Collective Communications II

Ned Nedialkov

McMaster University
Canada

SE/CS 4F03
January 2014

Outline

Scatter

Sends data from one process to all other processes in a communicator

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

`sendbuf`	starting address of send buffer (significant only at root)
`sendcount`	number of elements sent to each process (significant only at root)
`sendtype`	data type of `sendbuf` elements (significant only at root)
`recvbuf`	address of receive buffer
`recvcount`	number of elements for any single receive
`recvtype`	data type of `recvbuf` elements
`root`	rank of sending process
`comm`	communicator

- Assume p processes
- `MPI_Scatter` splits data at `sendbuf` on root into p segments
- each of `sendcount` elements, and
- sends these segments to processes $0, 1, \ldots, p-1$ in order

© 2013–14 Ned Nedialkov

```c
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *
    displs, MPI_Datatype sendtype, void *recvbuf, int
    recvcount, MPI_Datatype recvtype, int root, MPI_Comm
    comm)
```

`sendcounts`	integer array (of size *p*) specifying the number of elements to send to each process
`displs`	integer array (of size *p*) Entry `i` specifies the displacement (relative to `sendbuf`) from which to take the outgoing data to process `i`
`recvbuf`	address of receive buffer (output)

For an illustration, see http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html

© 2013–14 Ned Nedialkov

Example: parallel matrix times vector

- We want to compute the product

$$y = Ab,$$

  where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$ on a distributed memory machine
- Given p processes
  - distribute the work equally
  - each process multiplies
  - process 0 collects the result y
- How to distribute the work?

In this example, each process

- stores b
- stores m/p rows of A
- multiples (m/p rows of A)$\times b$

What if p does not divide m?

- Process i works on r_i rows, where

$$r_i = m \div p + \begin{cases} 1 & \text{if } i < m \mod p \\ 0 & \text{otherwise} \end{cases}$$

  - $\div$ is integer division
  - $\mod$ is remainder
- In C

```
#define NUM_ROWS(i,p,m) ( m/p + ( (i<m%p) ?1:0) )
```

Example: distributing a matrix

```
#define NUM_ROWS(i,p,m) ( m/p + ( (i<m%p) ?1:0) )

 /* Distribute matrix A of size num_rows x num_cols to
    p processes. A is stored as one-dimensional array of size
    num_rows*num_cols.
    B is of size local_num_rows*num_cols
    local_num_rows = NUM_ROWS(my_rank, p, num_rows); */
  int *displs, *sendcounts;
  if (my_rank == 0)
    {
      displs = malloc(sizeof(int)*p);
      sendcounts = malloc(sizeof(int)*p);
      sendcounts[0] = NUM_ROWS(0,p,num_rows)*num_cols;
      displs[0] = 0;
      for (i=1; i<p; i++)
        {
          displs[i] = displs[i-1] + sendcounts[i-1];
          sendcounts[i] = NUM_ROWS(i,p,num_rows)*num_cols;
        }
    }
  MPI_Scatterv(A, sendcounts, displs, MPI_DOUBLE, B,
      local_num_rows*num_cols, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Gather

Gathers together data from a group of processes

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

`sendbuf`	starting address of send buffer
`sendcount`	number of elements in send buffer
`sendtype`	data type of `sendbuf` elements
`recvbuf`	address of receive buffer (significant only at root)
`recvcount`	number of elements for any single receive (significant only at root)
`recvtype`	data type of `recvbuf` elements (significant only at root)
`root`	root rank of receiving process
`comm`	communicator

- `MPI_Gather` collects data, stored at `sendbuf`, from each process in `comm` and stores the data on `root` at `recvbuf`
- Data is received from processes in order, i.e. from process 0, then from process 1 and so on
- Usually `sendcount`, `sendtype` are the same as `recvcount`, `recvtype`
- `root` and `comm` must be the same on all processes
- The receive parameters are significant only on root
- Amount of data sent/received must be the same

MPI_Allgather

