Introduction to Parallel Computing

Vistas in Advanced Computing / Summer 2017

Collective operation

- All process of a process group have to participate in the same operation
  - process group is defined by a communicator
  - all processes have to provide the same arguments
  - for each communicator, you can have one collective operation ongoing at a time
- Collective operations are abstractions for often occurring communication patterns
  - eases programming
  - enables low-level optimizations and adaptations to the hardware infrastructure

MPI collective operations

MPI_Barrier
MPI_Bcast
MPI_Scatter
MPI_Scatterv
MPI_Gather
MPI_Gatherv
MPI_Allgather
MPI_Allgatherv
MPI_Alltoall
MPI_Alltoallv
MPI_Reduce
MPI_Allreduce
MPI_Reduce_scatter
MPI_Scan

MPI_Exscan
MPI_Alltoallw

More MPI collective operations

- Creating and freeing a communicator is considered a collective operation
  - e.g. `MPI_Comm_create`
  - e.g. `MPI_Comm_spawn`
- Collective I/O operations
  - e.g. `MPI_File_write_all`
- Window synchronization calls are collective operations
  - e.g. `MPI_Win_fence`

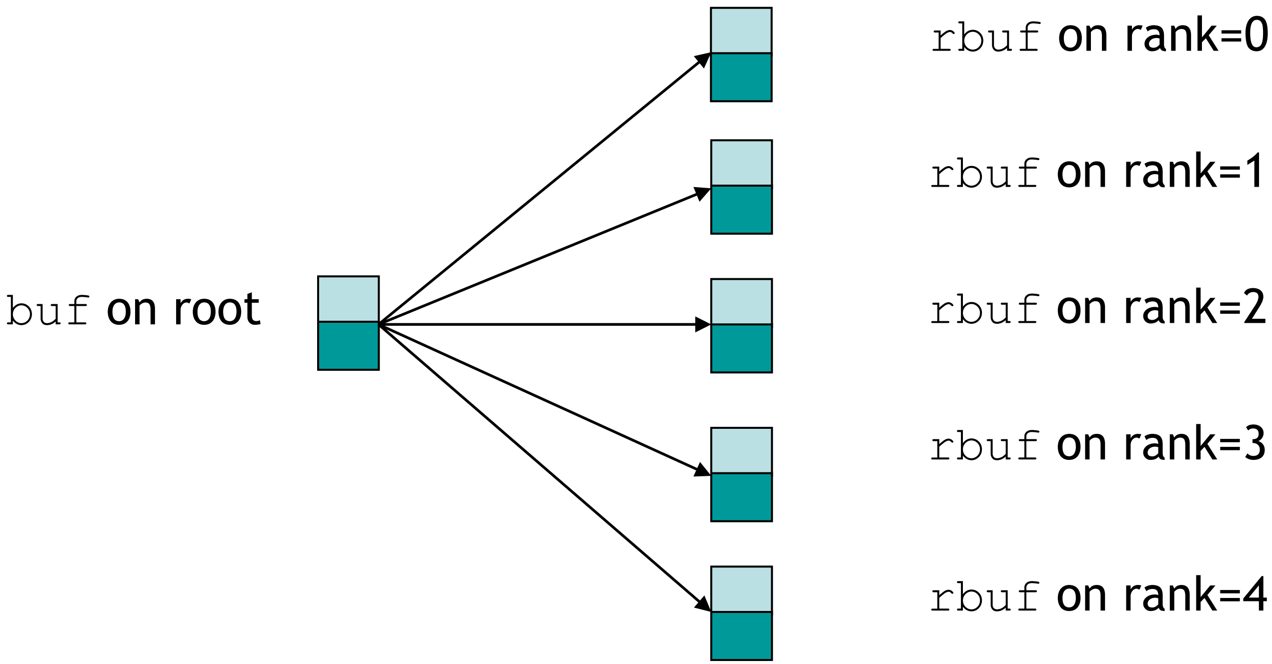
MPI_Bcast

```
MPI_Bcast (void *buf, int cnt, MPI_Datatype dat,
              int root, MPI_Comm comm);
```

