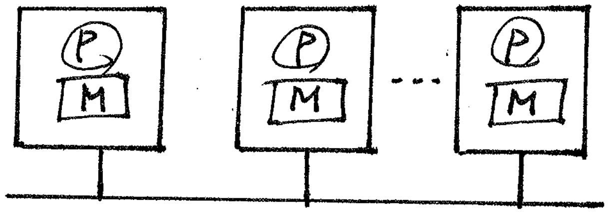
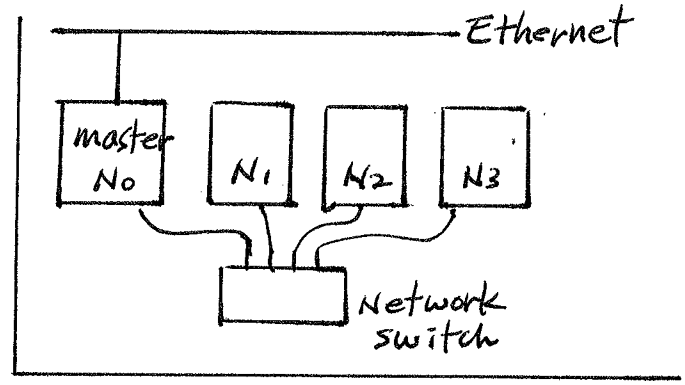
Ch3 Distributed-mem programming with MPI



IN e.g.) MPP, COW

msg-passing interface



$> mpicc ...

$> mpic++ p1.cpp ↵

$> mpiexec -n 4 ./a.out ↵
   mpirun -np 4 ./a.out ↵

MPI_Init (NULL, NULL); // initialize a MPI session

⋮

⋮

MPI_Finalize();

— MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

MPI_COMM_WORLD: Communicator, Collection of user defined processes

comm_size: int comm_size

gets the number of processes from -n 4

— MPI_comm_rank (MPI_COMM_WORLD, &my_rank);

my_rank: int my_rank

gets my_rank among comm_size

— SPMD programming
different task on different process

— 2 basic communication functions (p2p : point-to-point)

$$\texttt{int MPI\_send}(\underbrace{\underbrace{\texttt{void*}}_{\text{msg buffer}}, \underbrace{\texttt{int}}_{\text{msg size}}, \underbrace{\texttt{MPI\_Datatype}}_{\text{Table 3.1}}}_{\text{(msg)}}, \underbrace{\underbrace{\texttt{int}}_{\text{dest rank}}, \underbrace{\texttt{int}}_{\text{tag}}, \underbrace{\texttt{MPI\_Comm}}_{\text{communicator}}}_{\text{(dest)}});$$

tag: (special mark on msg)

Table 3.1 — ex) MPI_CHAR
MPI_short
MPI_int
MPI_float
MPI_double
...

Should match

$$\texttt{int MPI\_receive}(\underbrace{\underbrace{\texttt{void*}}_{\text{msg buffer}}, \underbrace{\texttt{int}}_{\text{msg size}}, \texttt{MPI\_Datatype}}_{\text{msg receiving space (mem)}}, \underbrace{\underbrace{\texttt{int}}_{\text{source rank}}, \underbrace{\texttt{int}}_{\text{tag}}}_{\text{(source)}}, \underbrace{\texttt{MPI\_Comm}}_{\text{communicator}}, \underbrace{\texttt{MPI\_status*}}_{\text{status pointer (can ignore)}});$$

— for a successful communication,

Send_comm = recv_comm
send_tag = recv_tag
dest = source

also, send_type = recv_type
send_buffer_size ≤ recv_buffer_size

— sender — must specify receiver
receiver — can use wild card — "MPI_ANY_Source" — (for source rank)

— problem : receiver can receive a msg with unknown {size, source, tag}

$\underbrace{\text{size}}_{\text{MPI\_Get\_Count(..)}}$, $\underbrace{\text{source, tag}}_{\text{using status}}$

sol:

↓

↓

ex)

⋮

MPI_Status status;

⋮

MPI_Recv(...., &status);

