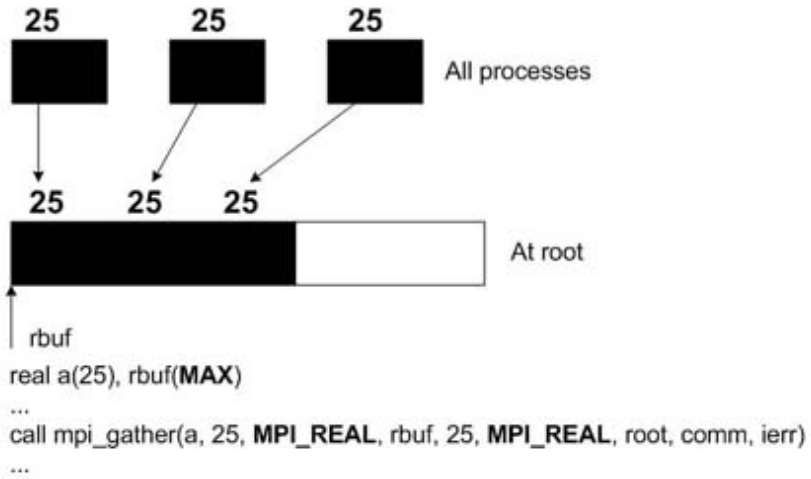
Data Movement – Gatherv (MPI_Gatherv)

C	`int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int displacement[], MPI_Datatype recvtype, int root, MPI_Comm comm)`
Fortran	`MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displacement, recvtype, root, comm, ierr)`



- "Variable" MPI_Gather
- May have a variable stride in between received buffers
- May have a variable sized receive buffer
- recvcount (was a scalar) is now an array for each process
- New variable "displacement" for determining recvbuf locations

MPI_Gatherv Example #1



- Use MPI_Gather Example #1 as basis – should get the same answer
- Each process will send an array of length 2 to the root process (as before)
- Insert a stride of length 2 in between sections of the receiving array
- Total array length is still 8 (4 calling processes * 2 stride length)
- Similar behavior to above figure (regular Gather)

MPI_GATHERV Example #1

Fortran Gatherv – Set Up to Match MPI_GATHER Ex.

```
program gatherv1
  use mpi
  implicit none
  integer, parameter :: NPROCS = 4
  integer, parameter :: SOME_MAX = 8
  integer sendbuf(2), recvbuf(SOME_MAX)
  integer recvcount(NPROCS)     ! Num of elements received from each process
  integer displacement(NPROCS) ! place data from process i at recvbuf(displacement(i))
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  do i = 1, 2
    sendbuf(i) = (100 * rank) + rank + 1
  end do
  ! recvbuf, recvcounts, displacement, recvtype are
  ! significant for the root process only.
  if(rank == root)then
    do i = 1, numprocs
      recvcount(i)    = 2 ! Recv 2 elements from each process
      displacement(i) = 2 * (i-1) ! Skip 2 elements
    end do
  end if

  call MPI_GATHERV(sendbuf, 2, MPI_INTEGER, &
                   recvbuf, recvcount, displacement, MPI_INTEGER, &
                   root, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf
  call MPI_FINALIZE(ierr)
end program gatherv1
```

MPI_Gatherv Example #1

C Gatherv – Set Up to Match MPI_Gather Ex.

```c
#include <stdio.h>
#include "mpi.h"
#define SOME_MAX 8
#define NPROCS 4
int main(int argc, char *argv[]){
  int sendbuf[2], recvbuf[SOME_MAX] = { 0 };
  int recvcount[NPROCS]; // Num of elements received from each process
  int displacement[NPROCS]; // Place data from process i at recvbuf[displacement[i]]
  int i, rank, numprocs, root = 3;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i = 0; i < 2; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }
  // recvbuf, recvcounts, displacement, recvtype are
  // significant for the root process only.
  if(rank == root){
    for(i = 0; i < numprocs; i++){
      recvcount[i] = 2; // Recv 2 elements from each process
      displacement[i] = 2 * i; // Skip 2 elements
   } } // squish

  // All processes call MPI_Scatter
  MPI_Gatherv(&sendbuf[0], 2, MPI_INT,
              &recvbuf[0], recvcount, displacement, MPI_INT,
              root, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");
  MPI_Finalize();
```

