L7 # COLLECTIVE COMMUNICATION ①

PPMPI CH5

issues with the trap. rule program: (on 8 procs)

(a) procs 1-7 are idle, while proc 0 performs I/O

(b) after proc 0 has collected data, INPUT higher rank procs ~~[illegible]~~ continue to wait, until proc 0 sends input data to lower rank procs.

(c) proc 0 collects all partial answers and performs addition

main point of parallel computing: multiple procs collaborate on solving a problem

TREE-STRUCTURED comm/tion

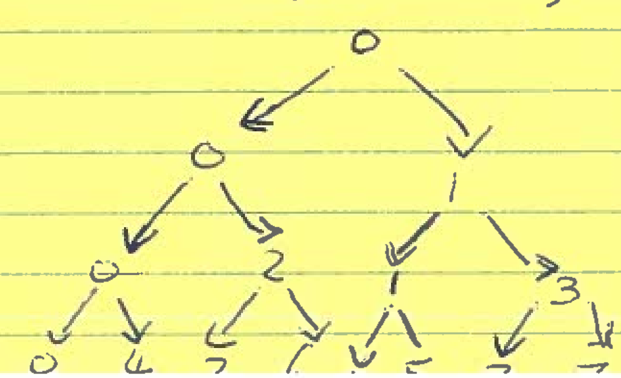
focus on the distribution of the input data

divide the work more evenly among procs

imagine that we have a tree of procs with 0 at the root.

1st stage 0 sends the data to 1
2nd " 0 ~> 2 & 1 ~> 3
3rd " 0 ~> 4 & 1 ~> 5, 2 ~> 6, 3 ~> 7

we reduced the input distrib. loop, from 7 stages ~~[illegible]~~ to 3 stages



ceiling ②

if we have p proc's, the input data is distributed in $\lceil \log_2 p \rceil$ stages (rather than p-1)

case study: $\log_2 8 = 3$ (previous example)

$\log_2 1024 = 10$ (reduction by a factor of 100)

modify Get_data fct to use a tree-structured distribution scheme:

```
for (stage=first; stage<=last; stage++)
    if (I_receive(stage, my_rank, &source))
        Receive (data, source);
    else if (I_send(stage, my_rank, p, &dest))
        Send (data, dest);
```

I_receive fct returns $\begin{cases} 1, & \text{calling process receives data during current stage} \\ 0, & \text{otherwise} \end{cases}$

if the calling proc. receives data, the parameter "source" is used to return the rank of the sender.

I_send fct returns $\begin{cases} 1, & \text{proc sends during current stage} \\ 0, & \text{otherwise} \end{cases}$

implementation: we need to calculate
(1) whether a proc. receives, and the source
(2) " " " sends, " " destination

(several possible tree-schemes are possible. no canonical choice of ordering
deciding the best scheme requires some knowledge of the topology of our system.)

general tree scheme:

stages numbered starting at 0, 1, ...

if $2^{stage} \le$ my_rank $< 2^{stage+1}$
then I_receive from proc my_rank $- 2^{stage}$

if my_rank $< 2^{stage}$
then I_send to proc my_rank $+ 2^{stage}$

(2nd version of Get_data fct)

```
int I_receive (
        int stage       ] IN
        int my_rank     ] PARAMS
  out [ int* source_ptr ) {
PARAM
    int power_2_stage;        power_2_stage = 1<< stage
    if ((power_2_stage <= my_rank)
          && (my_rank < 2*power_2_stage))
        { *source_ptr = my_rank - power_2_stage;
          return 1;
        }
    else return 0;
}
```

$2^{\wedge}stage$ = 1<< stage

```
                   IN PARAMS
int I_send (int stage, int my_rank, int p,
            int* dest_ptr)      PARAM OUT
...
```

```
void Send (                                4
              float a,                    IN
              float b,                    PARAMS
              int n,
              int dest) {

MPI_Send (&a, 1, MPI_FLOAT, dest, 0, MPI_C_W);
MPI_Send (&b, 1,   "        "  , 1,   "   );
MPI_Send (&n, 1, MPI_INT,   "  , 2,   "   );
}

void Receive (
              float* a_ptr,    OUT
              float* b_ptr,    PARAMS
              int*   n_ptr,
              int    source )  IN PARAM
...
                          OUT PARAMS
                    float*    float*    int*
void Get_data1( a_ptr, b_ptr, n_ptr,
                int my_rank, int p ) {
                    IN PARAMS
   int source, dest, stage;

   if (my_rank == 0) { scanf(a_ptr, b_ptr,
                                  n_ptr);
   }
   for(stage=0; stage < log2p; stage++)
      if ( I_receive(stage, my_rank, &source))
           Receive(a_ptr, b_ptr, n_ptr, source);
   } else if I_send( ... )
}          Send (*a_ptr, *b_ptr, *n_ptr, dest);
```

