# Parallel Programming

Distributed Memory Programming
with MPI (1)

Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.

A distributed memory system

3
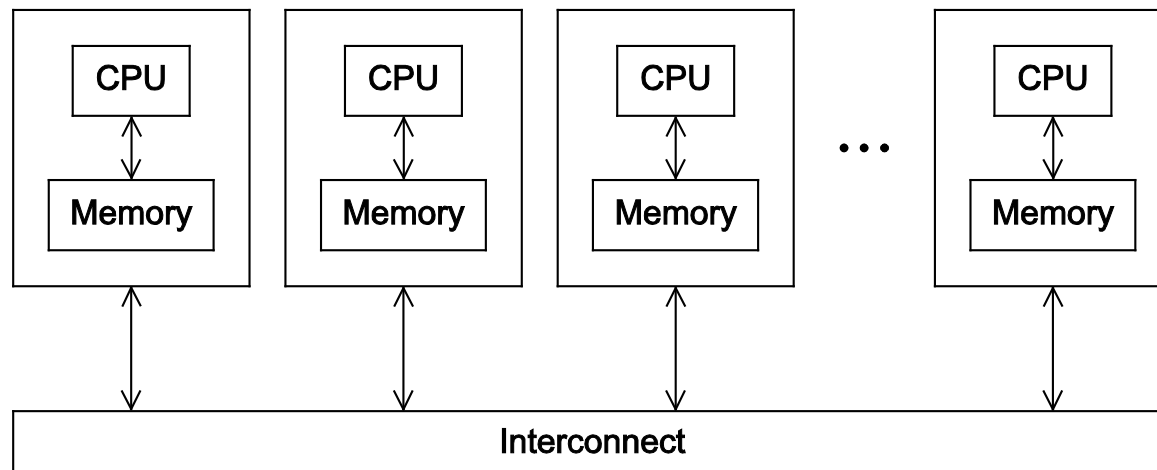
Hello World!

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.

- p processes are numbered $0, 1, 2, .. p-1$

Our first MPI program

```c
1  #include <stdio.h>
2  #include <string.h>   /* For strlen           */
3  #include <mpi.h>       /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8      char       greeting[MAX_STRING];
9      int        comm_sz;   /* Number of processes */
10     int        my_rank;   /* My process rank      */
11
12     MPI_Init(NULL, NULL);
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16     if (my_rank != 0) {
17         sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19         MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21     } else {
22         printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23         for (int q = 1; q < comm_sz; q++) {
24             MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26             printf("%s\n", greeting);
27         }
28     }
29
30     MPI_Finalize();
31     return 0;
32  }   /* main */
```

Compilation

wrapper script to compile

source file

mpicc -g -Wall -o mpi_hello mpi_hello.c

produce debugging information

create this executable file name (as opposed to default a.out)

turns on all warnings

Execution

mpiexec -n <number of processes>   <executable>

---

mpiexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes

Execution

```
mpiexec  -n  1  ./mpi_hello

        Greetings from process 0 of 1 !
```

```
mpiexec  -n  4  ./mpi_hello

        Greetings from process 0 of 4 !

        Greetings from process 1 of 4 !

        Greetings from process 2 of 4 !

        Greetings from process 3 of 4 !
```

MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with "MPI_".
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.

MPI Components

- ## MPI_Init
  - Tells MPI to do all the necessary setup.

```
int MPI_Init(
      int*      argc_p   /* in/out */,
      char***   argv_p   /* in/out */);
```

- ## MPI_Finalize
  - Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

Basic Outline

```
.  .  .
#include <mpi.h>
.  .  .
int main(int argc, char* argv[]) {
    .  .  .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    .  .  .
    MPI_Finalize();
    /* No MPI calls after this */
    .  .  .
    return 0;
}
```

Communicators

- A collection of processes that can send messages to each other.

- MPI_Init defines a communicator that consists of all the processes created when the program is started.

