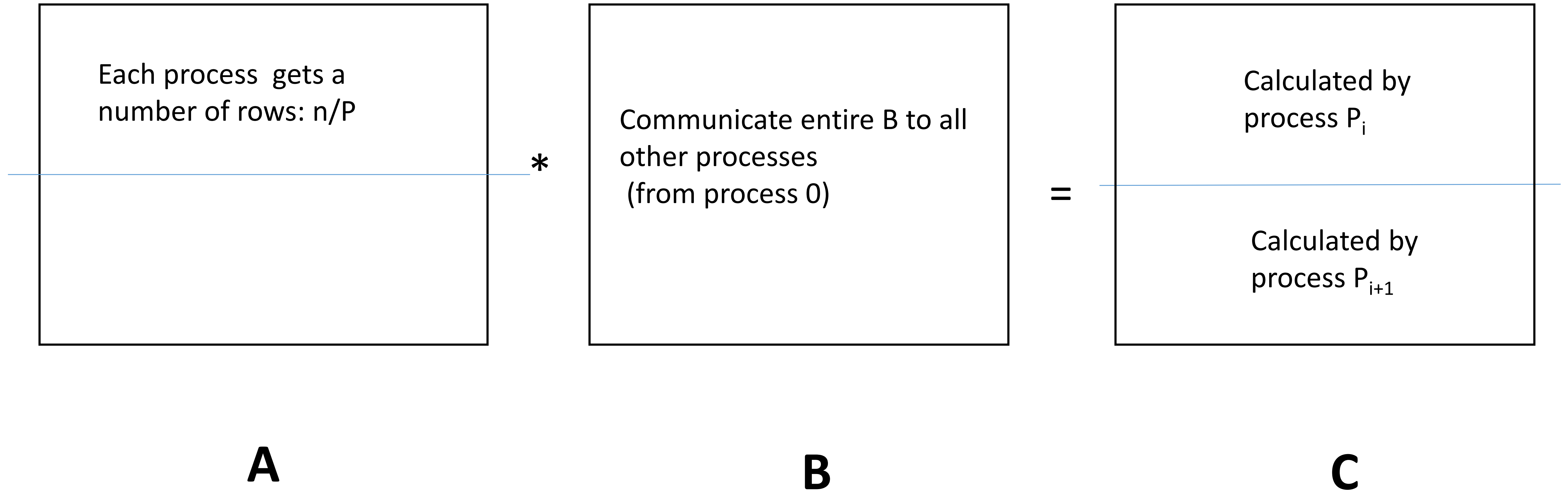
$C=A*B$


Each process gets a number of rows: n/P
*
Communicate entire B to all other processes (from process 0)
=
Calculated by process P_i
Calculated by process P_{i+1}
A
B
C

```
double  a[NRA][NCA],         /* matrix A to be multiplied */
        b[NCA][NCB],         /* matrix B to be multiplied */
        c[NRA][NCB];         /* result matrix C */

…

/* Parallelize the computation of the following matrix-matrix
multiplication.
   How to partition and distribute the initial matrices, the work, and
collecting
   final results.
 */

 for (i=0; i<NRA; i++)
  {
  for(j=0; j<NCB; j++)
   for (k=0; k<NCA; k++)
    c[i][j] += a[i][k] * b[k][j];
  }
```

SPMD programming style:

```
if (myID == 0)
{
    For i=1..(P-1) Send n/P rows of A and B to
    P(i);
}
else
{
    Receive n/P rows of A and B from P(0);
}
Calculate part of A(i)*B, and send back
```

What?

```
MPI_Send ( void *data, int count, MPI_Datatype type ,
           int dest, int tag, MPI_Comm comm )
```

To whom?

C

Pointers and contents: &, *, (int *) a, (double *) b, etc.

```
double a;
MPI_Send(&a, 1, ….);
```

```
double *a;
MPI_Send(a, 1, ….)
```

```
double b[100];
MPI_Send(&b[0], 100, ….);
Or
MPI_Send(b, 100, …)
```

Function: call by reference and call by value

MPI processes (with local variables), similarity with call by value

MPI Collective Communications

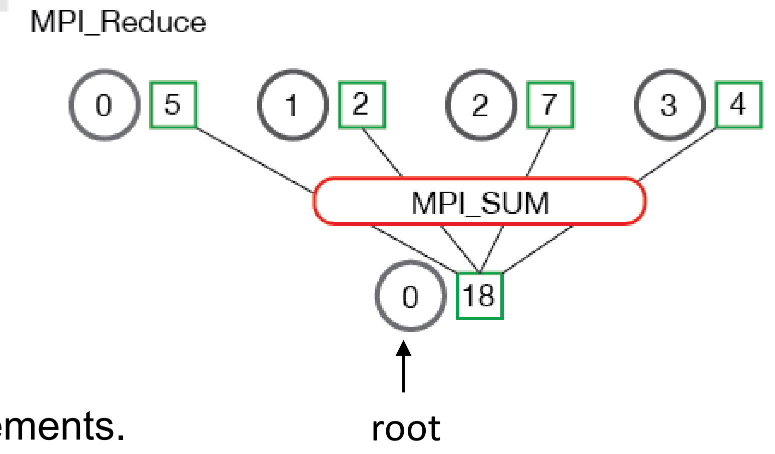
- MPI_Reduce
- MPI_Allreduce
- MPI_Gather, MPI_Gatherv
- MPI_Scatter, MPI_Scatterv