MPI_Gatherv Example #1 Output

Run with ibrun -np 4 ./a.out

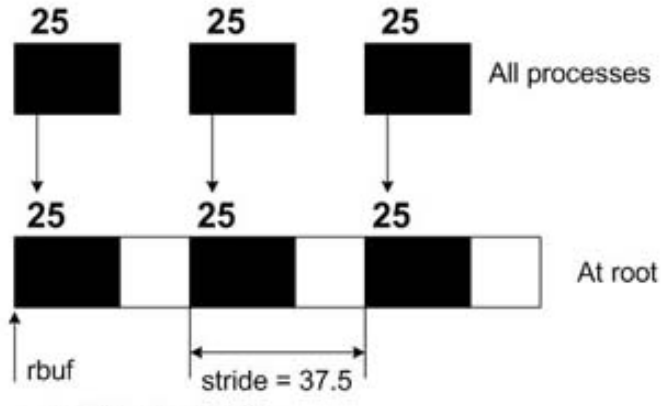
Fortran Output – Set Up to Match MPI_GATHER Ex. Output

```
(Rank 1 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3 ): recvbuf: 1 1 102 102 203 203 304 304
```

C Output – Set Up to Match MPI_Gather Ex. Output

```
(Rank 1): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3): recvbuf: 1 1 102 102 203 203 304 304
```

MPI_Gatherv Example #2



- Use MPI_Gatherv Example #1 as basis
- Each process will send an array of length 2 to the root process (as before)
- This time, insert a stride of length 3 in between sections of the receiving array
- Total array length is now 12 (4 calling processes * 3 stride length)
- Similar behavior to above figure

MPI_GATHERV Example #2

Fortran Gatherv – Stride >Length

```
program gatherv2
  use mpi
  implicit none
  integer, parameter :: NPROCS = 4
  integer, parameter :: SOME_MAX = 12
  integer sendbuf(2), recvbuf(SOME_MAX)
  integer recvcount(NPROCS)     ! Num of elements received from each process
  integer displacement(NPROCS) ! place data from process i at recvbuf(displacement(i))
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  do i = 1, 2
    sendbuf(i) = (100 * rank) + rank + 1
  end do
  ! recvbuf, recvcounts, displacement, recvtype are
  ! significant for the root process only.
  if(rank == root)then
    do i = 1, numprocs
      recvcount(i)    = 2 ! Recv 2 elements from each process
      displacement(i) = 3 * (i-1) ! Skip 3 elements
    end do
  end if

  call MPI_GATHERV(sendbuf, 2, MPI_INTEGER, &
                   recvbuf, recvcount, displacement, MPI_INTEGER, &
                   root, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf
  call MPI_FINALIZE(ierr)
end program gatherv2
```

MPI_Gatherv Example #2

C Gatherv – Stride >Length

```
#include <stdio.h>
#include "mpi.h"
#define SOME_MAX 12
#define NPROCS 4
int main(int argc, char *argv[]){
  int sendbuf[2], recvbuf[SOME_MAX] = { 0 };
  int recvcount[NPROCS]; // Num of elements received from each process
  int displacement[NPROCS]; // Place data from process i at recvbuf[displacement[i]]
  int i, rank, numprocs;
  int root = 3;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i = 0; i < 2; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }
  // recvbuf, recvcounts, displacement, recvtype are
  // significant for the root process only.
  if(rank == root){
    for(i = 0; i < numprocs; i++){
      recvcount[i] = 2; // Recv 2 elements from each process
      displacement[i] = 3 * i; // Skip 3 elements
  } } //squish

  // All processes call MPI_Scatter
  MPI_Gatherv(&sendbuf[0], 2, MPI_INT,
              &recvbuf[0], recvcount, displacement, MPI_INT,
              root, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");
  MPI_Finalize();
```