BROADCAST

A communication pattern that involves all the proc^s in a comm/tor is a collective communication.

A broadcast is a collective comm/tion in which a single process sends the same data to every process in the comm/tor.

tree-structured com/tion is much more efficient than a sequential from the root node

```
int MPI_Bcast (
    void*         message,    IN OUT PARAM  (IN on root proc, OUT on other proc^s)
    int           count,      IN PARAMS
    MPI_Datatype  datatype,   IN PARAMS
    int           root,       IN PARAMS
    MPI_Comm      comm )      IN PARAMS
```

(hand-coded is effic?)

send a copy of the data in "message" on the proc with rank "root" to each process in the comm/tor "comm".

should be called by ALL proc^s in the comm/tor, with the same arg^s for root/comm.

count/datatype: specify how much memory is needed for the message.

→ these two parameters should be the same on all the proc^s of the comm/tor.

(no tags in collective comm/tions mechanism)

the reason is that in some cases a single proc receives data from many other proc^s. so we would need a tag

OUT

void Get_data2(a_ptr, b_ptr, n_ptr, my_rank)

IN

```
{
  if(my_rank==0) { scanf... }
  MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MCW);
    "     "   (b_ptr, ",  ",     0, " );
    "     "   (n_ptr, ", MPI_INT, 0, " );
}
```

~~zero also executes this part~~

faster & more easily comprehensible version.

TAGS, SAFETY, BUFFERING, SYNCHRONIZATION

MPI_Bcast does not use tags, WHY?

MPI_Send / MPI_Recv
use tags: proc A sends several msgs to proc B, and B handles them according to their tags.

example 1
consider this seq. of events

TIME	Proc A	Proc B
1	MPI_Send to B tag=0	local work
2	MPI_Send to B tag=1	local work
3	local work	MPI_Recv from A tag=1
4	local work	MPI_Recv from A tag=0

this sequence requires <u>buffering</u>
(set aside memory for storing msg^s,
before a "receive" has been executed)

msg. env. ~> {rank of sender/receiver
tag, comm/tor}

until B calls MPI_Recv, the system
does not know where the msg that A
is sending should be stored.

when B calls MPI_Recv, the system
looks for any buffered msg^s that
has an env. that matches the recv. param^s.
if there is no such message,
then it will wait until one arrives.

if no buffering is available A cannot
send data to B, until it knows that
B is ready to receive.
send cannot complete until receiver is ready to receive ‖ send uses synchronous mode

if a program that assumes buffering is
available, is run on a system that
does not provide buffering ~> DEADLOCK
UNSAFE MPI Program
A hangs while it waits for B to receive
1st send

B hangs while it waits for A to execute
2nd send

example 2 (uses Bcasts) (—— ↦ LOCAL WORK)

TIME	Proc A	Proc B	Proc C
1	MPI_Bcast &x	——	——
2	MPI_Bcast &y	——	——
3	——	MPI_Bcast &y	MPI_Bcast &x
4	——	MPI_Bcast &x	MPI_Bcast &y

suppose A broadcasts two floats x, y
in (that x=5, y=10 on proc A) to B, C

when bcasts are completed on all 3 proc^s, x=5, y=10 on proc^s A, C

but on proc B, x=10, y=5 // reversed values WHY?

⚠ first parameter of Bcast is IN/OUT

broadcasts assumed synchorization: on a given process the bcast would not return until every process had received the bcast data.

this restriction is relaxed, when buffering is available, A can complte its bcasts, before B, C begin their bcast calls

BUT the EFFECT in terms of data communicated, must be the same, as if there was sychronization

so 1st BCAST on B matches 1st BCAST on A and stores 5 in y