⋮

(Status.MPI_Source ⇝ source
(Status.MPI_Tag ⇝ tag

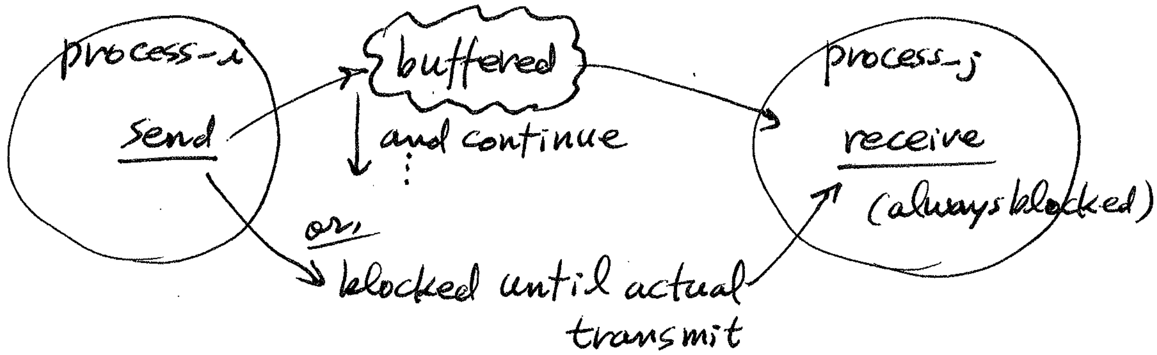
⋮

MPI_Get_count(&status, recv_type, &count);

size of msg

— should avoid hanging receive (#sender)

— Semantics of MPI_Send / MPI_Recv



— hybrid scheme:

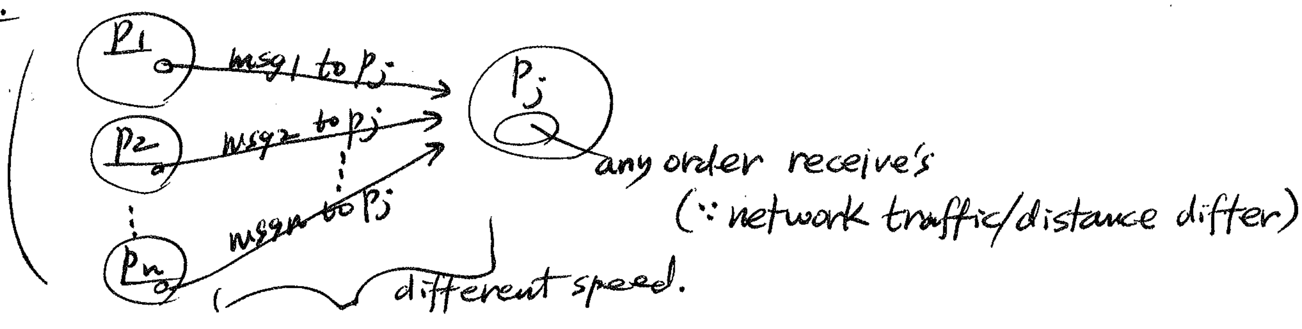
using cut-off, msg_size < cut-off ⇒ buffered
else ⇒ blocked

— Non_overtaking (used in MPI) — between 2 processes

process_i	process_j
– send msg1 to P_j	– receive msg1
– send msg2 to P_j	– receive msg2

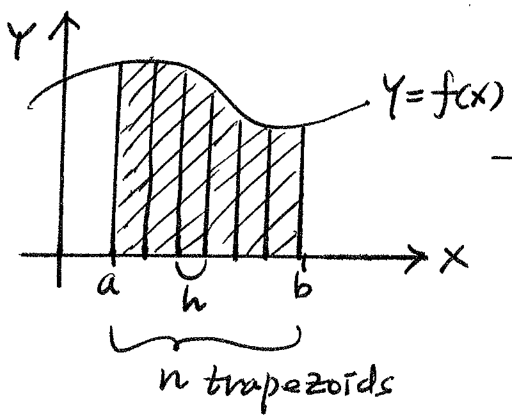
keep same order as sender process.

vs.