MPI_Gatherv Example #2 Output

Run with ibrun -np 4 ./a.out

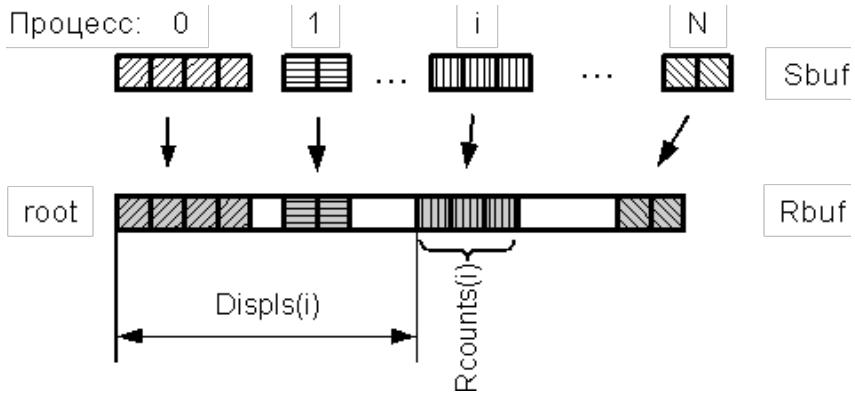
Fortran Output – Now a zero separating each received buffer

```
(Rank 1 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 2 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 0 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 3 ): recvbuf: 1 1 0 102 102 0 203 203 0 304 304 0
```

C Output – Now a zero separating each received buffer

```
(Rank 1): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 2): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 0): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 3): recvbuf: 1 1 0 102 102 0 203 203 0 304 304 0
```

MPI_Gatherv Example #3



- Use MPI_Gatherv Example #2 as basis
- Each process will send an array of length (rank+1) to the root process
- The stride is set to maintain one zero element in between sections of the receiving arrays
- Total array length is now 13 (4 calling processes)
- Similar behavior to above figure

MPI_GATHERV Example #3

Fortran Gatherv – Variable Recvcounts

```
program gatherv3
  use mpi
  implicit none
  integer, parameter :: NPROCS = 4
  integer, parameter :: SOME_MAX = 13
  integer sendbuf(NPROCS), recvbuf(SOME_MAX)
  integer recvcount(NPROCS)     ! Num of elements received from each process
  integer displacement(NPROCS) ! place data from process i at recvbuf(displacement(i))
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  do i = 1, NPROCS
    sendbuf(i) = (100 * rank) + rank + 1
  end do
  ! recvbuf, recvcounts, displs, recvtype are significant
  ! for the root process only.
  if(rank == root)then
    do i = 1, numprocs
      recvcount(i)    = i ! Recv i=rank elements from each process
      displacement(i) = sum(recvcount(1:i))-1 ! offset based on sum
    end do
  end if

  call MPI_GATHERV(sendbuf, rank+1, MPI_INTEGER, &
                   recvbuf, recvcount, displacement, MPI_INTEGER, &
                   root, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf
  call MPI_FINALIZE(ierr)
end program gatherv3
```

MPI_Gatherv Example #3

C Gatherv – Variable Recvcounts

```
#include <stdio.h>
#include "mpi.h"
#define SOME_MAX 13
#define NPROCS 4
int main(int argc, char *argv[]){
  int sendbuf[NPROCS], recvbuf[SOME_MAX] = { 0 };
  int recvcount[NPROCS]; // Num of elements received from each process
  int displacement[NPROCS]; // Place data from process i at recvbuf[displacement[i]]
  int i, rank, numprocs, sum;
  int root = 3;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i = 0; i < NPROCS; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }
  // recvbuf, recvcounts, displacement, recvtype are
  // significant for the root process only.
  if(rank == root){
    for(i = 0; i < numprocs; i++){
      recvcount[i] = (i+1); // Recv i=rank elements from each process
      sum += (i+1); displacement[i] = sum-1 ; // Offset based on sum
  } } // squish

  // All processes call MPI_Scatter
  MPI_Gatherv(&sendbuf[0], rank+1, MPI_INT,
              &recvbuf[0], recvcount, displacement, MPI_INT,
              root, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");
  MPI_Finalize();
```

