Introduction to Message Passing Interface (MPI)

Message Passing Interface

- API for distributed memory programming
  - Now de-fact standard in HPC/parallel computing
  - Explicitly write data communication
    - Message Passing
  - What is a message:
    - “data” and additional information
  - We will use MPI to parallelize codes in our exercise problems

MPI APIs

- Initialize MPI enviroment
- Query process information
- 1 to 1 communication
  - Synchronous/Asynchronous
  - Blocking/Non-blocking
- Collective communication
  - Broadcast
  - Reduction
  - Synchronization

Initialize MPI

- Must call following two functions

```
#include "mpi.h“

int main(int argc, char *argv[] )
{
  MPI_Init(&argc, &argv);

  // application body

  MPI_Finalize();
  return 0;
}
```

Query process information

- Each process has own ID number

```
#include "mpi.h“

int main(int argc, char *argv[] )
{
  int np, id;
  MPI_Init(&argc, &argv);

  // Query the number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  // Query own ID
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  MPI_Finalize();
  return 0;
}
```

Used APIs

- MPI_Init(int *, char **)
  - Initialize MPI environment
- MPI_Finalize()
  - Finish MPI environment
- MPI_COMM_WORLD (MPI_Comm struct)
  - It is called a communicator
  - We use the communicator to access process information
  - This is a default communicator
- MPI_Comm_size(MPI_Comm, int *)
  - Query the number of processes in a given communicator
- MPI_Comm_rank(MPI_Comm, int *)
  - Query own ID

Hello World example

- Compile
  - mpicc hello.c –o hello
- Execution
  - mpirun -np ### executable
- Example with 4 processes

```
std4dc1[~] mpicc hello.c -o hello
std4dc1[~] mpirun -np 4 ./hello
Hello world from process 2 (4)
Hello world from process 1 (4)
Hello world from process 0 (4)
Hello world from process 3 (4)
std4dc1[~]
```

Single Program Multiple Data

- MPI is a type of SPMD parallel programing
  - Source code is a single file.
  - An Execution file is also single.
    - The Program working on multiple data
  - We launch multiple processes running the same program in different memory space.
  - We need “data communication” between processes

SPMD example (1)

```
int np, id;
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
...

switch(id) {
  case 0:
    do something assigned id = 0
    break
  case 1:
    do something assigned id = 1
    break
  case 2:
    do something assigned id = 2
    break
}
```

SPMD example (2)

- Ex. 1
  - Calculation of pi

$$\int_0^1 \frac{4\,dx}{1+x^2}$$

```
h = 1.0/(double)n;
sum = 0.0;

// main loop
for (i = 0; i < n; i++) {
  x = h*((double)i - 0.5);
  sum += f(x);
}
mypi = h*sum;
```

  - We have “n” evaluations of f(x).
  - We have summation of the partial sum.
- How do we parallelize this?

SPMD example (3)

- We divide the loop into “np” parts.
  - Serial: i = 0 to n-1
  - Parallel: i = s[id] to e[id]

```
// main loop
for (i = s[id]; i < e[id]; i++) {
  x = h*((double)i - 0.5);
  sum += f(x);
}
```

