CME213/ME339
Lecture 19

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

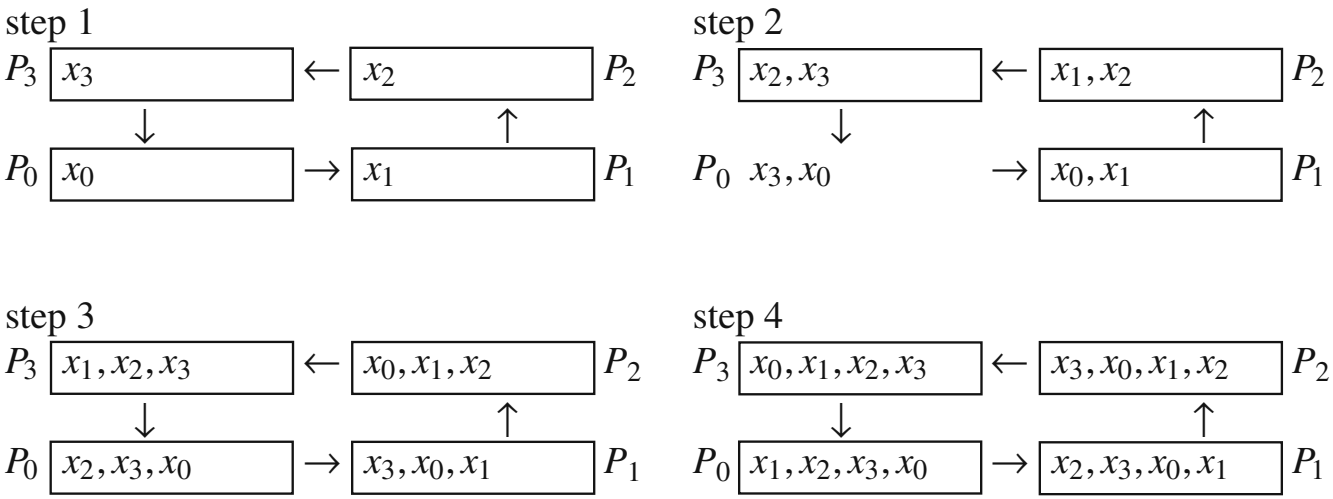
Spring 2012

Example

Non-blocking gather ring.

Assume we have a ring of processes. Each process has some data x_i. We want all processes to have all the data.

At each step, a process receives from its left and sends to its right the data it received at the previous step.



Code

```
1  void Gather_ring(float * x, int blocksize, float * y) {
2    int i, p, my_rank, next, prev;
3    int send_offset, recv_offset;
4    MPI_Status status;
5    MPI_Request send_request, recv_request;
6
7    MPI_Comm_size(MPI_COMM_WORLD, &p);
8    // Number of processes
9    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10   // My process ID or rank
11   ...
```

Code

```
1   ...
2   for (i=0; i<blocksize; i++)
3     y[i+my_rank*blocksize] = x[i];
4   /* Copying data that was produced or is owned
5      by process my_rank */
6
7   next = (my_rank+1) % p;    // next process in ring
8   prev = (my_rank-1+p) % p; // previous process
9   send_offset = my_rank * blocksize;
10  recv_offset = ((my_rank-1+p) % p) * blocksize;
11  ...
```

Code

```
1    for (i=0; i<p-1; i++) {
2      // Sending and receiving the different x blocks
3      MPI_Isend(y+send_offset, blocksize, MPI_FLOAT, next,
4              0, MPI_COMM_WORLD, &send_request);
5      MPI_Irecv(y+recv_offset, blocksize, MPI_FLOAT, prev,
6              0, MPI_COMM_WORLD, &recv_request);
7      send_offset = ((my_rank-i-1+p) % p) * blocksize;
8      recv_offset = ((my_rank-i-2+p) % p) * blocksize;
9      MPI_Wait(&send_request, &status);
10     MPI_Wait(&recv_request, &status);
11   }
12 }
```

Communication modes

This is a more advanced topic. We cover the only the key concepts.

Standard: this the mode we have used up to now. This is in most cases sufficient. It provided good performance and relies on the MPI library for several optimizations. For example, MPI will decide whether or not buffers should be used.

Synchronous mode

In synchronous mode, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent.

This leads to a form of synchronization between the sending and the receiving processes: the completion of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer.

Note: completion does not imply that the receiving node has finished receiving the data.

Blocking: returns when completed (sending buffer can be reused + receiving process started executing the matching receive).

Non-blocking: an `MPI_Wait` is required to check completion.

See documentation for more info. Commands include for example `MPI_Ssend()`.