MPI_Gatherv Example #3 Output

Run with ibrun -np 4 ./a.out

Fortran Output – Now recv size varies with rank

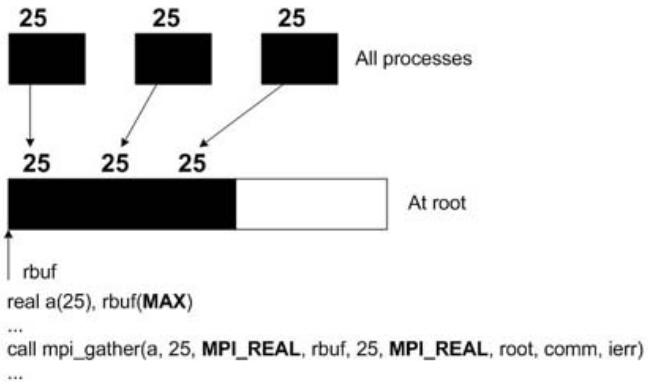
```
(Rank 1 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 2 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 0 ): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 3 ): recvbuf: 1 0 102 102 0 203 203 203 0 304 304 304 304
```

C Output – Now recv size varies with rank

```
(Rank 1): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 2): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 0): recvbuf: 0 0 0 0 0 0 0 0 0 0 0 0 0
(Rank 3): recvbuf: 1 0 102 102 0 203 203 203 0 304 304 304 304
```

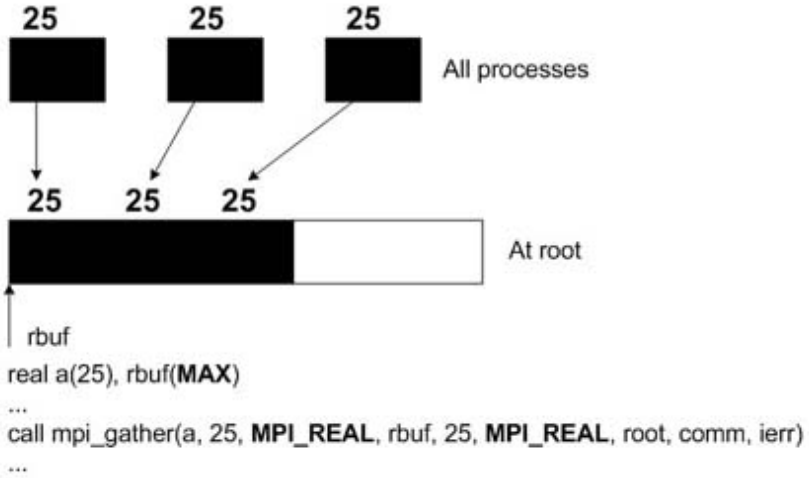
Data Movement – Igather (MPI_Igather)

C	`int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)`
Fortran	`MPI_IGATHER(sendbuf, sendcout, sendtype, recvbuf, recvcount, recvtype, root, comm, request, ierr)`



- Non-blocking version of MPI_Gather
- Array is scattered across n processes
- Gather all pieces of array onto root process
- Initiates the communication and returns; it does not wait (block)!
- Use with MPI_Wait and its variants or MPI_Test and it's variants to ensure data is safe to operate on

MPI_Igather Example



- Create a local array on all processes and populate them
- Send the local array from all processes to the root process
- Similar to the figure above

MPI_IGATHER Example

Fortran Igather

```
program igather
  use mpi
  implicit none
  integer, parameter :: SOME_MAX = 8
  integer sendbuf(2), recvbuf(SOME_MAX)
  integer req ! Request handle to query when transfer has completed
  integer stat(MPI_STATUS_SIZE) ! Holds information on function call
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  do i = 1, 2
    sendbuf(i) = (100 * rank) + rank + 1
  end do

  ! All processes call MPI_GATHER
  call MPI_IGATHER(sendbuf, 2, MPI_INTEGER, &
                   recvbuf, 2, MPI_INTEGER, &
                   root, MPI_COMM_WORLD, req, ierr)
  ! Potentially do work here
  call MPI_WAIT(req, stat, ierr) ! Wait until data has safely transferred

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program igather
```