```
int MPI_Allgather(void *sendbuf, int sendcount,
   MPI_Datatype sendtype,void *recvbuf, int recvcount,
   MPI_Datatype recvtype, MPI_Comm comm)
```

- The block of data sent from the *i*th process is received by every process and placed in the *i*th block of the buffer `recvbuf`

```
int MPI_Allgatherv(void *sendbuf, int sendcount,
   MPI_Datatype sendtype,void *recvbuf, int *recvcounts,
    int *displs, MPI_Datatype recvtype, MPI_Comm comm)
```

- The “opposite” of `MPI_Scatterv`

Serial $A \times b$

```c
/* dotproduct.c
compMatrixTimesVector computes the result of matrix times vector.
The input matrix A is of size m x n, and it is stored as a
one-dimensional array. The result is stored at y, which contains the
computed y = A*b.
Copyright 2014 Ned Nedialkov
*/
void compMatrixTimesVector(int m, int n, const double *A,
                           const double *b, double *y)
{
  int i, j;
  for (i = 0; i < m; i++)
    {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
        {
          const double *pA = A + i * n;
          y[i] += *(pA + j) * b[j];
        }
    }
}
```

Parallel $A \times b$

```
/*  parallelmatvec.c

    parallelMatrixTimesVector performs parallel matrix-vector
    multiplication of a matrix A times vector b.  The matrix is
    distributed by rows. Each process contains a
    (num_local_rows)x(cols) matrix local_A stored as a
    one-dimensional array.
    The vector b is stored on each process.
    Each process computes its result and then
    process root collects the results and returns it in y.

    num_local_rows number of rows on my_rank
    cols           number of columns on each process
    local_A        pointer to the matrix on my_rank
    b              pointer to the vector b of size cols
    y              pointer to the result on the root process.
                   y is significant only on root.

    Copyright 2014 Ned Nedialkov
  */
```

Parallel $A \times b$

```c
#include <mpi.h>
#include <stdlib.h>

void compMatrixTimesVector(int m, int n, const double *A,
                           const double *b, double *y);

void parallelMatrixTimesVector(int num_local_rows, int cols,
                               double *local_A, double *b, double *y,
                               int root, int my_rank, int p,
                               MPI_Comm comm)
{
  /* Allocate memory for the local result on my_rank */
  double *local_y = malloc(sizeof(double)*num_local_rows);

  /* Compute the local matrix times vector */
  compMatrixTimesVector(num_local_rows, cols, local_A, b, local_y);
```

```c
/* Gather the result on process 0. recvcounts[i] is the number of
   doubles to be received from process i */
int *recvcounts;
if (my_rank==root)
  recvcounts = malloc(sizeof(int)*p);

/* Gather num_local_rows from each process */
MPI_Gather(&num_local_rows, 1, MPI_INT, recvcounts, 1,
           MPI_INT, root, comm);

/* Calculate displs for MPI_Gatterv */
int *displs;
if (my_rank==root)
  {
    displs = malloc(sizeof(int)*p);
    displs[0] = 0;
    int i;
    for (i = 1; i < p; i++)
      displs[i] = displs[i-1] + recvcounts[i-1];
  }

/* Gather y */
MPI_Gatherv(local_y, num_local_rows, MPI_DOUBLE,
            y, recvcounts, displs, MPI_DOUBLE, root, comm);
```

Allocating memory

- `malloc` may return 0
- MPI may crash
- The following function is a better `malloc`

```
void * Malloc(int num, int size, MPI_Comm comm, int rank)
{
  void *b = malloc(num*size);
  if (b==NULL)
    {
      fprintf(stderr,"*** PROCESS %d: MALLOC COULD NOT ALLOCATE"
              " %d ELEMENTS OF SIZE %d BYTES\n",
              rank, num, size);
      MPI_Abort(comm, 1);
    }
  return b;
}
```

Final remarks

- Amount of data sent must match amount of data received
- Blocking versions only
- No tags: calls are matched according to order of execution
- A collective function can return as soon as its participation is complete

The complete code is at `http://www.cas.mcmaster.ca/~nedialk/COURSES/4f03/code/parmatvec.zip`

- type `make`
- run `./do_all`
- study the code and the makefile

© 2013–14 Ned Nedialkov