MPI_Reduce

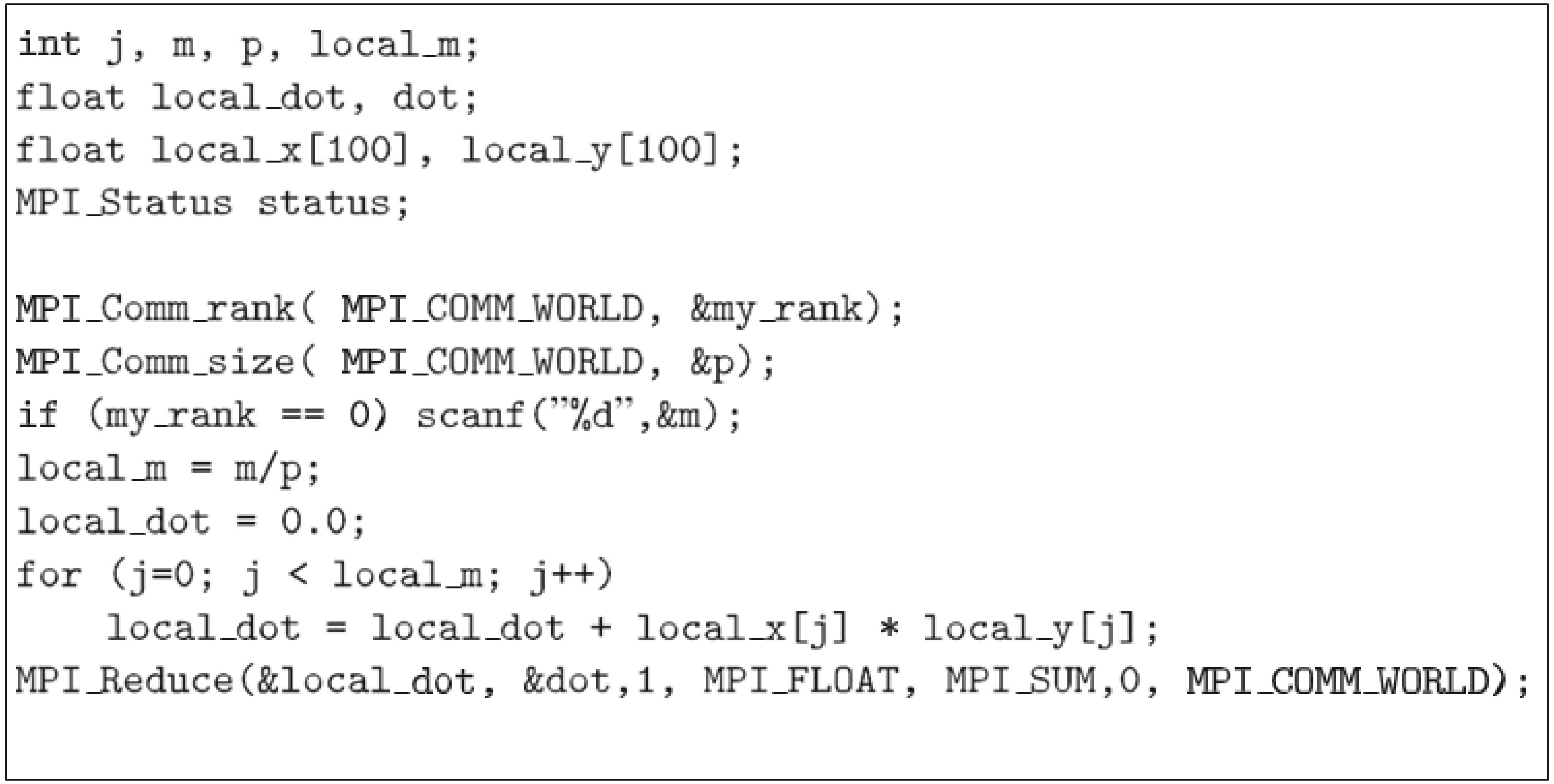
```
MPI_Reduce( void* send_data, void* recv_data, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
communicator)
```

The reduction operations defined by MPI include:

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.



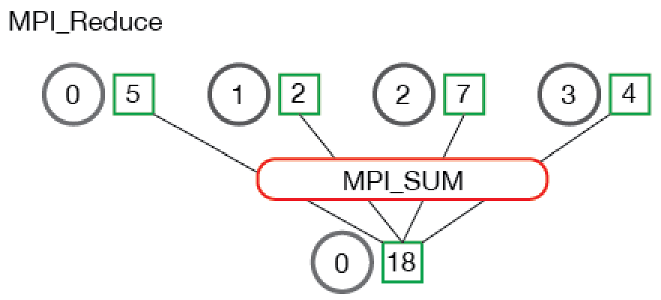
Dot product using reduction

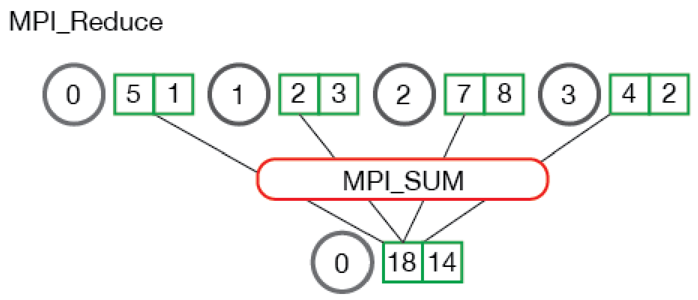


```
int j, m, p, local_m;
float local_dot, dot;
float local_x[100], local_y[100];
MPI_Status status;

MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
MPI_Comm_size( MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf(”%d”,&m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j < local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce(&local_dot, &dot,1, MPI_FLOAT, MPI_SUM,0, MPI_COMM_WORLD);
```

MPI program for the parallel computation of a scalar product

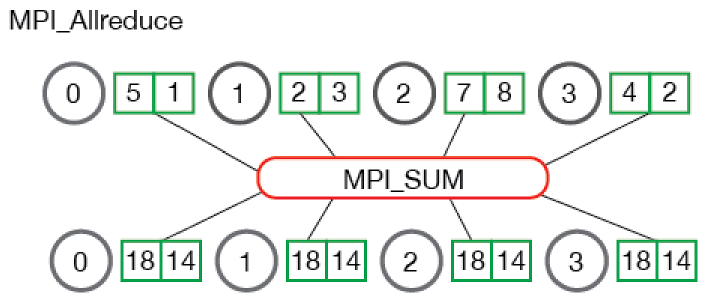
MPI_Reduce and MPI_Allreduce





```
MPI_Reduce(&local,&global_sum,1,MPI_INT,

MPI_SUM,0,MPI_COMM_WORLD)
```