Two other modes

Buffered mode: the user can allocated space for the MPI system buffer. This guarantees that a buffer is used. This is therefore a **guaranteed local operation.** It never depends on the receiving node. `MPI_Bsend()`

Ready mode: it can only be used if the user can guarantee that a matching receive has already been posted. The user is responsible for writing a correct program. Ready mode aims to minimize system overhead and synchronization overhead incurred by the sending task. `MPI_Rsend()`

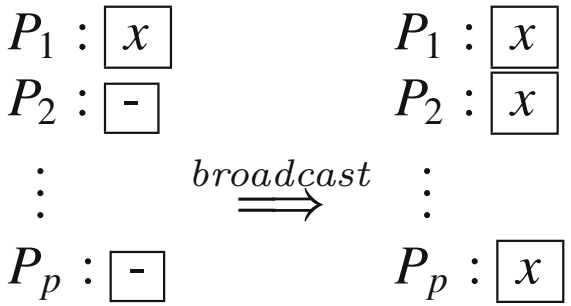
Collective communications

The most important communication type after point-to-point.

Relates to all communication involving many processors.

A few generic operations are provided. This is typically sufficient for most applications.

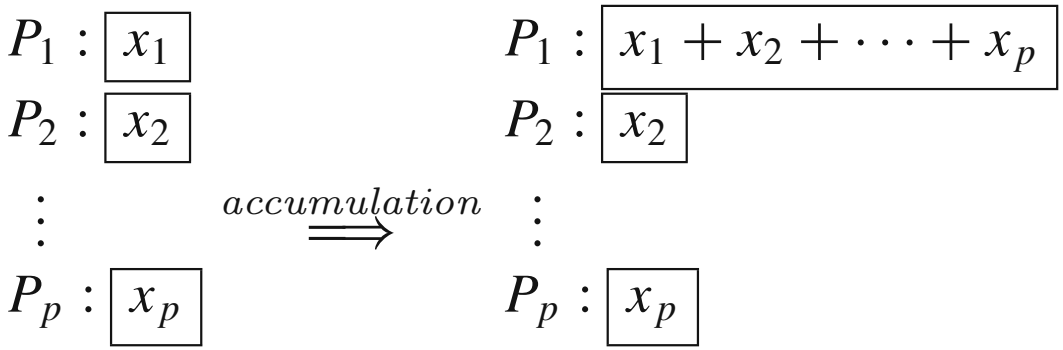
Broadcast




```
1  int MPI_Bcast(void *message, int count,
2                MPI Datatype type, int root,
3                MPI Comm comm)
```

Collective MPI communication operations are always blocking.

Reduce




```
1  int MPI_Reduce(void *sendbuf, void *recvbuf,
2                 int count,
3                 MPI Datatype type,
4                 int root,
5                 MPI Comm comm)
```

The accumulated result is collected at a root process.

The parameter `recvbuf` specifies the receive buffer which is provided by the root process `root`.

Reduction operations supported

Representation	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bit-wise and
MPI_LOR	Logical or
MPI_BOR	Bit-wise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bit-wise exclusive or
MPI_MAXLOC	Maximum value and corresponding index
MPI_MINLOC	Minimum value and corresponding index

Aplications: pivot element in LU, norm of residual in iterative methods, dot product, ...

Example: calculating π

```
1  #include "mpi.h"
2
3  double func(double x) { return (4.0 / (1.0 + x*x)); }
4
5  int main(int argc,char *argv[]) {
6      int    NoInterval, interval;
7      int    myrank, nprocs, root = 0;
8      double mypi, pi, h, sum, x;
9      double PI25DT = 3.14159265358979323846264643;
```

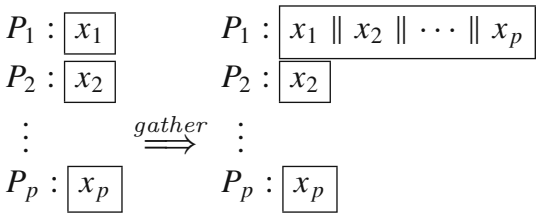
```
1     /* MPI initialisation */
2     MPI_Init(&argc,&argv);
3     MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
4     MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
5
6     if(myrank == root){
7         printf("Enter the number of intervals: ");
8         fflush(stdout);
9         scanf("%d",&NoInterval);
10    }
```

```
    /* Broadcast the number of subintervals to each processor */
    MPI_Bcast(&NoInterval, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double)NoInterval; sum = 0.0;
    for (interval = myrank + 1; interval <= NoInterval;
         interval += nprocs) {
        x = h * ((double) interval - 0.5);
        sum += func(x);
    }
    mypi = h * sum;

    /* Collect all the partial results */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, root,
               MPI_COMM_WORLD);

    if (myrank == root)
        printf("pi is approximately %.16f, Error is %.8e\n",
               pi, fabs(pi - PI25DT));

    MPI_Finalize();
    exit(0);
}
```