- The process with the rank `root` distributes the data stored in `buf` to all other processes in the communicator `comm`.
- Data in `buf` is identical on all processes after the bcast
- Compared to point-to-point operations no `tag`, since you cannot have several ongoing collective operations

MPI_Bcast (II)

```
MPI_Bcast (buf, 2, MPI_INT, 0, comm);
```



UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Example: distributing global parameters

```
int rank, problemsize;
float precision;
MPI_Comm comm=MPI_COMM_WORLD;

MPI_Comm_rank ( comm, &rank );
if (rank == 0 ) {
  FILE *myfile;
  myfile = fopen("testfile.txt", "r");
  fscanf (myfile, "%d", &problemsize);
  fscanf (myfile, "%f", &precision);
  fclose (myfile);
}

MPI_Bcast (&problemsize, 1, MPI_INT, 0, comm);
MPI_Bcast (&precision, 1, MPI_FLOAT, 0, comm);
```

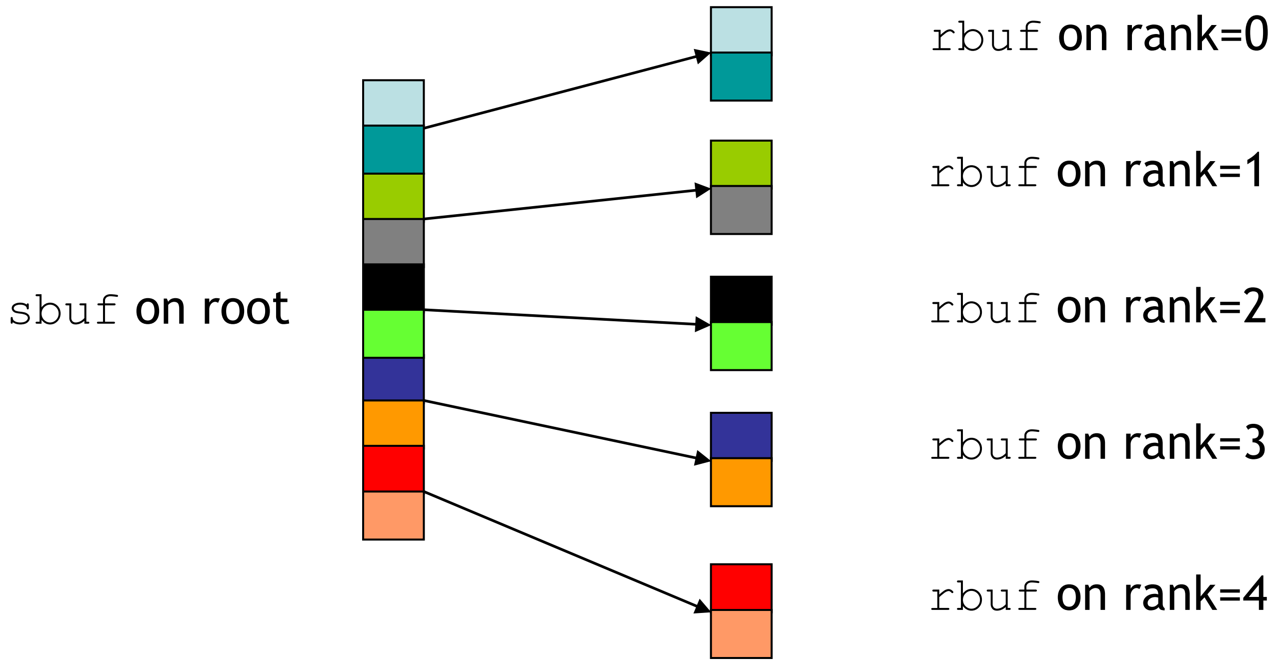
MPI_Scatter

```
MPI_Scatter (void *sbuf, int scnt, MPI_Datatype sdat,
             void *rbuf, int rcnt, MPI_Datatype rdat,
             int root, MPI_Comm comm);
```