MPI_IGather Example

C Igather

```c
#include <stdio.h>
#include "mpi.h"
#define SOME_MAX 8
int main(int argc, char *argv[]){
  int sendbuf[2], recvbuf[SOME_MAX] = { 0 };
  int i, rank, numprocs;
  MPI_Request req;
  MPI_Status stat;
  int root = 3;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for(i = 0; i < 2; i++){
    sendbuf[i] = (100 * rank) + rank + 1;
  }

  // All processes call MPI_Igather
  MPI_Igather(&sendbuf[0], 2, MPI_INT,
              &recvbuf[0], 2, MPI_INT,
              root, MPI_COMM_WORLD, &req);
  // Potentially do work here
  MPI_Wait(&req, &stat); // Wait for transfer to safely complete

  printf("(Rank %i): recvbuf:", rank);
  for(i = 0; i < SOME_MAX; i++){
    printf(" %i",recvbuf[i]);
  } printf("\n");

  MPI_Finalize();
}
```

MPI_Igather Example Output

Run with ibrun -np 4 ./a.out

Fortran Output – Only root process receives data

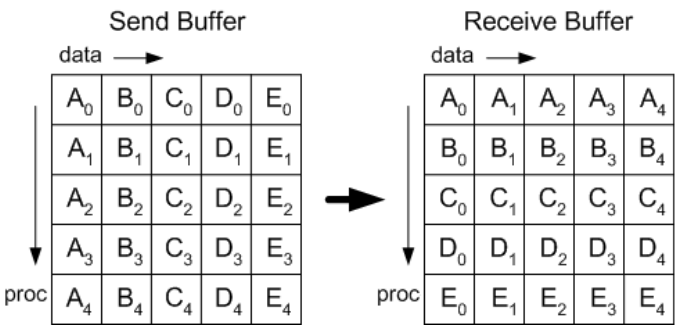
```
(Rank 1 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0 ): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3 ): recvbuf: 1 1 102 102 203 203 304 304
```

C Output – Only root process receives data

```
(Rank 1): recvbuf: 0 0 0 0 0 0 0 0
(Rank 2): recvbuf: 0 0 0 0 0 0 0 0
(Rank 0): recvbuf: 0 0 0 0 0 0 0 0
(Rank 3): recvbuf: 1 1 102 102 203 203 304 304
```

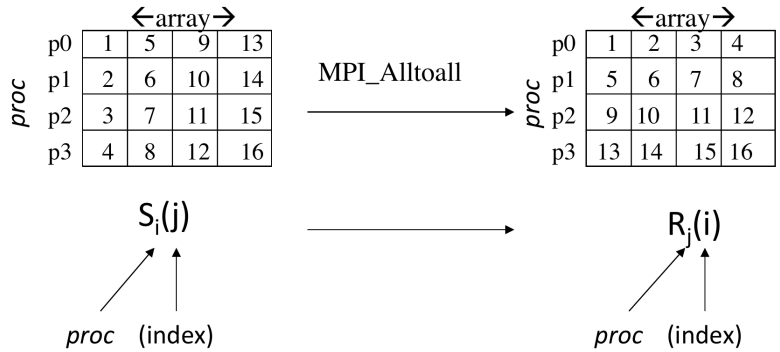
Data Movement – Alltoall (MPI_Alltoall)

C	`int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
Fortran	`MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)`



- An extension to MPI_Allgather where each process sends distinct data to each receiver
- Useful for matrix transposes or Fast Fourier Transforms (FFTs)
- Same specification as MPI_Allgather, except sendbuf must contain sendcount*NPROC elements

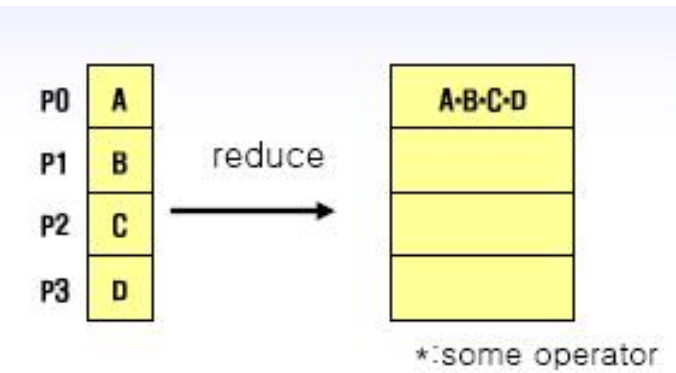
MPI_Alltoall Example



- Create a local array on all processes and populate them
- Send the local array from all processes to all processes
- Similar to the figure above
- We'll pass on a code example for now