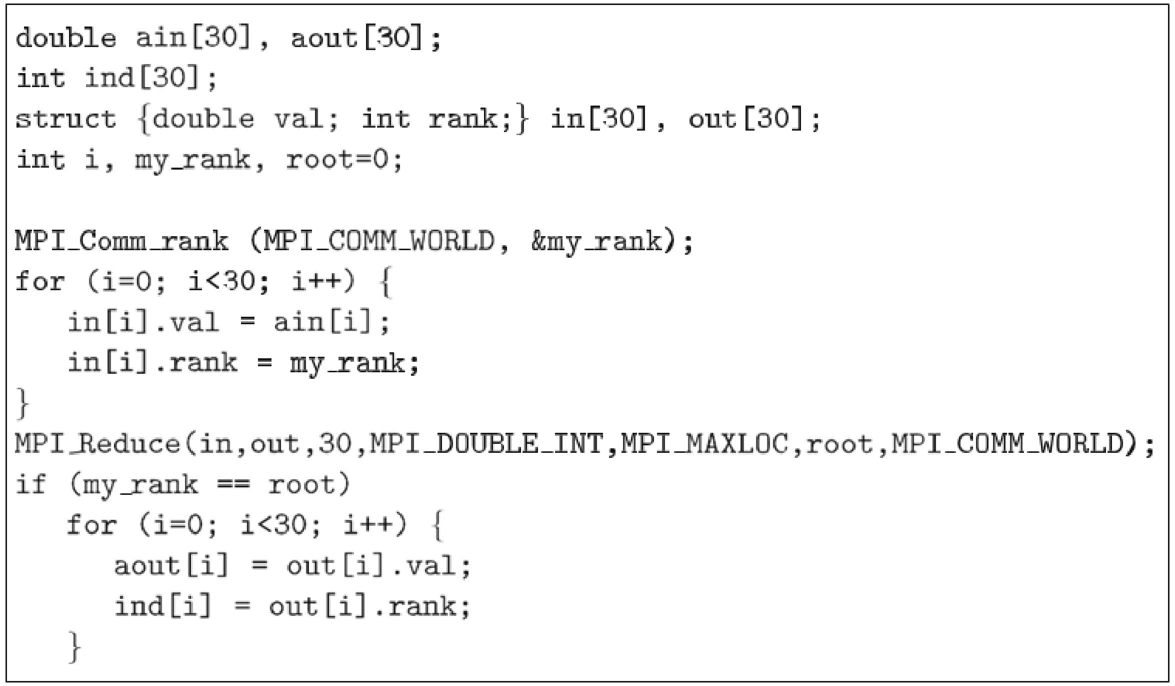
```
MPI_Reduce(&local,&global_sum,2,MPI_INT,

MPI_SUM,0,MPI_COMM_WORLD)
```



```
MPI_Allreduce(&local,&global_sum,2,MPI_INT,
              MPI_SUM,MPI_COMM_WORLD)
```

MPI Reduction operation

```
int MPI_Reduce (void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype type,
                MPI_Op op,
                int root,
                MPI_Comm comm)
```



```
double ain[30], aout[30];
int ind[30];
struct {double val; int rank;} in[30], out[30];
int i, my_rank, root=0;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
for (i=0; i<30; i++) {
   in[i].val = ain[i];
   in[i].rank = my_rank;
}
MPI_Reduce(in,out,30,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
if (my_rank == root)
   for (i=0; i<30; i++) {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;
   }
```

Example for the use of `MPI_Reduce()` using `MPI_MAXLOC` for the reduction operator.

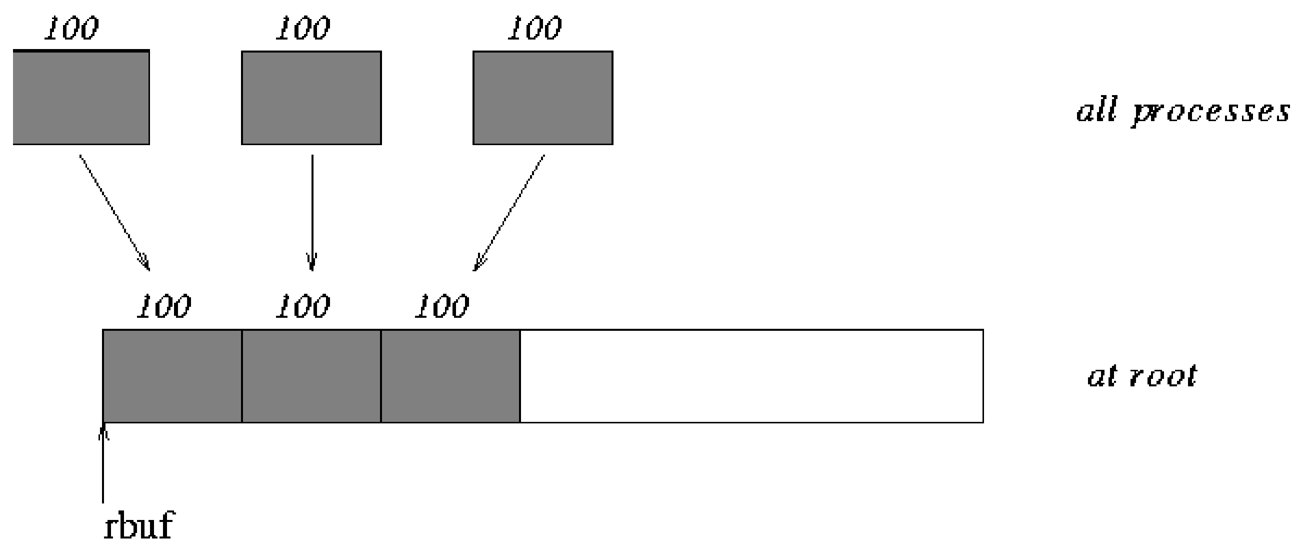
MPI_Gather and MPI_Scatter

MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Examples from www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70