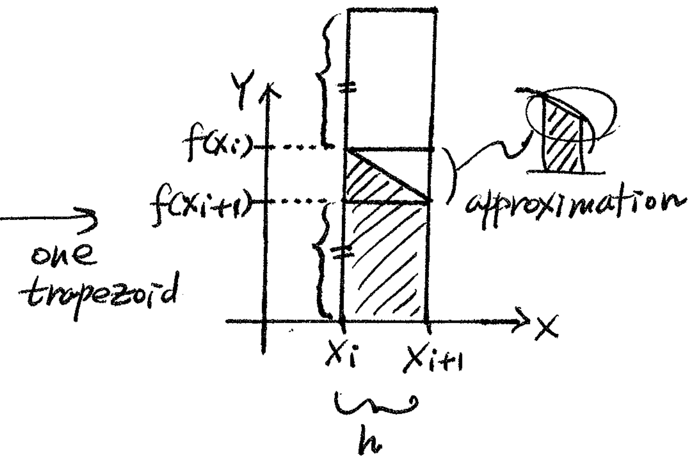


— Trapezoid computation in MPI

review

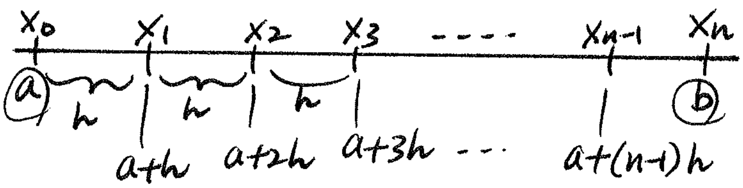


$\left(h = \frac{b-a}{n}\right)$



— local integral

$$= \left(\frac{f(x_i) + f(x_{i+1})}{2}\right) * h$$



— global integral

$$= \left(\frac{f(x_0) + f(x_1)}{2}\right) * h + \left(\frac{f(x_1) + f(x_2)}{2}\right) * h + \cdots + \left(\frac{f(x_{n-1}) + f(x_n)}{2}\right) * h$$

$$= \frac{h}{2}\left[(f(x_0) + f(x_1)) + (f(x_1) + f(x_2)) + (f(x_2) + f(x_3)) + \cdots + (f(x_{n-1}) + f(x_n))\right]$$

$$= h\left[\frac{f(x_0)}{2} + \underbrace{f(x_1) + f(x_2) + \cdots + f(x_{n-1})} + \frac{f(x_n)}{2}\right]$$

Serial code

```
h = (b-a)/n;
approx = (f(a)+f(b))/2;

for(i=1 ~ n-1)
  ( xi = a + i*h;
    approx += f(xi);

approx = approx*h;
```

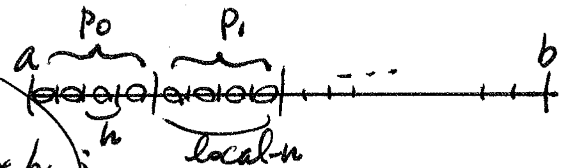
— Trapezoid — parallel code (MPI) — msg passing

concept
- divide n into $\frac{n}{p}$
- each process computes $\frac{n}{p}$ trapezoids
- process_ϕ combines local integrals

```
get a, b, n ;   — using only 1 process, get inputs and
                  send them to all other processes
h = (b-a) x n ;
local_n = n/p ;
local_a = a + my_rank * local_n * h ;
local_b = local_a + local_n * h ;

local_integral = Trap(local_a, local_b, local_n, h) ;

if (my_rank ≠ ϕ)
    send local_integral to process_ϕ ;
else
    tot_integral ← local_integral (of P0) ;
    for (i=1 ~ p-1)
        Receive local_integral from Process_i ;
        tot_integral += local_integral ;
    display tot_integral ;
```



- I/O for MPI program (suggestion)
  - any process accesses std_out
  - only process_ϕ accesses std_in

- Collective Comm ($\neq$ p2p — send/receive)

Communication function involves all the processes in a communicator.

ex) global sum computation

- MPI_Reduce (&local_integral, — input data ⎫ must be
  &tot_integral, — output data ⎭ different
  1, — count
  MPI_Double, — data type
  MPI_sum, — see Table 3.2 (MPI_MAX, MPI_MIN, MPI_prod, MPI_sum, ⋮)
  ∅, — dest rank
  MPI_Comm_World);

1. all processes must call the same collective func.

   ex) one process calls MPI_Recv(--), others call MPI_Reduce(--) → crash

2. arguments (from processes) to collective func. must be compatible.

3. output data arg. is only used in dest-process; others should use NULL.