- The process with the rank `root` distributes the data stored in `sbuf` to all other processes in the communicator `comm`
- Difference to Broadcast: every process gets different segment of the original data at the root process
- Arguments `sbuf, scnt, sdat` only relevant and have to be set at the root-process

MPI_Scatter (II)

```
MPI_Scatter (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```



Example: partition a vector among processes

```
int rank, size;
float *sbuf, rbuf[3] ;
MPI_Comm comm=MPI_COMM_WORLD;

MPI_Comm_rank ( comm, &rank );
MPI_Comm_size ( comm, &size );

if (rank == root ) {
      sbuf = malloc (3*size*sizeof(float);
      /* set sbuf to required values etc. */
}

/* distribute the vector, 3 Elements for each process
*/
MPI_Scatter (sbuf, 3, MPI_FLOAT, rbuf, 3, MPI_FLOAT,
             root, comm);
if ( rank == root ) {
      free (sbuf);
}
```

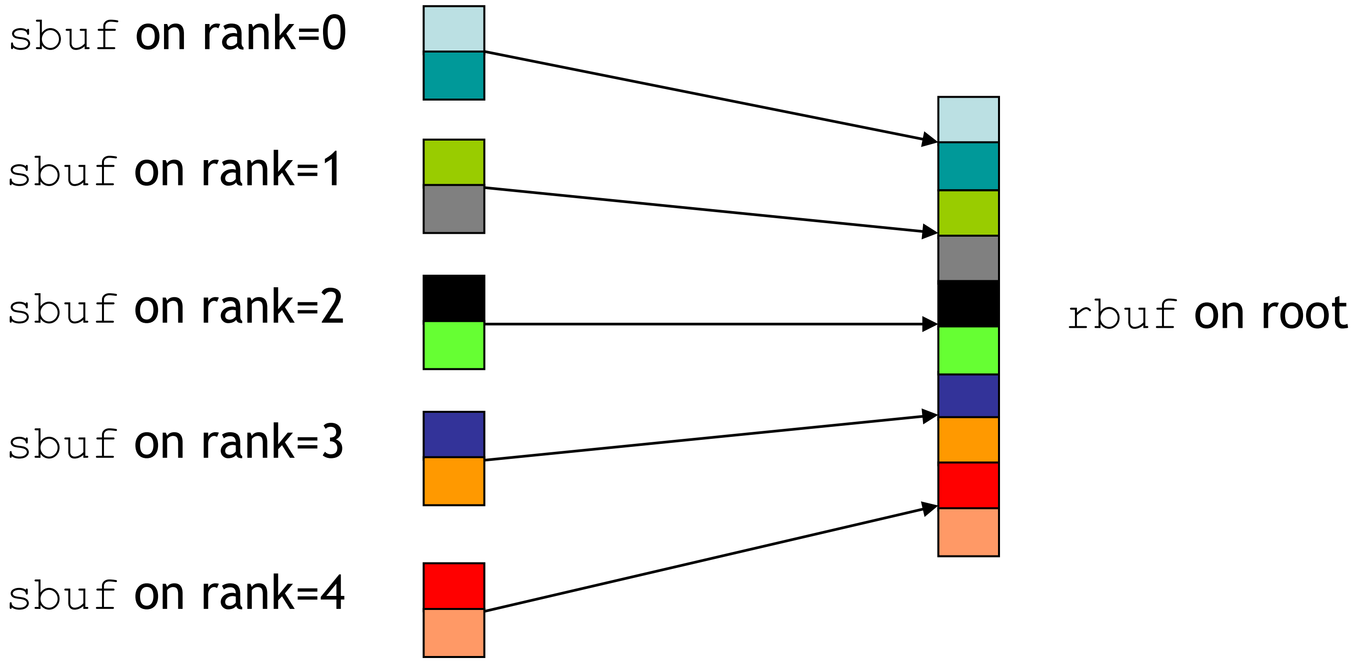
MPI_Gather

