MPI — Collective Communications

Salvatore Filippone

salvatore.filippone@uniroma2.it

The problem of collective communications

Let's review an item from a previous example:

```
if (my_rank == 0) {
  for (i=1; i<p; i++) {
    MPI_Send(&a,1,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
  }
 } else {
  MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
 }
```

- This is a *collective communication* since *all* processes participate in its implementation;
- It is very frequent in parallel codes, and it's called a *Broadcast*: the data on one *root* process should be received by all other processes;
- Just from a software engineering perspective, it makes sense to encapsulate it in a function.

The problem of collective communications

How long does it take to execute this code?

```
if (my_rank == 0) {
  for (i=1; i<p; i++) {
    MPI_Send(&a,1,MPI_FLOAT,i,tag,MPI_COMM_WORLD);
  }
 } else {
  MPI_Recv(&a,1,MPI_FLOAT,0,tag,MPI_COMM_WORLD);
 }
```

A model of communication costs

Communicating N bytes requires between *any* two processes (sender and receiver) costs

$$T(N) = \lambda + \frac{N}{B},$$

where

- λ is the *latency*, a fixed cost to be paid for all messages;
- B is the *bandwidth*, the speed with which you can send data.

What determines these parameters?

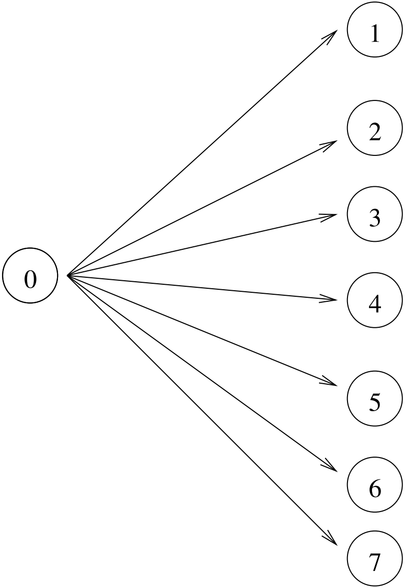
- λ depends almost entirely on the operating system stack (for performance, avoid TCP/IP!);
- B depends on the operating system stack *and* on the actual communication device.

Let's look at the structure of the previous code

- With fast networks, cost for 1 float is dominated by λ;
- Cost of this algorithm is therefore

$$T(p) \approx \lambda \cdot (p-1)$$

or, linear in p.



Communications

Can we do any better?

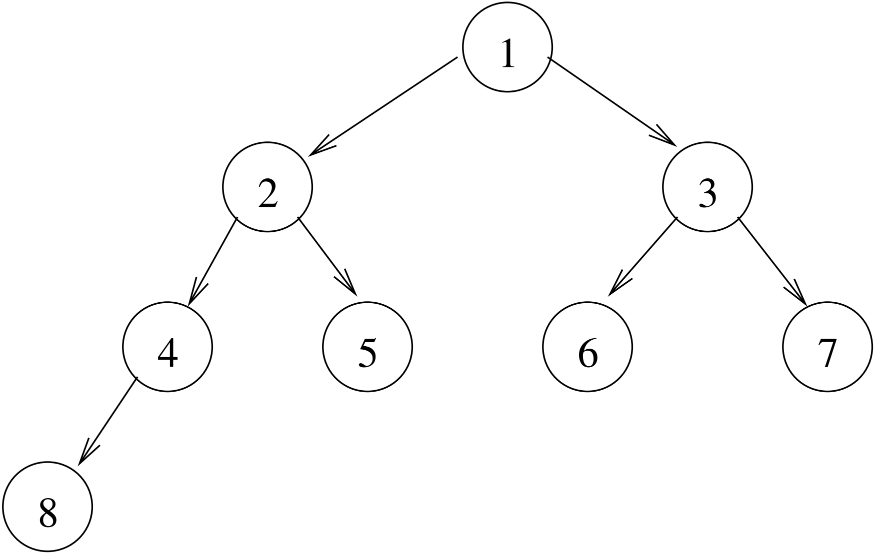
Let's make some assumptions about the network:

- The cost of communication between any two network nodes is uniform, and is given by $\lambda + N/B$;
- In particular, it is *possible* to send a message between *any* two nodes[1]
- The network is capable of sustaining multiple messages (*noisy topology*) at the same time, provided pairs of nodes involved in the messages do not overlap.