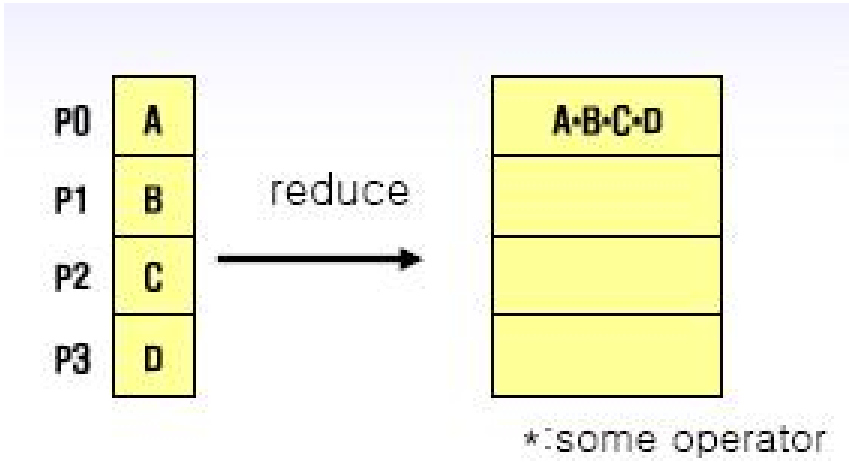
Basic Global Computation – Reduction (MPI_Reduce)

C	`int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype sendtype, MPI_Op op, int root, MPI_Comm comm)`
Fortran	`MPI_REDUCE(sendbuf, recvbuf, count, sendtype, op, root, comm, ierr)`



- All processes call MPI_Reduce
- The root process collects each recvbuf and performs an operation (+,*,min,max,..)
- The root process stores the output

MPI_Reduce Example



- Create a local value on each process
- Use the sum reduction operator to collect the sum across processes
- Similar to the figure above

MPI_REDUCE Example

Fortran Sum Reduction

```
program reduce
  use mpi
  implicit none
  integer, parameter :: SOME_MAX = 8
  integer sendbuf, recvbuf
  integer i, rank, numprocs, root, ierr
  data root/3/
  recvbuf = 0

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  sendbuf = rank

  ! All processes call MPI_REDUCE
  call MPI_REDUCE(sendbuf, recvbuf, 1, MPI_INTEGER, &
                  MPI_SUM, root, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program reduce
```

MPI_Reduce Example

C Sum Reduction

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]){
  int sendbuf, recvbuf = 0 ;
  int i, rank, numprocs;
  int root = 3;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  sendbuf = rank;

  // All processes call MPI_Reduce
  MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT,
             MPI_SUM, root, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  printf(" %i",recvbuf);
  printf("\n");

  MPI_Finalize();
}
```

MPI_Reduce Example Output

Run with ibrun -np 4 ./a.out

Fortran Output