```
MPI_Gather (void *sbuf, int scnt, MPI_Datatype sdat,
            void *rbuf, int rcnt, MPI_Datatype rdat,
            int root, MPI_Comm comm);
```

- Reverse operation of `MPI_Scatter`
- The process with the rank `root` receives the data stored in `sbuf` on all other processes in the communicator `comm` into the `rbuf`
- Arguments `rbuf, rcnt, rdat` only relevant and have to be set at the root-process

MPI_Gather (II)

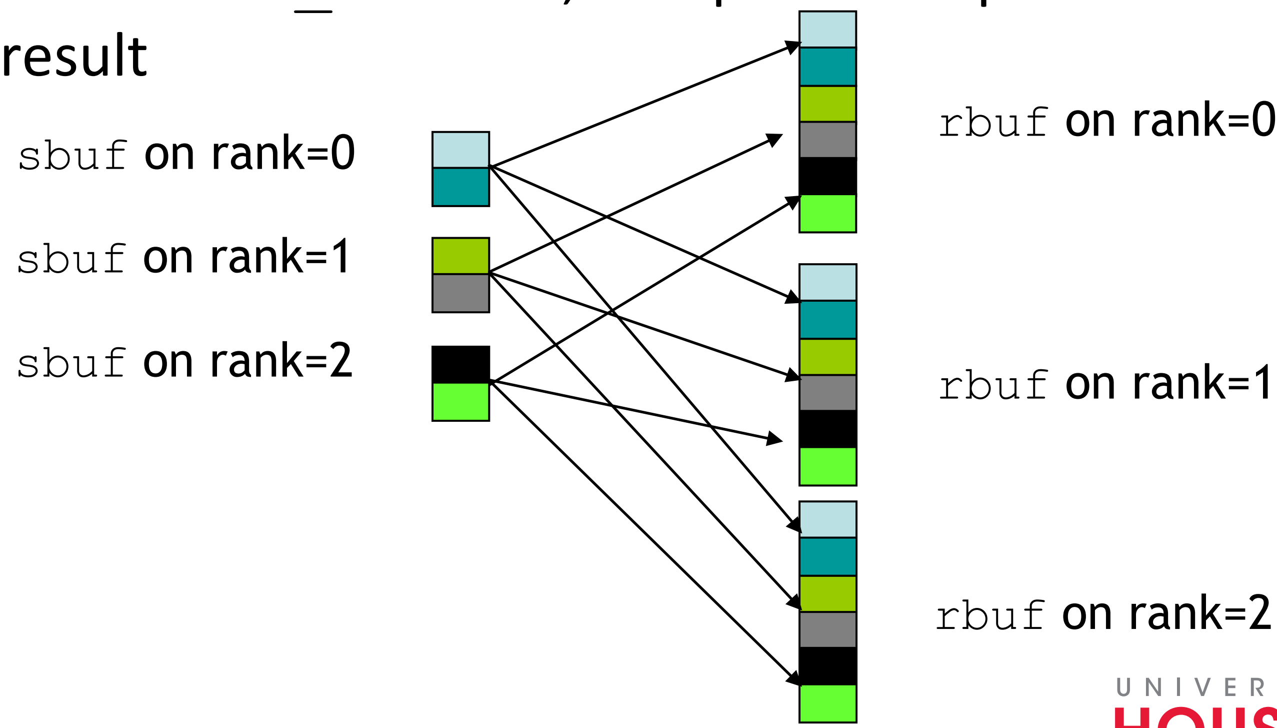
```
MPI_Gather (sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 0, comm);
```



MPI_Allgather

```
MPI_Allgather (void *sbuf, int scnt, MPI_Datatype sdat,
               void *rbuf, int rcnt, MPI_Datatype rdat,
               MPI_Comm comm);
```

- Identical to `MPI_Gather`, except that all processes have the final result



UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

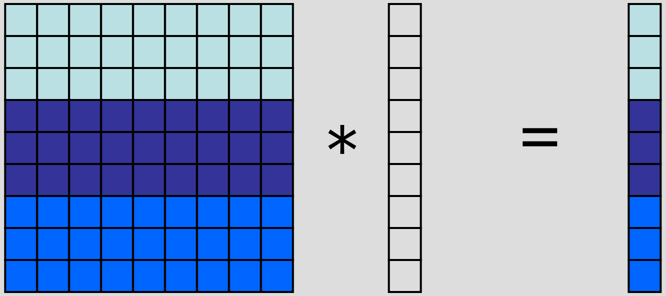
Example: matrix-vector multiplication with row-wise block distribution

