Simulation and High-Performance Computing

Part 18: Collective Communication in MPI

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 8th, 2020

Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

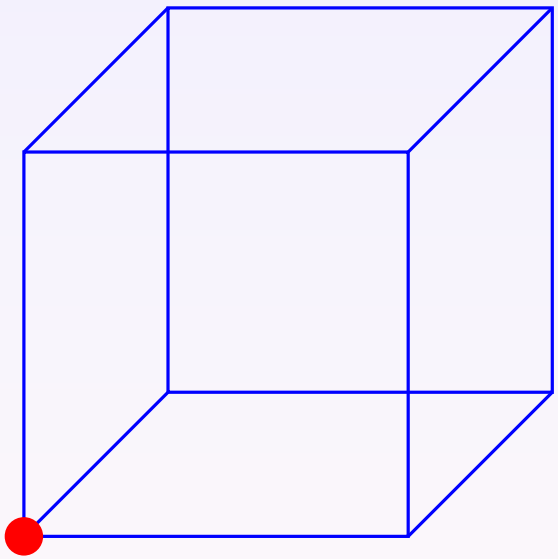
Example: Broadcast data from one process to all other processes.

Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.

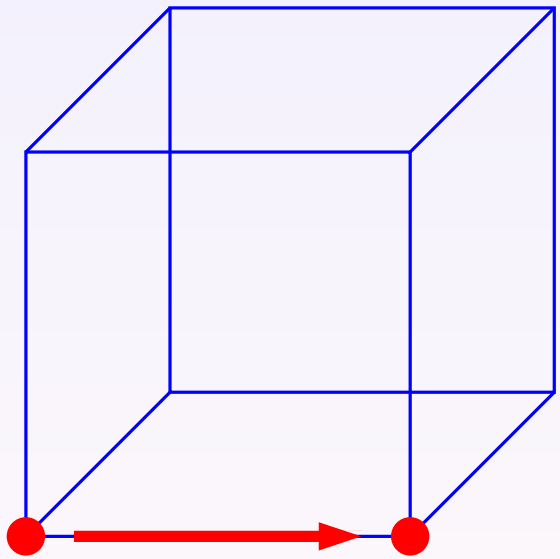


Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.

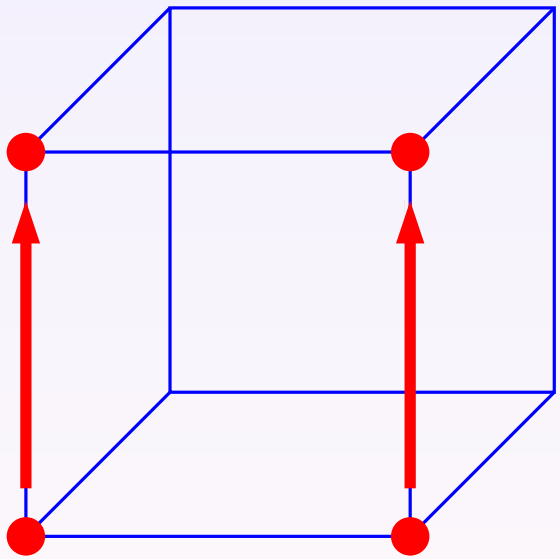


Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.

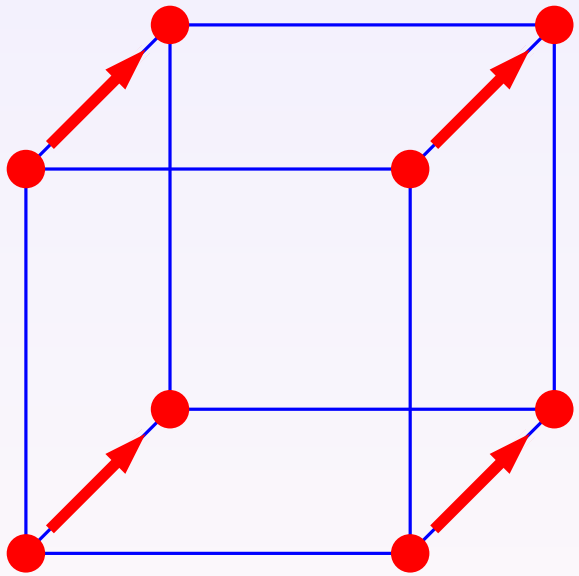


Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.