```
(Rank            0 ): recvbuf:           0
(Rank            1 ): recvbuf:           0
(Rank            2 ): recvbuf:           0
(Rank            3 ): recvbuf:           6
```

C Output

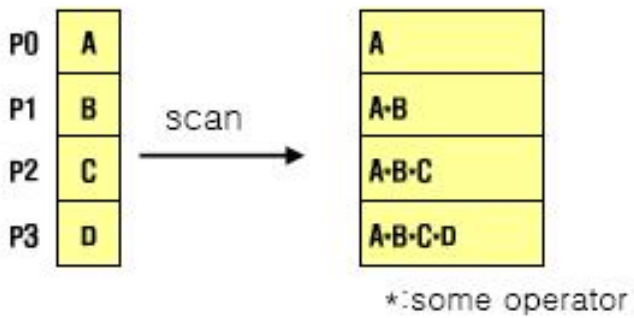
```
(Rank 0): recvbuf: 0
(Rank 1): recvbuf: 0
(Rank 2): recvbuf: 0
(Rank 3): recvbuf: 6
```

MPI Defined Reduction Operations

MPI_PROD	Product
MPI_SUM	Sum
MPI_**L**AND	**L**ogical and
MPI_**L**OR	**L**ogical or
MPI_**L**XOR	**L**ogical exclusive or
MPI_**B**AND	**B**itwise and
MPI_**B**OR	**B**itwise or
MPI_**B**XOR	**B**itwise exclusive or
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

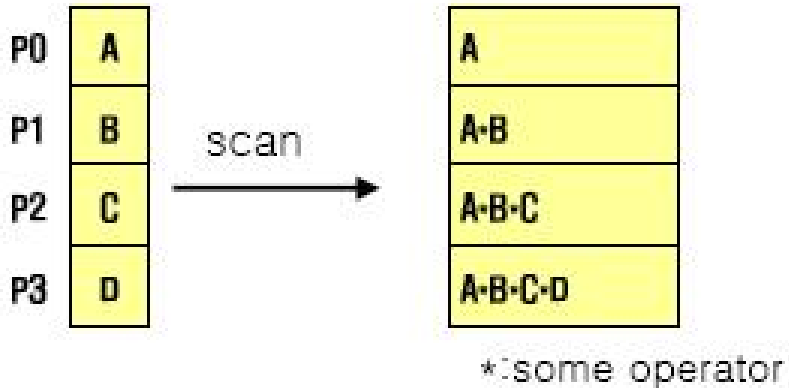
Basic Global Computation – Scan (MPI_Scan)

C	`int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
Fortran	`MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm, ierr)`



- All processes call MPI_Scan
- Performs partial reductions on distributed data
- No root process
- Conceptually, imagine MPI_Reduce leaving its intermediate work behind

MPI_Scan Example



- Create local value (equal to rank) on each process
- Use the sum reduction operator to collect the cumulative sum locally with scan
- Similar to the figure above

MPI_SCAN Example

Fortran Scan Sum Reduction

```
program myscan
  use mpi
  implicit none
  integer sendbuf, recvbuf
  integer i, rank, numprocs, ierr
  recvbuf = 0

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

  sendbuf = rank

  ! All processes call MPI_SCAN
  call MPI_scan(sendbuf, recvbuf, 1, MPI_INTEGER, &
                MPI_SUM, MPI_COMM_WORLD, ierr)

  write(*,*)"(Rank ", rank ,"): recvbuf: ", recvbuf

  call MPI_FINALIZE(ierr)
end program myscan
```

MPI_Scan Example

C Scan Sum Reduction

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]){
  int sendbuf, recvbuf = 0 ;
  int i, rank, numprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  sendbuf = rank;

  // All processes call MPI_Scan
  MPI_Scan(&sendbuf, &recvbuf, 1, MPI_INT,
           MPI_SUM, MPI_COMM_WORLD);

  printf("(Rank %i): recvbuf:", rank);
  printf(" %i",recvbuf);
  printf("\n");

  MPI_Finalize();
}
```

MPI_Scan Example Output

Run with ibrun -np 4 ./a.out

Fortran Output

```
(Rank            0 ): recvbuf:           0
(Rank            1 ): recvbuf:           1
(Rank            2 ): recvbuf:           3
(Rank            3 ): recvbuf:           6
```

C Output

```
(Rank 0): recvbuf: 0
(Rank 1): recvbuf: 1
(Rank 2): recvbuf: 3
(Rank 3): recvbuf: 6
```

Summary

- Basic routines include Bcast, Scatter, Gather, Reduce
- "All" variants perform operation and distribute to all processes
- "Variable" variants allow for generalized size and placement of data
- "Initiate" variants are the non-blocking cousins to their blocking counterparts
- Variants are usually combined together with a basic routine (exception: Alltoall)
- In several cases, variants can also stack (e.g. Iallgatherv)

References

- Victor Eijkhout, "Introduction to High Performance Scientific Computing"
- Victor Eijkhout, "Parallel Computing for Science and Engineering"
- Blaise Barney, "MPI Tutorial"
- Mark Lubin, "Introduction into new features of MPI-3.0 Standard"
- Brandon Barker, "MPI Collective Communications"
- "MPI: A Message-Passing Interface Standard Version-3.0"