The latter assumption is especially important: we can improve communication if we can have multiple messages "flying" through the network at the same time.

[1] Historically there existed networks where only neighbouring nodes could exchange messages

Tree broadcast — simple minded



Tree broadcast — simple minded

Assume processes are numbered from 1:

- Each process i (except 1) receives from $\lfloor (p-1)/2 \rfloor$;
- Each process such that $2i <= p$ sends first to process $2i$, then to process $2i+1$.

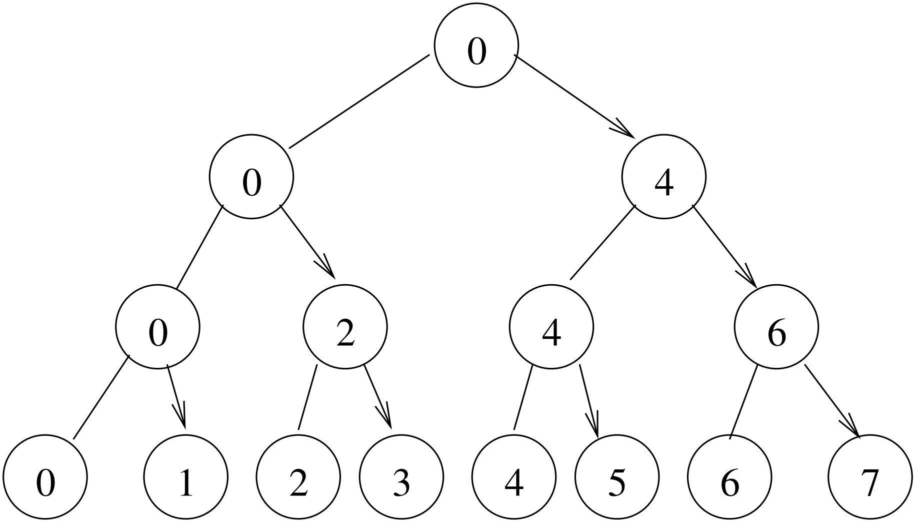
Cost:

$$T(p) \approx 2\log(p),$$

or logarithmic in p. To be precise, with $p > 1$ then

$$T(p) = \begin{cases} 0 & \text{for } p = 1 \\ 2 \cdot (k-1) + 1 & \text{for } p = 2^k, k > 0 \\ 2 \cdot \lfloor \log_2(p) \rfloor & \text{otherwise} \end{cases}$$

Tree broadcast — Recursive Doubling



Tree broadcast — Recursive Doubling

- Consider that there are p processes with root 0;
- Set K the minimum power of 2 such that $K \geq p$;
- Process 0 sends to process $K/2$;
- Divide the processes in two groups: from 0 to $K/2-1$, and from $K/2$ to $\min(p-1, K-1)$;
- Apply recursively to:
  - Processes 0 to $K/2-1$ with root 0;
  - Processes $K/2$ to $\min(p-1, K-1)$ with root $K/2$.

Cost:

$$T(p) = \lceil \log(p) \rceil,$$

or logarithmic in p.

Collective communications

Considerations for collective communications:

- Their functionality can be defined in terms of simple loops;
- There exist much better implementations;
- The optimal implementation for a given collective depends on:
  - The operation;
  - The network interface;
  - The network topology;
  - The amount of data.

A good MPI implementation will switch internally among different algorithms where appropriate (another advantage of encapsulating the collective)

Exercise: Under the same assumptions of point-to-point cost $\lambda + N/B$ and "noisy" networks, what is the optimal broadcast strategy $T(p, N)$ when the data size N is very large?