4. ∄ tags and calling order is used for matching

ex)

time	process_∅ (dest)	process_1	process_2
T_0	a=1; c=2;	a=1; c=2;	a=1; c=2;
T_1	MPI_Reduce (&a, &b, ..);	MPI_Reduce (&c, &d, --);	MPI_Reduce (&a, &b, --);
T_2	MPI_Reduce (&c, &d, --);	MPI_Reduce (&a, &b, --);	MPI_Reduce (&c, &d, --);

Assume, MPI_SUM

- expected output: b=3, d=6
- actual output: b=1+2+1=4, d=2+1+2=5

Collective Communication

– MPI_Reduce

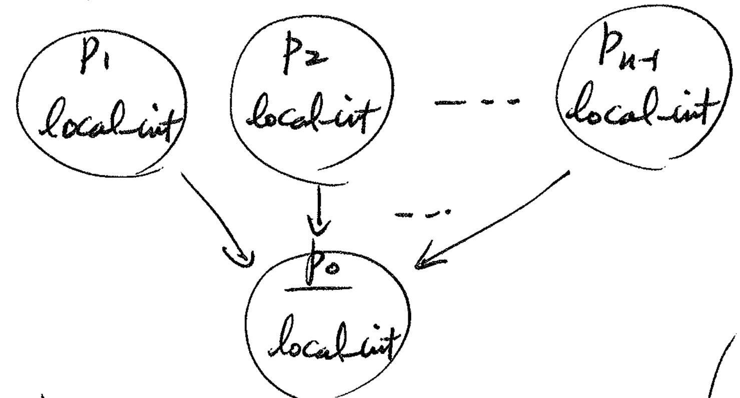
ex) trapezoid computing

p2p way

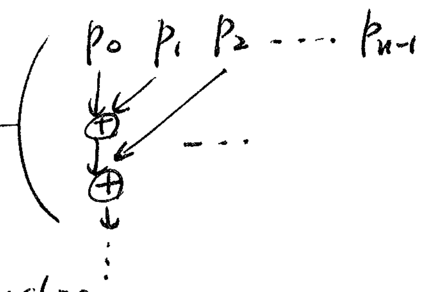
```
if (my_rank != 0)
    MPI_Send(&local_integral, 1, MPI_Double, 0, 0,
             MPI_Comm_World);
else // (P0)
    tot_integral = local_integral;
    for (source = 1; source < comm_size; source++)
        MPI_Recv(&local_integral, 1, MPI_Double, source, 0,
                 MPI_Comm_World,
                 MPI_Status_Ignore);
        tot_integral += local_integral;
```

dest(P0) tag = tag

→ it looks like:



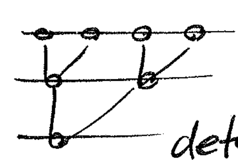
but, P_0 performs serial
tot_int += local_int



⇒ better way is using full-tree reduction.

MPI_Reduce(&local_integral, &tot_integral, 1, MPI_Double,
MPI_Sum, 0, MPI_Comm_World);

reduction operation   dest P_0

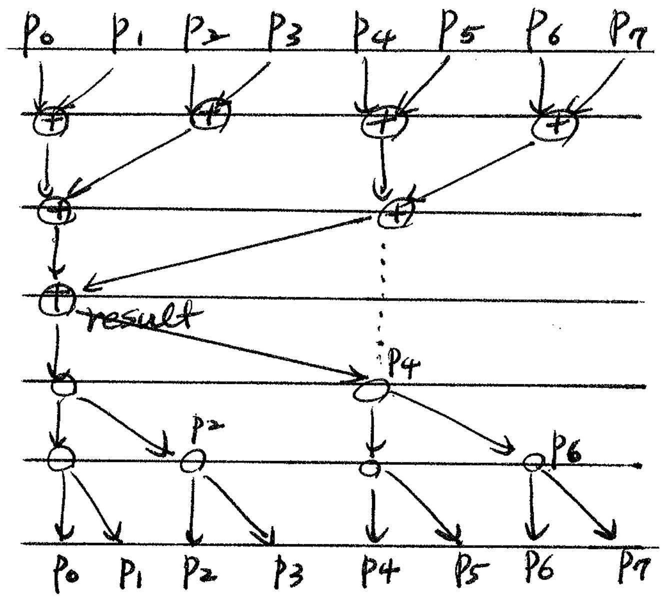


determine partner...