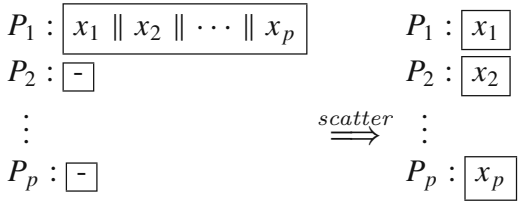
Output

```
$ mpirun -np 8 ./pi
Enter the number of intervals: 100000
pi is approximately 3.1415926535981349, Error is 8.34177172e-12
```

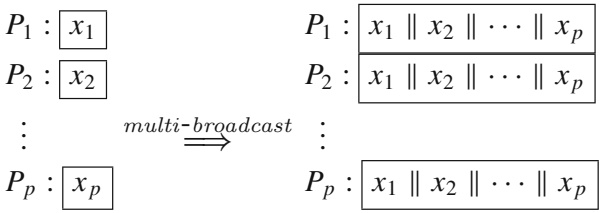
Many other collective operations



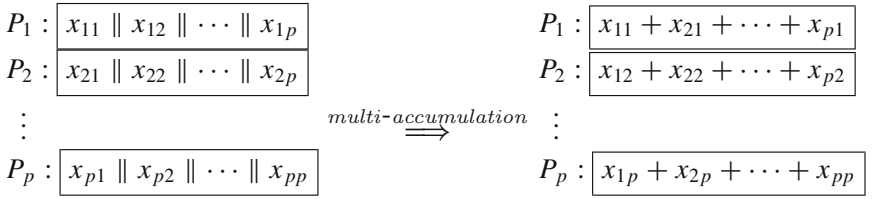
`MPI_Gather()`



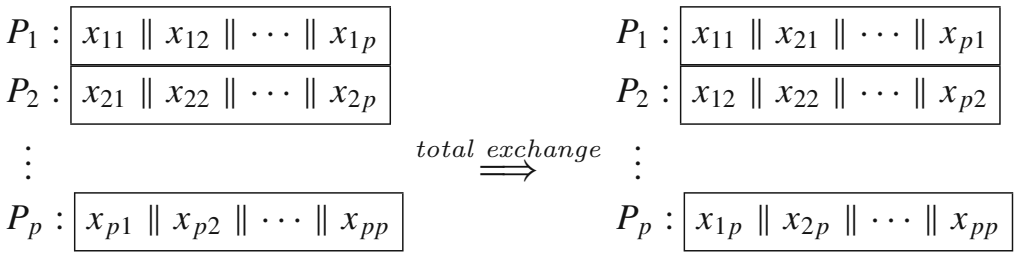
`MPI_Scatter()`



`MPI_Allgather()`



`MPI_Allreduce()`



`MPI_Alltoall()`

Other common collective operations:

`MPI_Barrier()`

`MPI_Scan()`

Example: row-wise matrix-vector product

```
1  RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                          MPI_Comm comm) {
3  /* a stores the rows of the matrix that this process
4     is going to compute with.
5     b stores only the part of b owned by this process */
6  int i, j;
7  int nlocal; /* Number of locally stored rows of A */
8  double *fb; /* Will store the entire vector b */
9  int nprocs, myrank;
10 MPI_Status status;
11
12 /* Get information about the communicator */
13 MPI_Comm_size(comm, &nprocs);
14 MPI_Comm_rank(comm, &myrank);
15
16 /* Allocate the memory used to store the entire b */
17 fb = (double *)malloc(n*sizeof(double));
18 nlocal = n/nprocs; // nprocs must divide n
```

```
1  /* Gather entire vector b on each processor using Allgather */
2  MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
3                comm);
4
5  /* Perform the matrix-vector multiplication involving the
6     locally stored submatrix. */
7  for (i=0; i<nlocal; i++) {
8    x[i] = 0.0;
9    for (j=0; j<n; j++)
10     x[i] += a[i*n+j]*fb[j];
11 }
12 /* Done! */
13 free(fb);
14 }
```

Example: column-wise matrix-vector product

```
1  ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                          MPI_Comm comm) {
3  int i, j;
4  int nlocal;
5  double *px;
6  double *fx;
7  int nprocs, myrank;
8  MPI_Status status;
9  /* Get identity and size information from the communicator */
10 MPI_Comm_size(comm, &nprocs);
11 MPI_Comm_rank(comm, &myrank);
12
13 nlocal = n/nprocs;
14
15 /* Allocate memory for arrays storing intermediate results. */
16 px = (double *)malloc(n*sizeof(double));
17 fx = (double *)malloc(n*sizeof(double));
```

```
/* Compute the partial-dot products that correspond to the
   local columns of A.*/
for (i=0; i<n; i++) {
  px[i] = 0.0;
  for (j=0; j<nlocal; j++)
    px[i] += a[i*nlocal+j]*b[j];
}

/* Sum-up the results by performing an element-wise
   reduction operation. Result is stored at 0. */
MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

/* Redistribute fx in a fashion similar to that of vector b.
   Data is sent from 0 to all other processes. */
MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
            comm);
free(px); free(fx);
}
```