Transfer data from one process (*root*) to all others

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

Functionally equivalent to

```
if (my_rank == root) {
  for (i=0; i<p; i++) {
    if (i != root) MPI_Send(buffer,count,datatype,
                            i,tag,comm);
  }
 } else {
  MPI_Recv(buffer,count,datatype,root,tag,comm);
 }
```

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

Functionally equivalent to

```
if (my_rank == root) {
  memcpy(recvbuf,sendbuf,count*extent(datatype));
  for (i=0; i<p; i++) {
    if (i != root) {
      MPI_Recv(tempbuf,count,datatype,
               i,tag,comm);
      op(tempbuf,recvbuf,count,datatype);
    }
  }
 } else {
  MPI_Send(sendbuf,count,datatype,root,tag,comm);
 }
```

MPI Collective — Reduce

Predefined values for op: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD` and others.

You can create a new op with

```
int MPI_Op_create(MPI_User_function* user_fn,
                  int commute, MPI_Op* op);
```

with

```
typedef void MPI_User_function(void* invec,
                               void* inoutvec, int *len,
                               MPI_Datatype *datatype);
```

The operation op is *assumed* to be associative; if `commute == false` the order of the operands must be forced in ascending process rank order, see the naive implementation example in the MPI standard document for details.

MPI Collective — Reduce

What is the output of a collective communication?

Collective features

- If the underlying operation is *not* associative, the results *cannot* be the same with different number of processes;
- If the collective is implemented *without* enforcing ordering, even *two successive runs on the same machine* will give different outputs.

Warnings

- Never test a floating point result for exact match;
- Never expect a specific value from different machine configurations;
- Always use the result of a collective to govern global application behaviour;
- Always test results for appropriate bounds.

MPI Collective — Scatter

Take a set of data, split it and distribute it to the other processes.

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

MPI Collective — Scatter

Functionally equivalent to

```
if (my_rank == root) {
  for (i=0; i<p; i++)
    MPI_Send(sendbuf+i*sendcount*extent(sendtype),
                sendcount,sendtype,i,tag,comm);
 }
MPI_Recv(recvbuf,recvcount,recvtype,root,tag,comm);
```

MPI Collective — ScatterV

```
int MPI_Scatterv(const void* sendbuf, const int sendcounts[],
                 const int displs[], MPI_Datatype sendtype,
                 void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm com
```

Functionally equivalent to

```
if (my_rank == root) {
  for (i=0; i<p; i++)
    MPI_Send(sendbuf+displs[i]*extent(sendtype),
             sendcounts[i]*extent(sendtype),
             sendtype,i,tag,comm);
 }
MPI_Recv(recvbuf,recvcount,recvtype,root,tag,comm);
```

MPI Collective — Gather

Inverse of scatter

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

Functionally equivalent to

```
MPI_Send(sendbuf,sendcount,sendtype,root,tag,comm);
if (my_rank == root) {
  for (i=0; i<p; i++)
    MPI_Recv(recvbuf+i*recvcount*extent(recvtype),
             recvcount,recvtype,i,tag,comm);
 }
```

MPI Collective — GatherV

```
int MPI_Gatherv(const void* sendbuf, int sendcount,
                MPI_Datatype sendtype,void* recvbuf,
                const int recvcounts[], const int displs[],
                MPI_Datatype recvtype, int root, MPI_Comm comm
```

Functionally equivalent to

```
MPI_Send(sendbuf,sendcount,sendtype,root,tag,comm);
if (my_rank == root) {
  for (i=0; i<p; i++)
    MPI_Recv(recvbuf+displs[i]*extent(recvtype),
             recvcounts[i],recvtype,i,tag,comm);
 }
```

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

```
int MPI_Alltoallw(const void* sendbuf, const int sendcounts[],
                  const int sdispls[],
                  const MPI_Datatype sendtypes[],
                  void* recvbuf, const int recvcounts[],
                  const int rdispls[],const
                  MPI_Datatype recvtypes[],
                  MPI_Comm comm)
```

MPI Collective — Others

Gather-to-all: equivalent to a gather followed by a broadcast;

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

reduce-scatter: name says all;

Scan: left as an exercise to the reader;

Barrier: synchronization of all processes;

Non-blocking: variants of the other collectives.