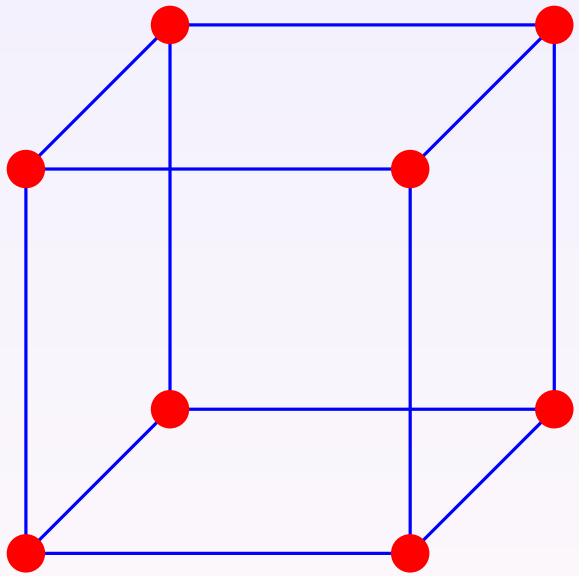


Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.

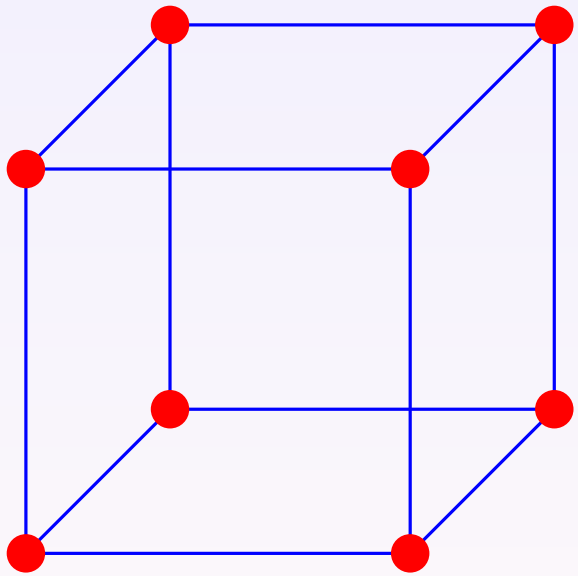


Collective communication

Idea: Some operations can be carried out very efficiently if all processes work together.

Example: Broadcast data from one process to all other processes.

If the network is a hypercube, we can broadcast very efficiently.



Result: p steps are sufficient to reach 2^p processes.

Broadcast and reduce

Broadcast: Send data from one process to all other processes.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype type,
              int root, MPI_Comm comm);
```

Reduce: Collect data from all processes in one process

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

The messages are merged by operations like `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, or `MPI_PROD`, i.e., multiple messages are reduced to one.

Example: Broadcast

Goal: Request user input and send the result to all processes.

```
MPI_Comm_rank(comm, &rank);

if(rank == 0) {
  printf("Number of timesteps?\n");
  scanf("%d", &n);
}

MPI_Bcast(&n, 1, MPI_INT, 0, comm);
```

Example: Reduction

Goal: Measure the runtimes for a computation on all processes and find the maximal time.

```
MPI_Comm_rank(comm, &rank);

t_start = MPI_Wtime();
/* ... do something ... */
t_run = MPI_Wtime() - t_start;

MPI_Reduce(&t_run, &t_max, 1, MPI_DOUBLE,
           MPI_MAX, 0, comm);
if(rank == 0)
  printf("  %.1f seconds maximum\n", t_max);
```

Example: Gravitation

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

Every process is responsible for m planets.

```
for(k=0; k<size; k++) {
  if(k == rank)
    memcpy(x, y, sizeof(double) * 4 * m);
  MPI_Bcast(x, 4 * m, MPI_DOUBLE, k, comm);

  for(i=0; i<m; i++)
    /* ... compute potential phi_k[i] ... */

  MPI_Reduce(phi_k, phi, m, MPI_DOUBLE, MPI_SUM, k, comm);
}
```

Experiment: Gravitation

Approach: Distribute 262 144 planets among several processes, evaluate total gravitational potential.

procs	time	ratio	speedup
1	483.04		
2	241.73	2.00	2.00
4	120.92	2.00	4.00
8	60.47	2.00	7.99
16	30.30	2.00	15.94
32	15.35	1.97	31.47

Result: Doubling the number of processes halves the runtime.

Scatter and gather

Scatter: One process sends different messages to all processes.

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

Gather: One process receives messages from all processes and stores them in an array.

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

Example: Wave equation

Reminder: In our domain-decomposition version of the leapfrog solver for the wave equation, every process stores its values of x in a local array `x[1]`,...,`x[n]`.

Goal: We want to gather all values in one large array, e.g., to visualize the result or store it on a hard disk.

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

if(rank == 0)
  x_all = (double *) malloc(sizeof(double) * n * size);

MPI_Gather(x+1,   n, MPI_DOUBLE,
           x_all, n, MPI_DOUBLE, 0, comm);
```

All-to-all

All-to-all: All processes send potentially different messages to all processes.