```
int main( int argc, char **argv)
{
  double A[nlocal][n], b[n];
  double c[nlocal], cglobal[n];
  int i,j;

  …

  for (i=0; i<nlocal; i++) {
    for ( j=0;j<n; j++ ) {
     c[i] = c[i] + A(i,j)*b(j);
    }
  }

  MPI_Allgather( c, nlocal, MPI_DOUBLE, cglobal, nlocal,
                       MPI_DOUBLE, MPI_COMM_WORLD );
```



Each process holds the final result for its part of c

Reduction operations

```
MPI_Reduce (void *inbuf, void *outbuf, int cnt,
            MPI_Datatype dat, MPI_Op op, int root,
            MPI_Comm comm);
MPI_Allreduce (void *inbuf, void *outbuf, int cnt,
               MPI_Datatype dat, MPI_Op op,
               MPI_Comm comm);
```

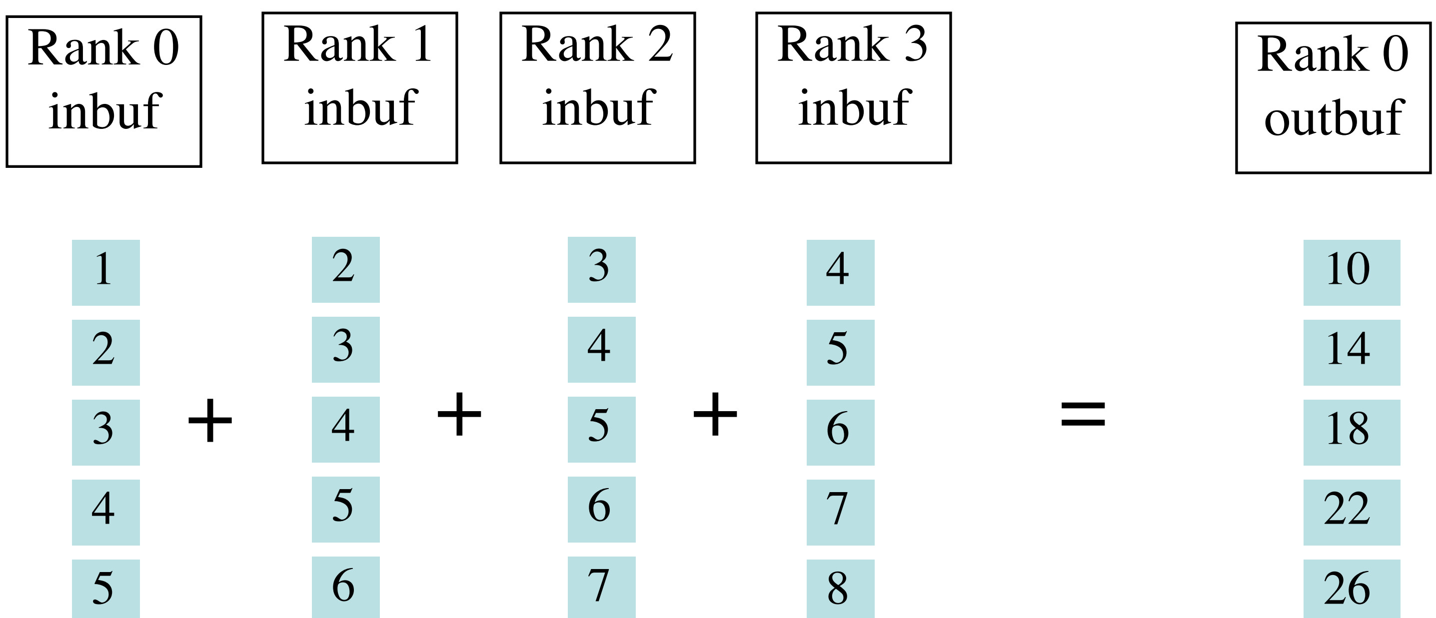
- Perform simple calculations (e.g. caculate the sum or the product) over all processes in the communicator
- `MPI_Reduce`
  - `outbuf` has to be provided by all processes
  - result is only available at `root`
- `MPI_Allreduce`
  - result available on all processes

Predefined reduction operations

`MPI_SUM`	sum
`MPI_PROD`	product
`MPI_MIN`	minimum
`MPI_MAX`	maximum
`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 value and location
`MPI_MINLOC`	minimum value and location

Reduction operations on vectors

- Reduce operation is executed element wise on each entry of the array



- Reduction of 5 elements with root = 0

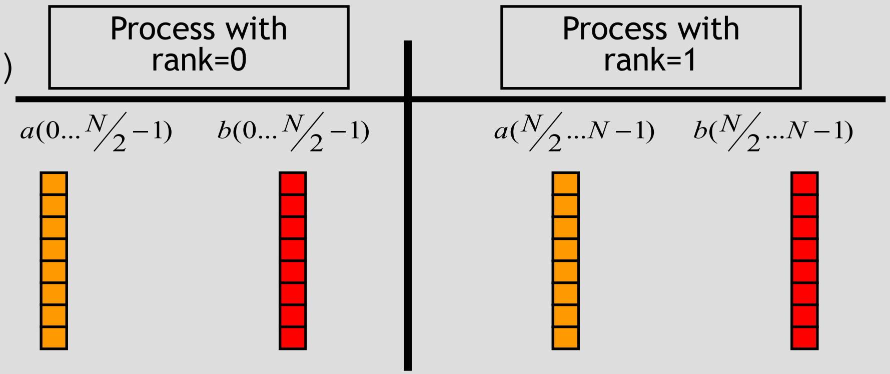
```
MPI_Reduce (inbuf, outbuf, 5, MPI_INT, MPI_SUM, 0,
            MPI_COMM_WORLD);
```

Example: scalar product of two vectors

