MPI lecture and labs 5

Victor Eijkhout

2016

Sub-computations

Sub-computations

Simultaneous groups of processes, doing different tasks, but loosely interacting:

- Simulation pipeline: produce input data, run simulation, post-process.
- Climate model: separate groups for air, ocean, land, ice.
- Quicksort: split data in two, run quicksort independently on the halves.
- Processor grid: do broadcast in each column.

New communicators are formed recursively from `MPI_COMM_WORLD`.

Communicator duplication

Simplest new communicator: identical to a previous one.

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

This is useful for library writers:

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

Use of a library

```
library my_library(comm);
MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
my_library.communication_start();
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
    comm,&(request[1]));
MPI_Waitall(2,request,status);
my_library.communication_end();
```

Use of a library

```
int library::communication_start() {
int sdata=6,rdata;
MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
    comm,&(request[1]));
return 0;
}

int library::communication_end() {
MPI_Status status[2];
MPI_Waitall(2,request,status);
return 0;
}
```

Wrong way

```
// commdup_wrong.cxx
class library {
private:
  MPI_Comm comm;
  int mytid,ntids,other;
  MPI_Request *request;
public:
  library(MPI_Comm incomm) {
    comm = incomm;
    MPI_Comm_rank(comm,&mytid);
    other = 1-mytid;
    request = new MPI_Request[2];
  };
  int communication_start();
  int communication_end();
};
```

Right way

```
// commdup_right.cxx
class library {
private:
  MPI_Comm comm;
  int mytid,ntids,other;
  MPI_Request *request;
public:
  library(MPI_Comm incomm) {
    MPI_Comm_dup(incomm,&comm);
    MPI_Comm_rank(comm,&mytid);
    other = 1-mytid;
    request = new MPI_Request[2];
  };
  ~library() {
    MPI_Comm_free(&comm);
  }
  int communication_start();
```

Disjoint splitting

Split a communicator in multiple disjoint others.

Give each process a ‘colour’, group processes by colour:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)
```

Row/column example

```
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;

MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );

MPI_Bcast( data, ... column_comm );
```

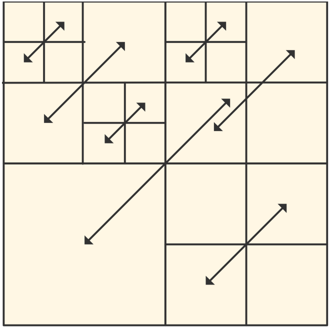
Exercise 1

Organize your processors in a grid, and make subcommunicators for the rows and columns. Do a broadcast from the first row and column through the columns and rows respectively.

If you let the broadcast value be the column/row number, then processor (i,j) winds up with the numbers i and j. Test this.

Exercise 2

Implement a recursive algorithm for matrix transposition:



- Swap blocks $(1,2)$ and $(2,1)$; then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.

More

- Non-disjoint subcommunicators through process groups.
- Intra-communicators and inter-communicators.
- Process topologies: cartesian and graph.

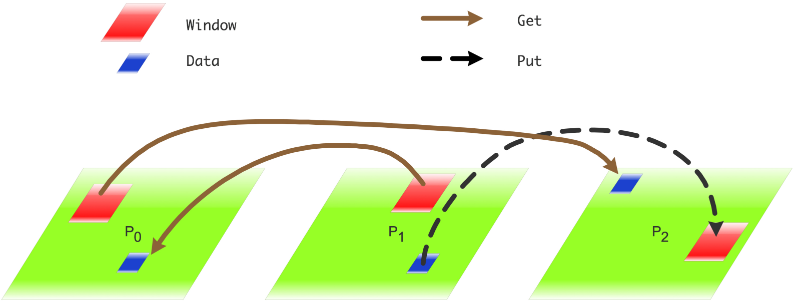
One-sided communication

Motivation

With two-sided messaging, you can not just put data on a different processor: the other has to expect it and receive it.

- Sparse matrix: it is easy to know what you are receiving, not what you need to send. Usually solved with complicated preprocessing step.
- Neuron simulation: spiking neuron propagates information to neighbours. Uncertain when this happens.
- Other irregular data structures: linked lists, hash tables.

One-sided concepts



- A process has a *window* that other processes can access.
- Origin: process doing a one-sided call; target: process being accessed.
- One-sided calls: `MPI_Put`, `MPI_Get`, `MPI_Accumulate`.
- Various synchronization mechanisms.

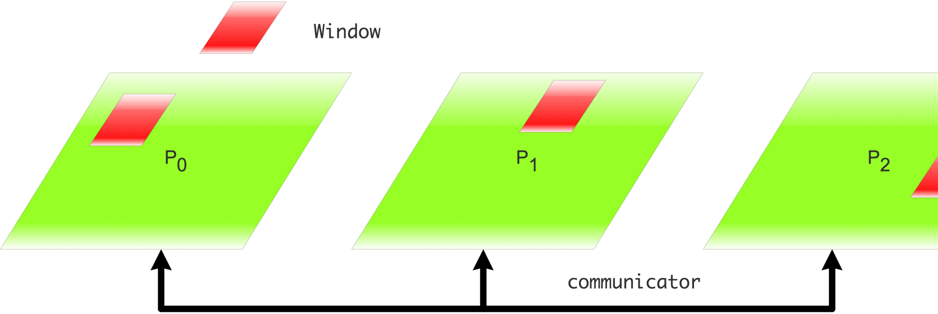
Active target synchronization

All processes call `MPI_Win_fence`. Epoch is between fences:

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
if (mytid==producer)
  MPI_Put( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

Second fence indicates that one-sided communication is concluded: target knows that data has been put.

Window creation




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

- `size`: in bytes
- `disp_unit`: `sizeof(type)`
- Also: `MPI_Win_allocate`, can use dedicated fast memory.