- n = 1000, np = 2

  s[0] = 0,   e[0] = 500

  s[1] = 500 e[1] = 1000

- n = 1000, np = 3

  s[0] = 0,   e[0] = 333

  s[1] = 333 e[1] = 666

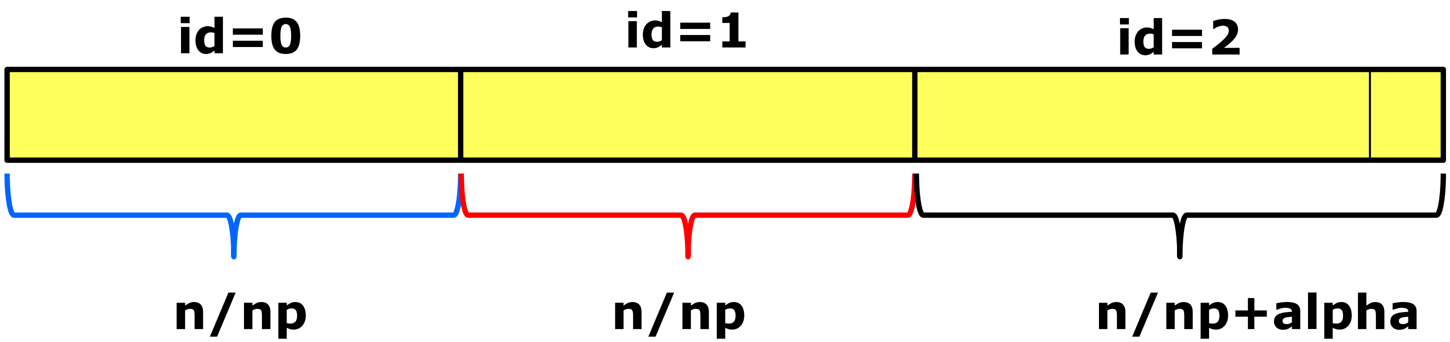
  s[2] = 666 e[1] = 1000 (treatment of last index)

SPMD example (4)

- A SPMD way solution

```
int s[128], e[128];

...
s[id] = (n/np)*id;
e[id] = (n/np)*(id+1) + (id-1 == np ? n % np : 0);
```



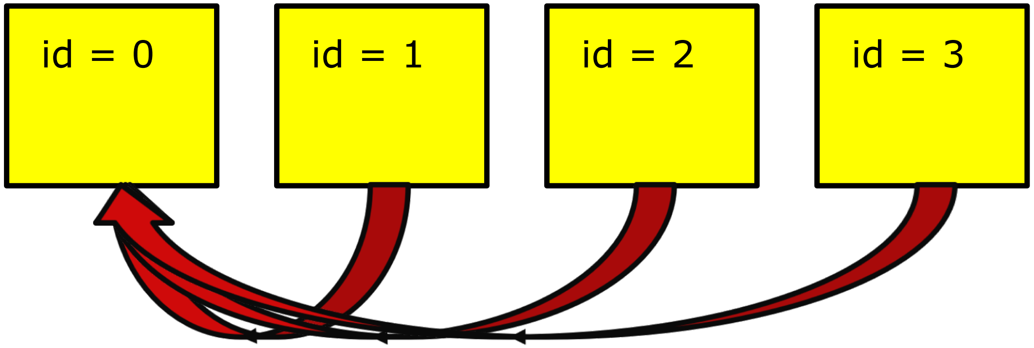
- We can calculate each partial sum in parallel
- How about the total sum?
  - We use communication!!

Measure elapsed time in MPI

- double MPI_Wtime(void)
  - Time measurement function in MPI.
  - Return value is the elapsed in seconds.

```
  double time_start, time_end;
  ...
  time_start = MPI_Wtime();
  for (i = s[id]; i < e[id]; i++) {
    x = h*((double)i - 0.5);
    sum += f(x);
  }
  time_end = MPI_Wtime();
// time_end-time_start is the elapsed time for the loop
```

Summation

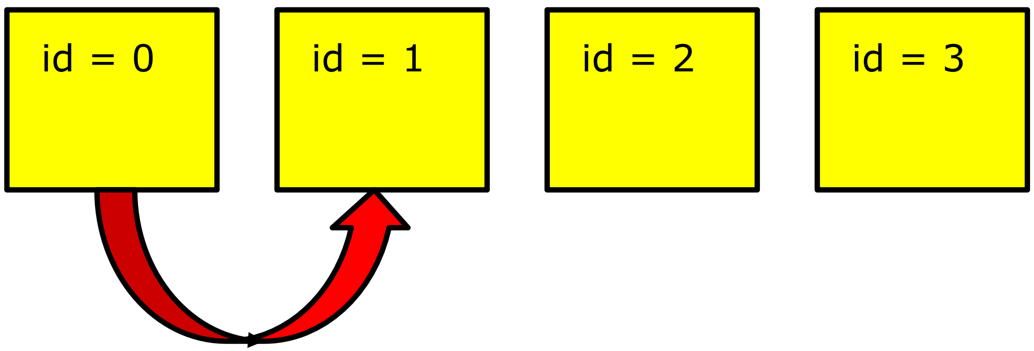


- Processes id = 1 to n-1 send own partial sum to process id=0 (main process)
- The main process calculate the total sum

```
if (id != 0) {
  send own partial sum to the main process
} else {
  receive partial sums from other processes
  calculate the total sum
}
```