```
int main( int argc, char **argv)
{
  int i, rank, size;
  double a_local[N/2];
  double b_local[N/2];
  double s_local, s;

  …
  s_local = 0;
  for ( i=0; i<N/2; i++ ) {
    s_local = s_local + a_local[i] * b_local[i];
  }

  MPI_Allreduce ( &s_local, &s, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD );
  …
```

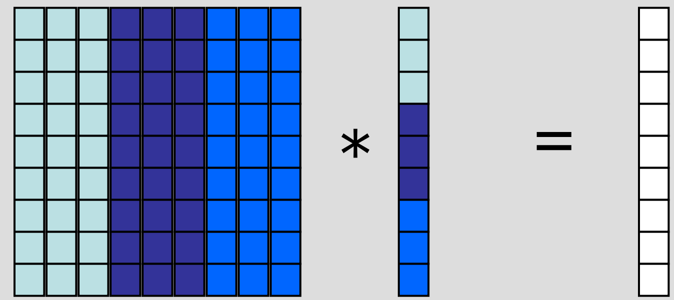


Example: matrix-vector multiplication with column-wise block distribution

```
int main( int argc, char **argv)
{
  double A[n][nlocal], b[nlocal];
  double c[n], ct[n];
  int i,j;

  …

  for (i=0; i<n; i++) {
    for ( j=0;j<nlocal;j++ ) {
     ct[i] = ct[i] + A(i,j)*b(j);
    }
  }
  MPI_Allreduce ( ct, c, n, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD );
```



Result of local computation in temporary buffer

MPI_Barrier

```
MPI_Barrier (MPI_Comm comm);
```

- Synchronizes all processes of the communicator
  - no process can continue with the execution of the application until all process of the communicator have reached this function
  - often used before timing certain sections of the application
- MPI makes no statement about the quality of the synchronization
- Advice: no scenario is known to me, which requires a barrier for <u>correctness</u>. Usage of `MPI_Barrier` strongly discouraged.