Example:
The root process gathers 100 ints from each process in the group.

```
MPI_Comm comm;
int NrProcs,sendarray[100];
int root, myrank, *rbuf; ...
MPI_Comm_rank( comm, myrank);
if ( myrank == root)
{
  MPI_Comm_size( comm, &NrProcs);
  rbuf = (int *)malloc(NrProcs*100*sizeof(int));
 }
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

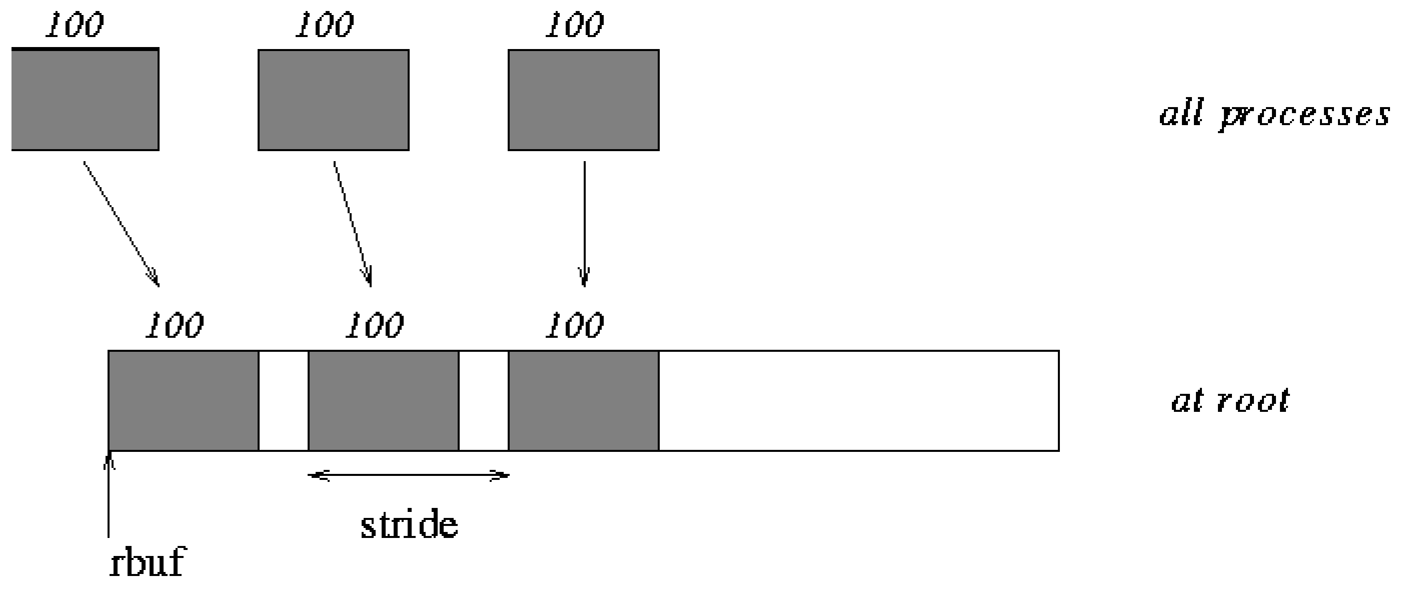


MPI_Gatherv

Example

Now have each process send 100 ints to root, but place each set (of 100) *stride* ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume *stride*≥100 .

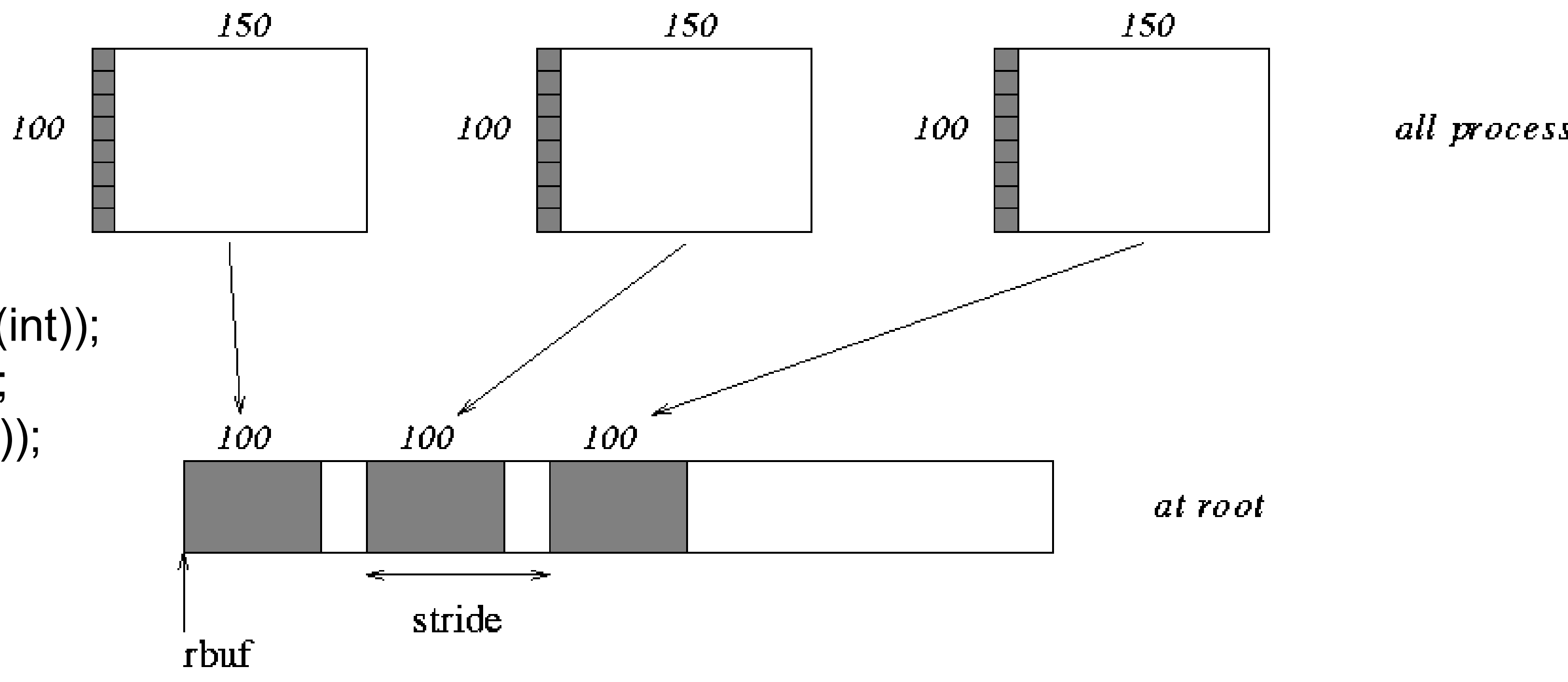
```
MPI_Comm comm;
int NrProcs, sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;
...
MPI_Comm_size( comm, &NrProcs);
rbuf = (int *)malloc(NrProcs*stride*sizeof(int));
displs = (int *)malloc(NrProcs*sizeof(int));
for (i=0; i<NrProcs; ++i)
{
   displs[i] = i*stride;
   rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT, root, comm);
```



MPI_Gatherv

Example
Same as Example as previous MPI_Gatherv, but now send the 100 ints from the 0th column of a 100×150 int array, in C.