— <u>MPI_Allreduce</u> — useful for the situation that all processes need the reduced result. (for further operation)

— naive method: <u>reduce</u> to P_0 and <u>broadcast</u> to all

ex)



reduce ($\log_2 n$)

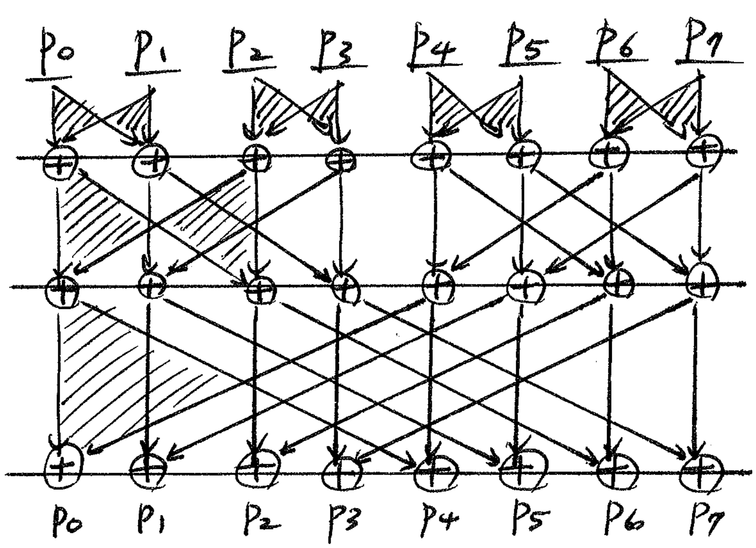
+

broadcast ($\log_2 n$)

ex) trapezoid computing

<u>MPI_Allreduce</u> (&local_integral, &tot_integral, 1, MPI_Double, MPI_Sum, MPI_Comm_World);

dest_process (all processes are destination)



— butterfly communication

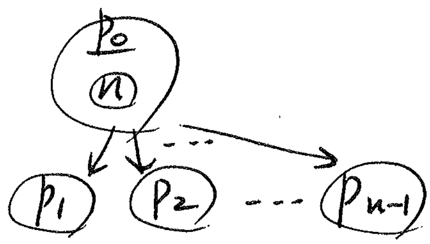
$\log_2 n$

$P = 8$

$\log_2 8 = 3$ levels

— MPI_Bcast

ex) p_0 accesses input n and distributes it to all others



— in fact, this is serial in p-2-p comm. with for-loop way, i.e.,

send to p1;
send to p2;
⋮
send to p_{n-1};

p2p way

```
if (my_rank==0) //P0
    n= atoi (argv[1]);
    for( dest=1 ; dest<comm_sz; dest++)
        MPI_Send (&n, 1, MPI_Int, dest, 0, MPI_Comm_World);
                                        tag
else //other processes
    MPI_Recv (&n, 1, MPI_Int, 0, 0, MPI_Comm_World);
                              source  tag
                              P0
```

vs.

broadcast way

```
if (my_rank==0)
    n= atoi (argv[1]);
MPI_Bcast (&n, 1, MPI_Int, 0, MPI_Comm_World);
                           source
                           P0
```

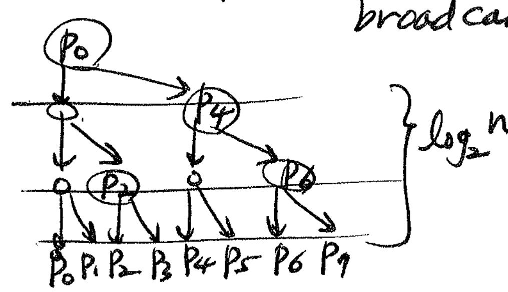
&n — in para — for source process; out para — for other processes

(*) Should call this out of P_0

∵ P_0 — sends; others — receive

— MPI_Bcast performs full tree broadcasting.

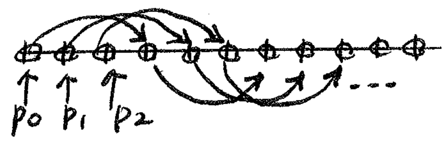
ex)



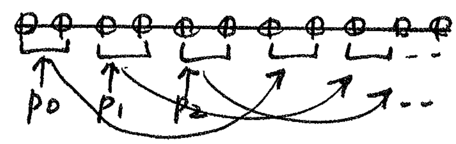
— Data distribution

ex) $p=3$

- block partition — $\underbrace{0,1,2,3}_{p_0}$, $\underbrace{4,5,6,7}_{p_1}$, $\underbrace{8,9,10,11}_{p_2}$
- cyclic partition :
  - p_0 : 0, 3, 6, 9
  - p_1 : 1, 4, 7, 10
  - p_2 : 2, 5, 8, 11



- block-cyclic partition
  - p_0 : $\underline{0,1}$, $\underline{6,7}$
  - p_1 : $\underline{2,3}$, $\underline{8,9}$
  - p_2 $\underline{4,5}$, $\underline{10,11}$



— MPI_Scatter

Assume: n is evenly divisible by p, and <u>block partition</u>.