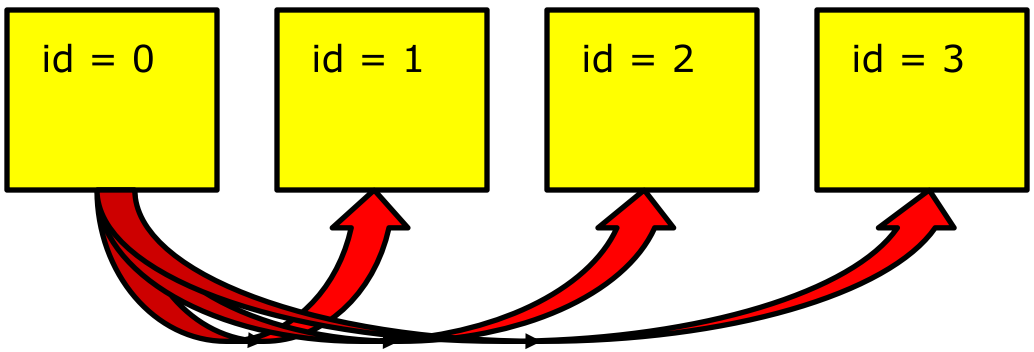
1 to 1 Communication

- We use ID in communication
- In 1 to 1 communication, we need to match send and receive operations
  - Sender :sends data to process (id=0)
  - Receiver :receives data from process (id=1)