```
MPI_Comm comm;
int NrProcs,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;
...
rbuf = (int *)malloc(NrProcs*stride*sizeof(int));
displs = (int *)malloc(NrProcs*sizeof(int));
rcounts = (int *)malloc(NrProcs*sizeof(int));
for (i=0; i<NrProcs; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array */
MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

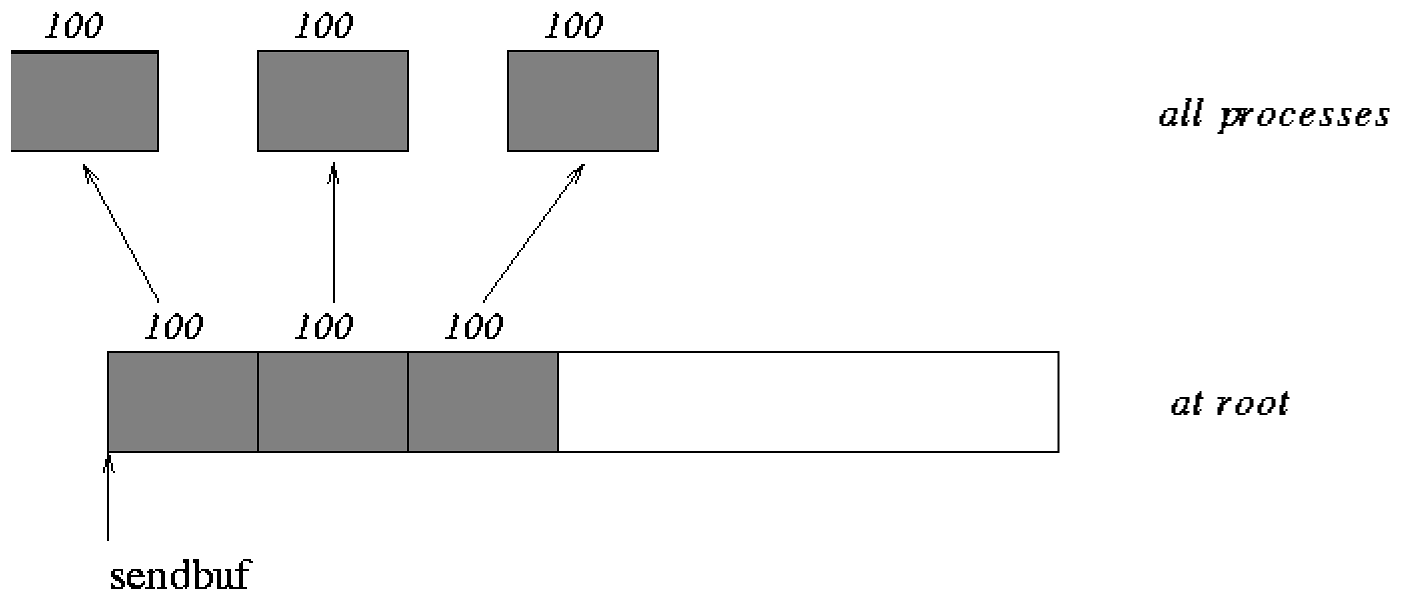


MPI_Scatter

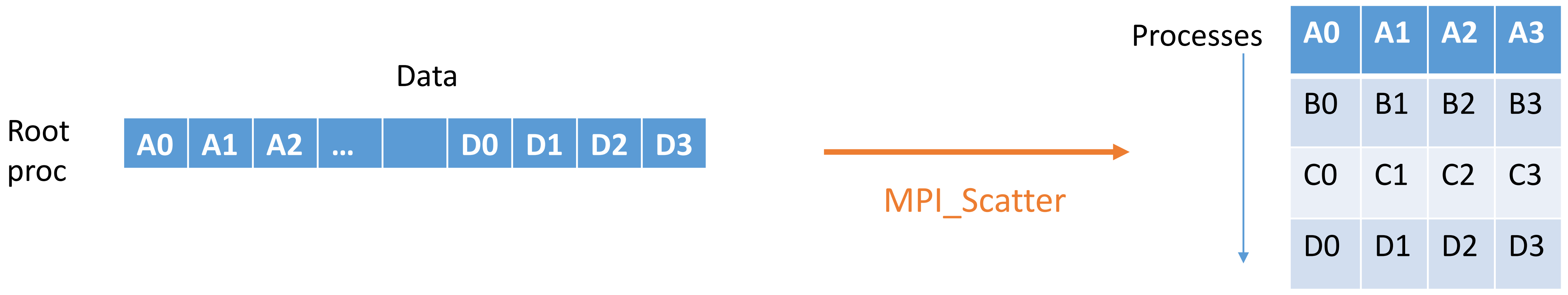
MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