```
C:
int MPI_Put(
  const void *origin_addr, int origin_count, MPI_Datatype origin_datat
  int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatyp
  MPI_Win win)

Semantics:
IN origin_addr: initial address of origin buffer (choice)
IN origin_count: number of entries in origin buffer (non-negative inte
IN origin_datatype: datatype of each entry in origin buffer (handle)
IN target_rank: rank of target (non-negative integer)
IN target_disp: displacement from start of window to target buffer (no
IN target_count: number of entries in target buffer (non-negative inte
IN target_datatype: datatype of each entry in target buffer (handle)
IN win: window object used for communication (handle)

Fortran:
MPI_Put(origin_addr, origin_count, origin_datatype,
  target_rank, target_disp, target_count, target_datatype, win, ierro
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_dataty
```

Exercise 3

Write code where process 0 randomly writes in the window on 1 or 2.

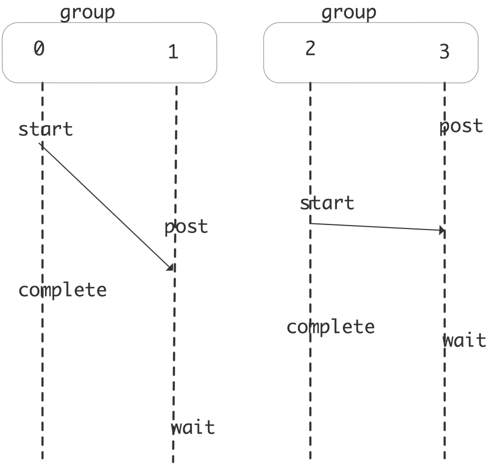
```
// randomput_skl.c
MPI_Win_create(&window_data,sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&the_window);

for (int c=0; c<10; c++) {
  float randomfraction = (rand() / (double)RAND_MAX);
  if (randomfraction>.5)
other = 2;
  else other = 1;
  window_data = 0;
  your_code_goes_here.........
  my_sum += window_data;
}

if (mytid>0 && mytid<3)
  printf("Sum on %d: %d\n",mytid,my_sum);
```

A second active synchronization

Use `Post,Wait,Start,Complete` calls



More fine-grained than fences.

Passive target synchronization

Lock a window on the target:

```
MPI_Win_lock (int locktype, int rank, int assert, MPI_Win win)
MPI_Win_unlock (int rank, MPI_Win win)
```

Atomic operations:

```
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
        MPI_Datatype datatype, int target_rank, MPI_Aint target_
        MPI_Op op, MPI_Win win)
```

```
// passive.cxx
if (mytid==repository) {
  // Processor zero creates a table of inputs
  // and associates that with the window
}
if (mytid!=repository) {
  float contribution=(float)mytid,table_element;
  int loc=0;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
    // read the table element by getting the result from adding
    err = MPI_Fetch_and_op(&contribution,&table_element,MPI_FLO
                  repository,loc,MPI_SUM,the_window); CHK(err);
    MPI_Win_unlock(repository,the_window);
}
```