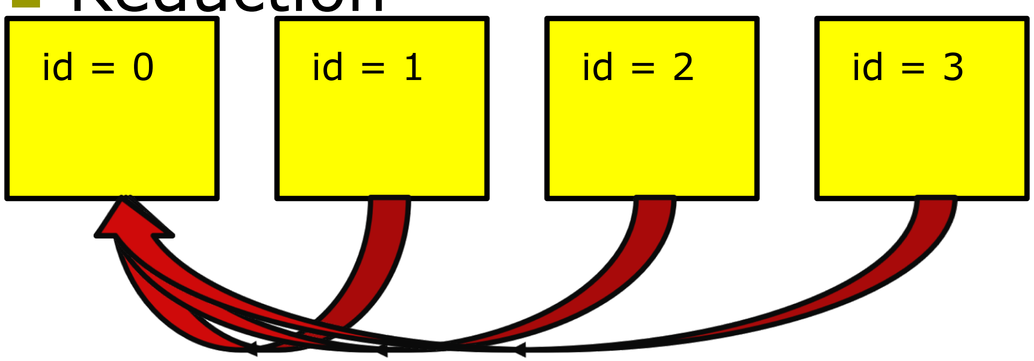


Collective communications

- Collective communications
  - Broadcast

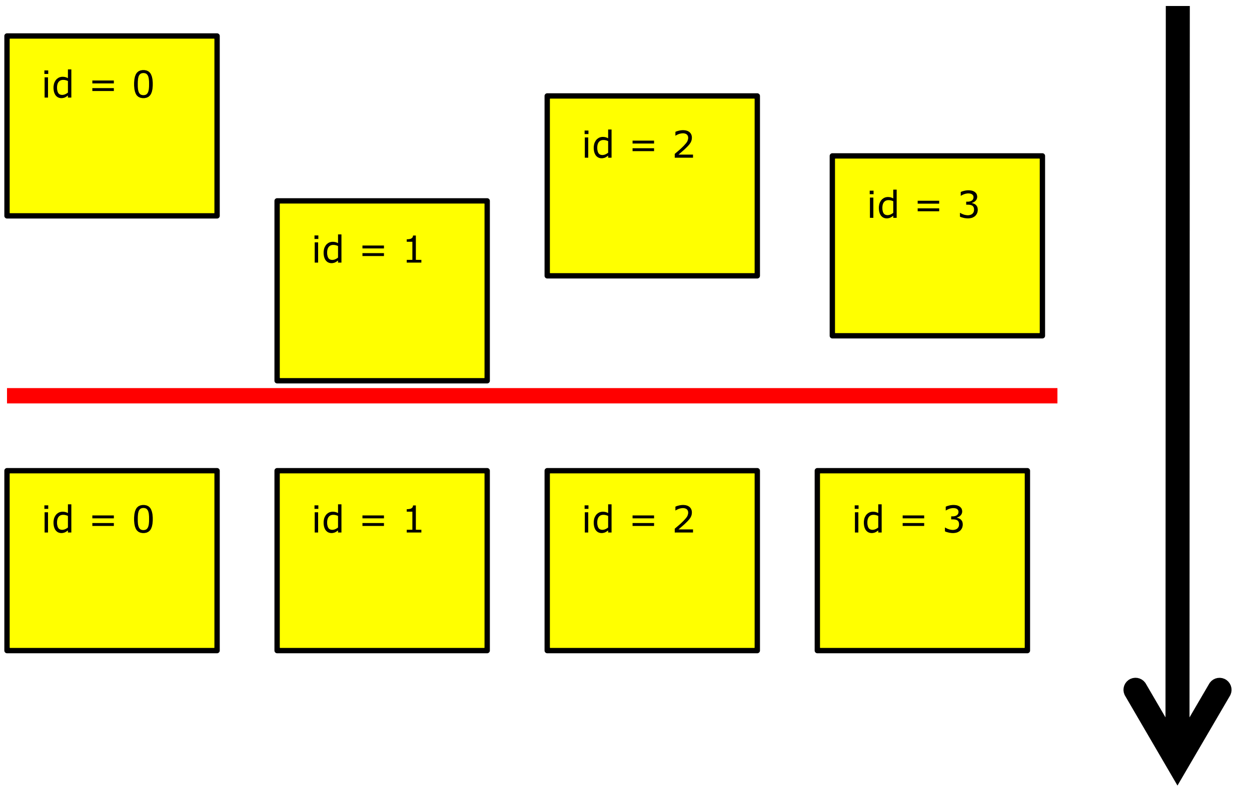


  - Reduction



Synchronization

- Synchronization



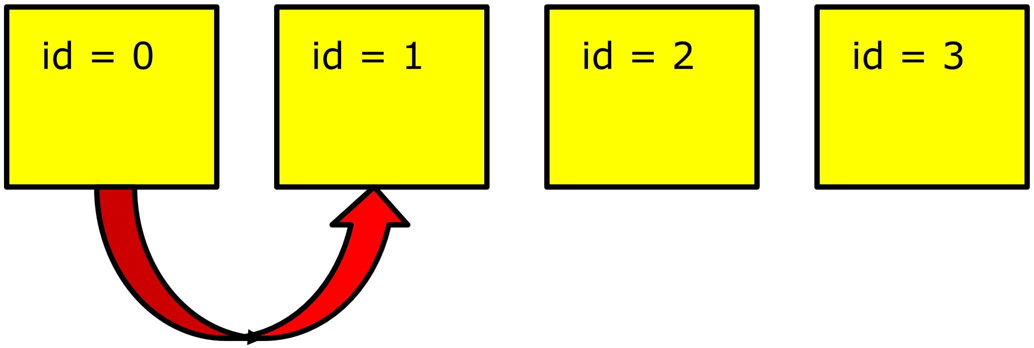
1 to 1 Communication details(1)

- Send : MPI_Send()
- Receive : MPI_Recv()

```
MPI_Status status;
int data;

if (id == 0) {
  data = 3;
  MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}

if (id == 1) {
  MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```



1 to 1 Communication details(2)

- **MPI_Send:**

```
MPI_Send(void *data, int count, MPI_Datatype type,
         int dest, int tag, MPI_Comm comm)
```

C/C++

- **data:** memory location containing data
- **count:** number of elements of type “type” to send
- **type:** MPI type of each element
- **dest:** destination rank for the data
- **tag:** identification of message
- **comm:** communicator

1 to 1 Communication details(3)

- **MPI_Recv:**

```
MPI_Recv(void *data, int count, MPI_Datatype type,
         int source, int tag, MPI_Comm comm, MPI_Status * status)
```

C/C++

- data: memory location intended for data
- count: number of elements of type “type” to send
- type: MPI type of each element
- source: rank to receive data from
  Special rank: MPI_ANY_SOURCE
- tag: identification of message to receive
  Special tag: MPI_ANY_TAG
- comm: communicator
- status: status of the return
  Special status: MPI_STATUS_IGNORE