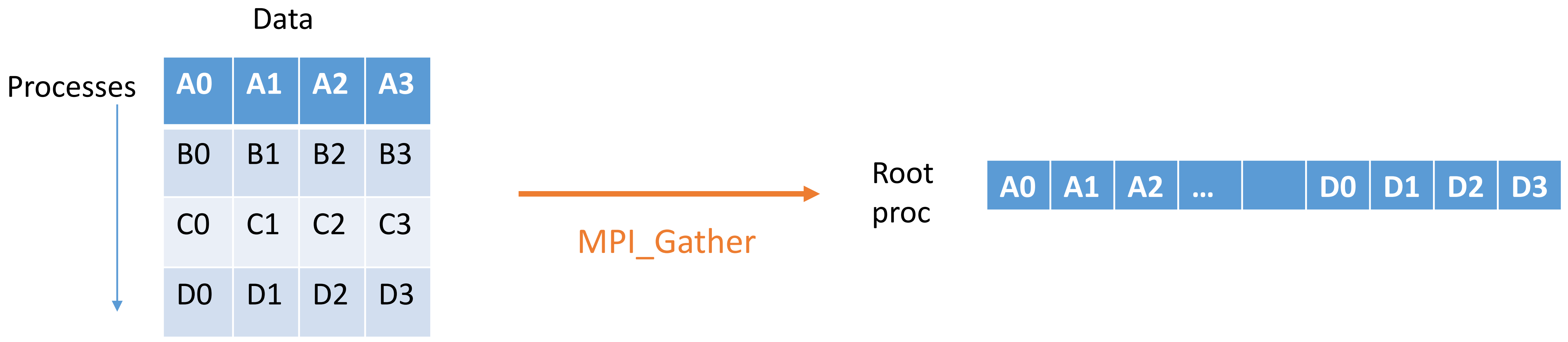
Example

The reverse of Example using MPI_GATHER . Scatter sets of 100 ints from the root to each process in the group.



```
MPI_Comm comm;
int NrProcs,*sendbuf;
int root, rbuf[100];
...

MPI_Comm_size( comm, &NrProcs);
sendbuf = (int *)malloc(NrProcs*100*sizeof(int));
 ...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```


Data
Root proc
A0 A1 A2 ... D0 D1 D2 D3
MPI_Scatter
Processes
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3

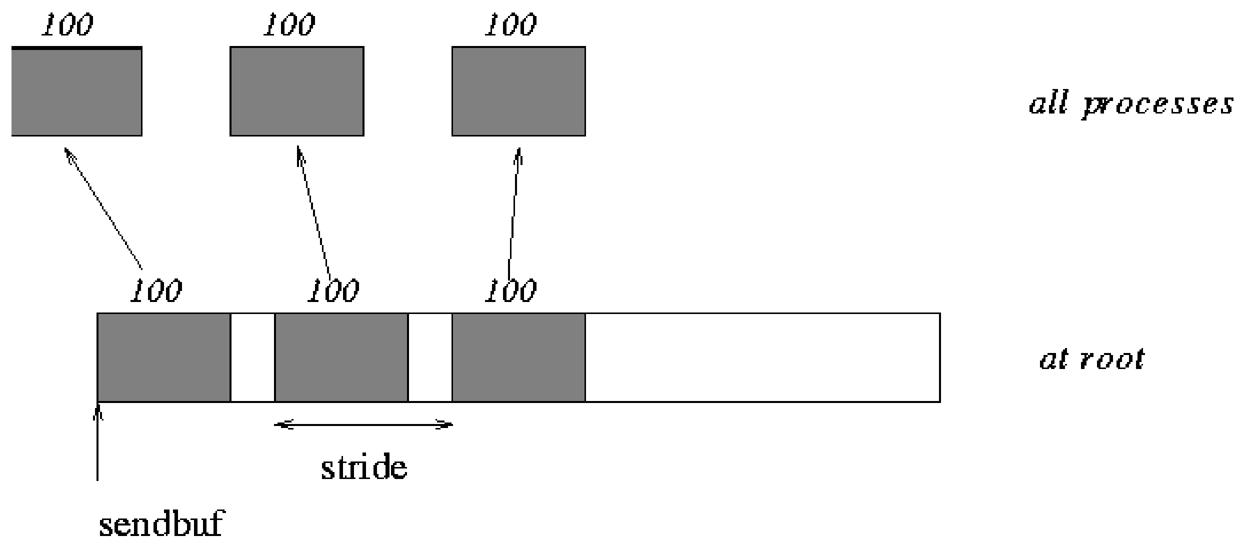

Data
Processes
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3
MPI_Gather
Root proc
A0 A1 A2 ... D0 D1 D2 D3

MPI_Scatterv

Example

The reverse of Example using MPI_GATHERV . The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride* ints apart in the sending buffer. Assume *stride*≥100 .

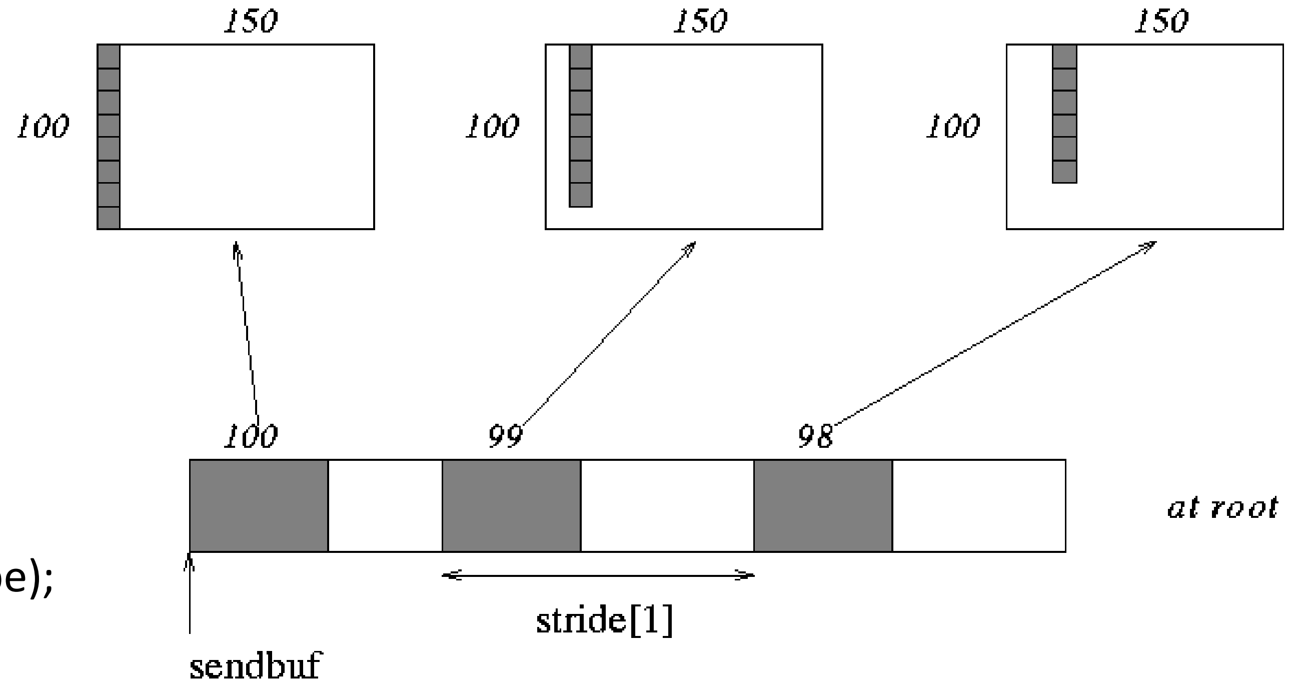
```
MPI_Comm comm;
int NrProcs,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size( comm, &NrProcs);
sendbuf = (int *)malloc(NrProcs*stride*sizeof(int));
...
displs = (int *)malloc(NrProcs*sizeof(int));
scounts = (int *)malloc(NrProcs*sizeof(int));
for (i=0; i<NrProcs; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



MPI_Scatterv

Example
The root scatters blocks of 100-i ints into column i of a 100x150 C array. At the sending side, the blocks are stride[i] ints apart.

```
MPI_Comm comm;
int NrProcs,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_rank( comm, &myrank );
stride = (int *)malloc(NrProcs*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow * sendbuf comes from elsewhere */
...
displs = (int *)malloc(NrProcs*sizeof(int));
scounts = (int *)malloc(NrProcs*sizeof(int));
offset = 0;
for (i=0; i<NrProcs; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype, root, comm);
```