— process_φ receives (accesses) size n vector, and

send ~~the~~
- 1st $\frac{n}{p}$ elements to p_0
- 2nd n/p elements to p_1
- 3rd n/p elements to p_2, and so on
- ...

(ref: not evenly divisible case, use MPI_Scatter<u>V</u>)

ex)

```
if(my_rank==φ)//p0
{ int* a = new int[n];
  get input values for a[n] array (vector);
  MPI_Scatter(a, local_n, MPI_Int, local_a, local_n,
              MPI_Int, φ, comm);
  delete[] a;
}
else
  MPI_Scatter(a, local_n, MPI_Int, local_a, local_n,
              MPI_Int, φ, comm);
```

p_0: a — (sending buffer*); local_n — (sending buffer size — local size — to each process); MPI_Int — type (sender); local_a — (receiving buffer* — int local_a[]); local_n — (receiving buffer size); MPI_Int — type (receiver); φ — source p_0

others: a, local_n, MPI_Int — sending buffer info; local_a, local_n, MPI_Int, φ, comm — receiving buffer

— Performance/time checking

— time checking for only computation part (not for elapsed time)

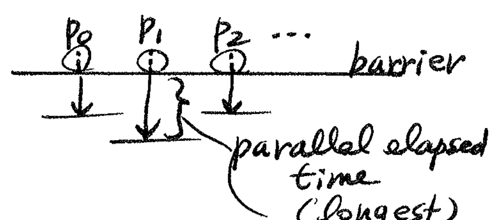
```
#include "timer.h"   → macro def
  :                      #define Get_Time(now)\
double start, finish;    { struct timeval t;\
  :                        gettimeofday(&t, NULL);\   (posix lib. func)
[ Get_Time(start);         now = t.tv_sec + \
  :  ///                       t.tv_usec/1000000.0; }
[ Get_Time(finish);                                   needs <sys/time.h>
display(finish-start);  — in μ seconds
```

— MPI supports MPI_Wtime()

```
double start, finish;
[ start = MPI_Wtime();
    ///                    — each process reports exec time.
[ finish = MPI_Wtime();
display my_rank, (finish-start);  — in μ seconds
```

— Parallel time checking — slowest process's exec. time

```
[ MPI_Barrier(MPI_Comm_World);
  local_start = MPI_Wtime();
  [ ///
  local_finish = MPI_Wtime();
  local_elapsed = local_finish - local_start;
  MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_Double,
                      MPI_Max, 0, MPI_Comm_World);
                      (reduction operation)  (dest = P0)
  If(my_rank == 0)
      display elapsed;
```



— Running MPI program on a hybrid system, each node is multicored.

— by default, only 1 process works on each node.

⇒ MPI + OpenMP model

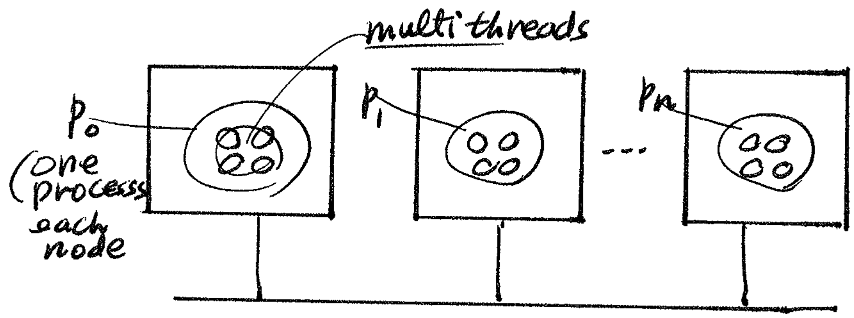
in each node (multicored)

1 process

ex) pragma omp parallel for ···

— How to compile/run?

```
$> mpic++ -fopenmp pl.cpp ↵
$> mpirun -n 4 ./a.out ↵
```



ex)

```
#incl(u)de <mpi.h>
#include <omp.h>
···
 :
MPI_Init(NULL, NULL);  //MPI starts
MPI_Comm_size(MPI_Comm_World, &comm_sz);  //get #process
MPI_Comm_rank(MPI_Comm_World, &my_rank);  //get my rank
   mpi operations
#pragma omp parallel for num_threads(4) ···
   omp operations
 :
   mpi operations.
 :
```