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

Variable-length all-to-all: All processes send potentially different messages with potentially different lengths to all processes.

```
int MPI_Alltoallv(const void *sendbuf, const int *sendcounts,
                  const int *sdispls, MPI_Datatype sendtype,
                  void *recvbuf, const int *recvcounts,
                  const int *rdispls, MPI_Datatype recvtype,
                  MPI_Comm comm);
```

Example: Exchanging variable-length messages

Approach: Every process has an array `sendcounts`, where `sendcounts[j]` is the number of data elements it will send to process j.

To use `MPI_Alltoallv`, every process needs an array `recvcounts`, where `recvcounts[i]` is the number of elements it will receive from process i.

```
MPI_Alltoall(sendcounts, 1, MPI_INT,
             recvcounts, 1, MPI_INT, comm);
sdispls = (int *) malloc(sizeof(int) * (size+1));
rdispls = (int *) malloc(sizeof(int) * (size+1));
sdispls[0] = 0; rdispls[0] = 0;
for(i=0; i<size; i++) {
  sdispls[i+1] = sdispls[i] + sendcounts[i];
  rdispls[i+1] = rdispls[i] + recvcounts[i];
}
/* ... allocate buffers, fill sendbuf ... */
MPI_Alltoallv(sendbuf, sendcounts, sdispls, MPI_DOUBLE,
              recvbuf, recvcounts, rdispls, MPI_DOUBLE, comm);
```

One-sided communication

Problem: The sending process has to know what elements of data the receiving process needs.

This can be problematic if only the receiving process can determine what data it needs.

One-sided communication: Processes provide a "window" into their address space, and other processes can read from and write to this window.

Non-blocking operations: Read and write operations to the window do not have to be completed immediately, but programs can wait for them.

Windows, put, and get

Windows into an address space can be opened in a collective operation.

```
int
MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm, MPI_Win *win);
```

Get and put: We can read from and write to a window.

```
int
MPI_Get(void *oaddr, int ocount, MPI_Datatype otype,
        int rank, MPI_Aint tdisp, int tcount, MPI_Datatype ttype,
        MPI_Win win);
int
MPI_Put(void *oaddr, int ocount, MPI_Datatype otype,
        int rank, MPI_Aint tdisp, int tcount, MPI_Datatype ttype,
        MPI_Win win);
```

Fences: Put and get operations need to be completed at a fence.

```
int MPI_Win_fence(int assrt, MPI_Win win);
```

Example: Unstructured sparse matrices

Sparse matrices contain mostly zero coefficients, e.g., because difference quotients only need immediate neighbours of a grid point.

Compressed row storage: For each row, we store only the non-zero coefficients and the corresponding columns.

For the sake of efficiency, they are all kept in two arrays `coeff` and `col`, and a third array `row` points to the first entry for a given row.

```
for(i=0; i<n; i++) {
  sum = 0.0;
  for(j=row[i]; j<row[i+1]; j++)
    sum += coeff[j] * x[col[j]];
  y[i] += alpha * sum;
}
```

Example: Distributed unstructured sparse matrices

Approach: Every process stores m rows of vectors and the sparse matrix.

```
MPI_Win_create(x, m * sizeof(double), sizeof(double),
               MPI_INFO_NULL, comm, &xwin);
MPI_Win_fence(0, xwin);

for(i=0; i<m; i++) {
  for(k=row[i]; k<row[i+1]; k++) {
    j = col[k]; jrank = j/m;
    MPI_Get(buf+k-row[i], 1, MPI_DOUBLE,
            jrank, j-m*jrank, 1, MPI_DOUBLE, xwin);
  }
  MPI_Win_fence(0, xwin);

  sum = 0.0;
  for(k=row[i]; k<row[i+1]; k++)
    sum += coeff[j] * buf[k-row[i]];
  y[i] += alpha * sum;
}
```

Summary

Collective communication operations allow all processes in a given communicator to work together.

Broadcast sends one message to all processes.

Reduce collects and merges messages from all processes.

Scatter and gather send or receive different messages from all processes.

All-to-all lets all processes send messages to all processes.

One-sided communication allows processes to access arrays owned by other processes. Put and get operations are non-blocking and need fences to wait for completion.