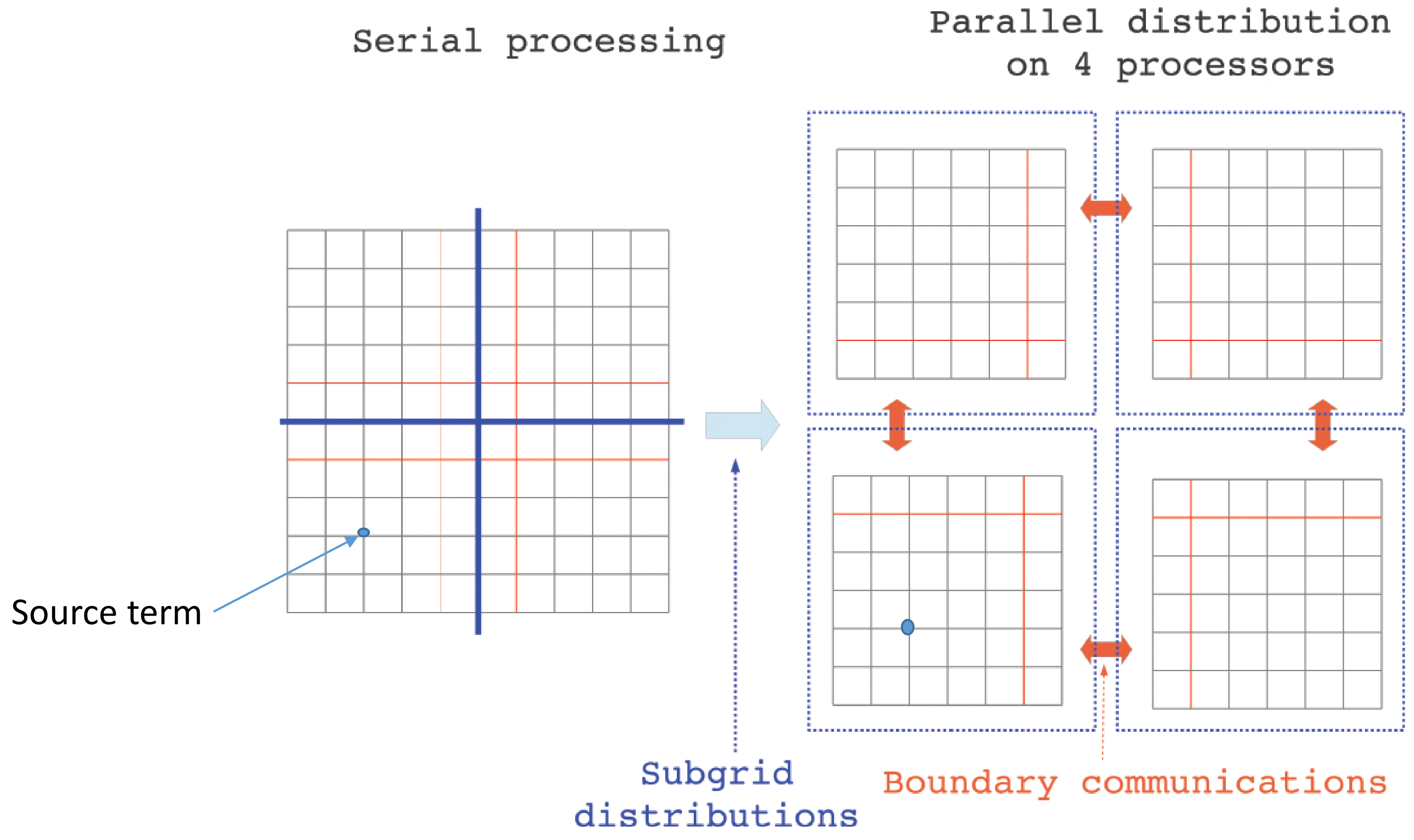


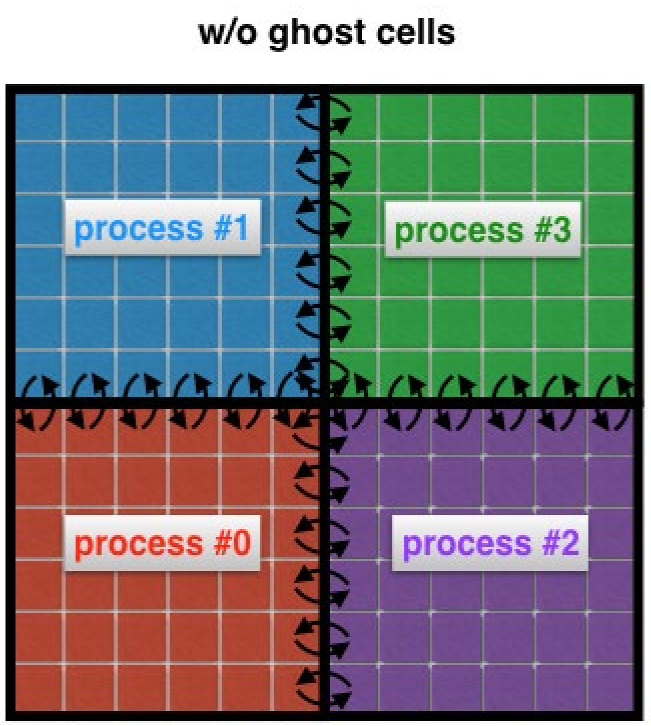
Exercise 1:
Poisson Solver

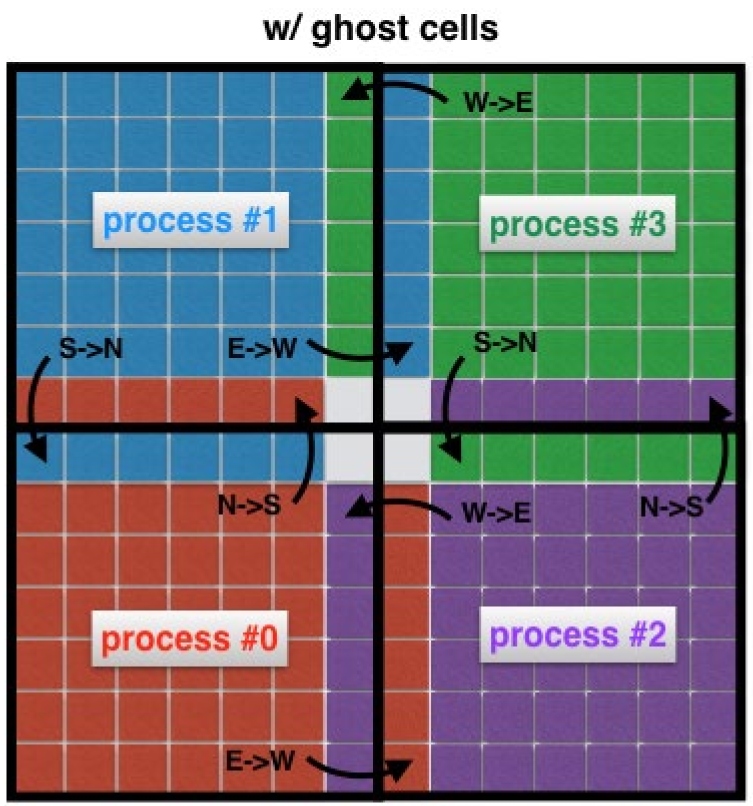
Ghost cells and HALO-exchange

Process topology (2-D mesh)

Discretization grid and partitioning





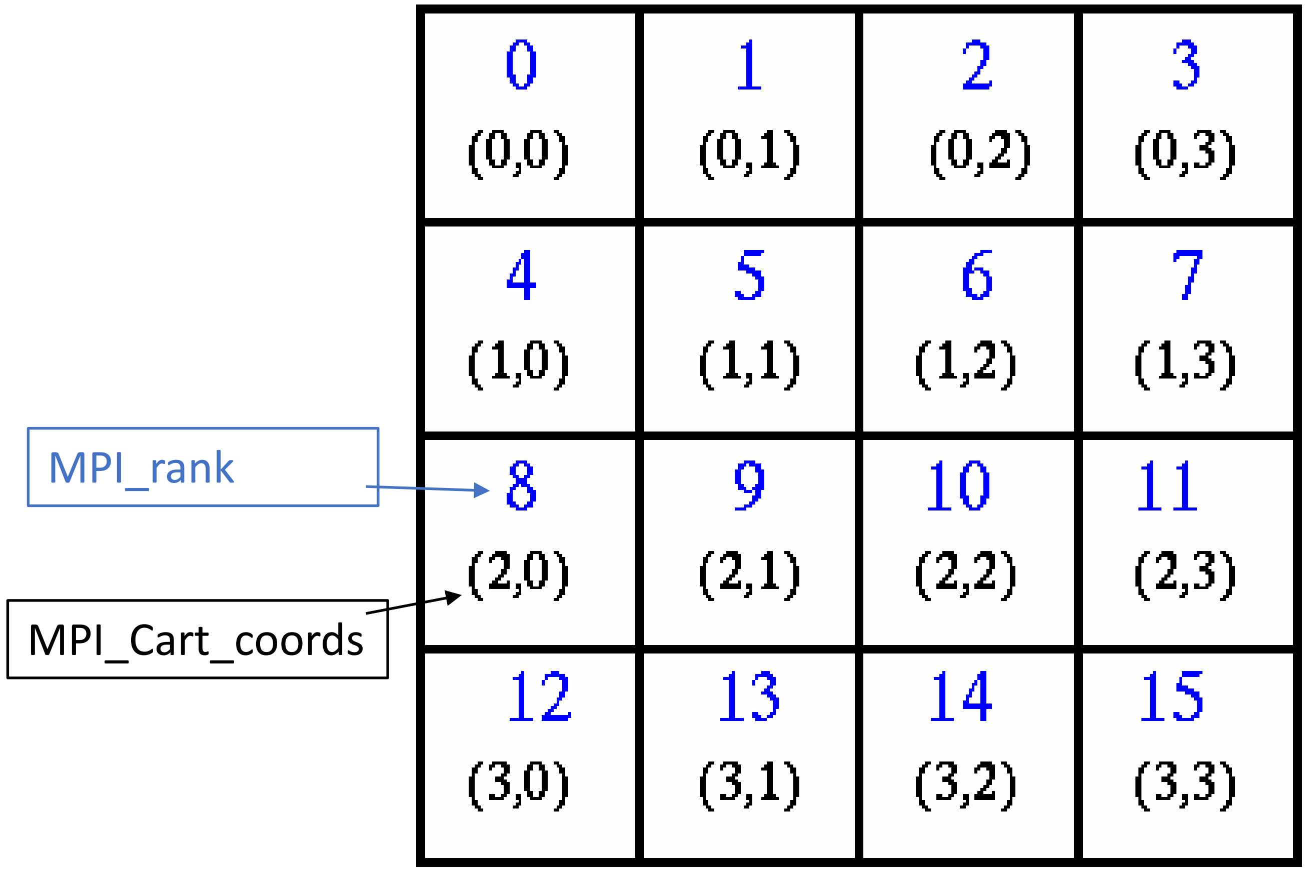


Suppose the size of a subgrid is $n_{row} \times n_{col}$, the array size increases with ghost cells to $(n_{row}+2)\times(n_{col}+2)$.

Row 0 and $n_{row}+1$ stores the values received from the top and bottom neighbour respectively.
Similary, for column 0 and $n_{col}+1$ …
(All local data start from (0,0)!)

Communication among processes without (left) and with (right) ghost cells. Without ghost cells, each cell on the boundary of a sub-domain needs to pass its own message to a neighboring process. Using ghost cells allows to minimize the number of messages passed, as many cells belonging to the boundaries of a process are exchanged at once with a single message. Here, for example, Process #0 is passing its entire North boundary to Process #1, and its entire East boundary to Process #2.

2-D Cartesian process topology (for easy neighbor communication)



Creating a virtual topology: comm_2d

– MPI_COMM_WORLD/oldcomm communicator

– **ndims** dimension of the Cartesian topology

– **dims** integer array (size ndims) that defines the number of processes in each dimension

– **periods** array that defines the periodicity of each dimension – reorder is MPI allowed to renumber the ranks

– **comm_2d** new Cartesian communicator

```
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_2d)
```
Here, ndims=2, dims[0]=4, dims[1]=4, periods[0]=periods[1]=0,reorder=0

```
MPI_Comm_rank(comm_2d, my_rank, ierror)
MPI_Cart_coords(comm_2d, my_rank, maxdims, my_coords, ierror)
```

Cartesian neighbours

Counting ”hops” in the Cartesian grid to allow for e.g., elegant nearest-neighbor communication

MPI_Cart_shift(comm, direction, displ, source, dest)

– comm Cartesian communicator
– direction shift direction (e.g., 0 or 1 in 2D)
– displ shift displacement (1 for next cell etc., <0 for “down”/”left” directions)
– source rank of source process
– dest rank of destination process

With non-periodic grid, *source* or *dest* can land outside of the grid; then MPI_PROC_NULL (or sometimes a negative rank number) is returned

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

invisible input argument: my_rank of the running process executing:
MPI_Cart_shift(comm_2d, direction, displ, &prev, &next)

example on process rank=6

direction	displ	&prev	&next
0	+1	5	7
1	+1	2	10
0	-1	7	5