- Called MPI_COMM_WORLD.

13

Communicators

```
int MPI_Comm_size(
    MPI_Comm    comm         /* in   */,
    int*        comm_sz_p    /* out  */);
```

number of processes in the communicator

```
int MPI_Comm_rank(
    MPI_Comm    comm         /* in   */,
    int*        my_rank_p    /* out  */);
```

my rank
(the process making this call)

SPMD

- Single-Program Multiple-Data

- We compile <u>one</u> program.

- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.

- The if-else construct makes our program SPMD.

Communication

```
int MPI_Send(
    void*          msg_buf_p       /* in */,
    int            msg_size        /* in */,
    MPI_Datatype   msg_type        /* in */,
    int            dest            /* in */,
    int            tag             /* in */,
    MPI_Comm       communicator    /* in */);
```

Data types

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Communication

```
int MPI_Recv(
    void*           msg_buf_p       /* out */,
    int             buf_size        /* in  */,
    MPI_Datatype    buf_type        /* in  */,
    int             source          /* in  */,
    int             tag             /* in  */,

    MPI_Comm        communicator    /* in  */,
    MPI_Status*     status_p        /* out */);
```

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
        send_comm);
```

r

MPI_Send

src = q

MPI_Recv

dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
        recv_comm, &status);
```

q

Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.

status_p argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

MPI_SOURCE
MPI_TAG
MPI_ERROR

MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG

How much data am I receiving?
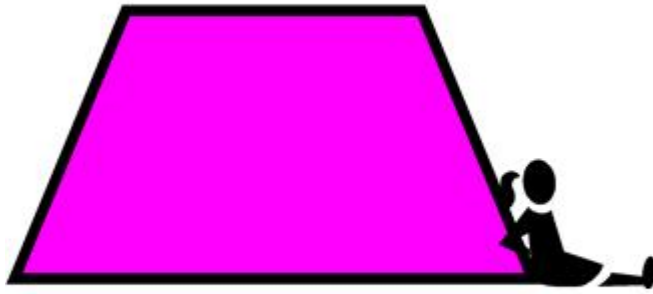
```
int MPI_Get_count(
        MPI_Status*   status_p   /* in  */,
        MPI_Datatype  type       /* in  */,
        int*          count_p    /* out */);
```
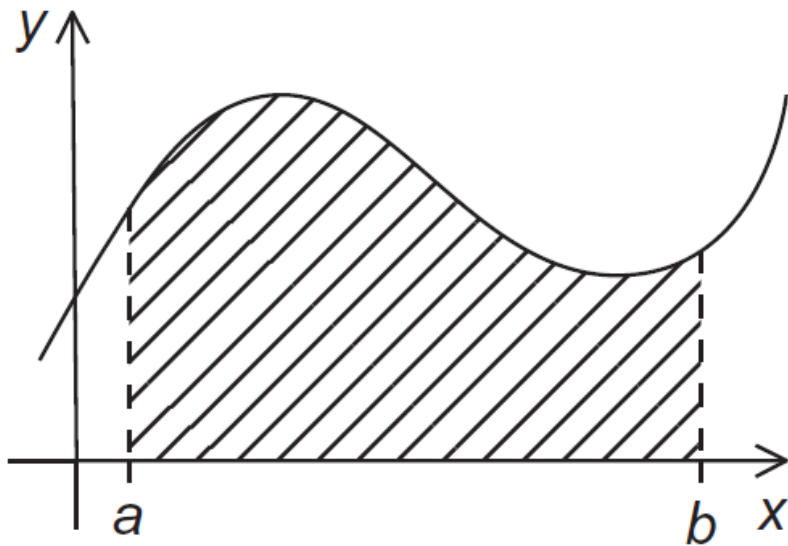
Issues with send and receive

- Exact behavior is determined by the MPI implementation.

- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.

- MPI_Recv always blocks until a matching message is received.

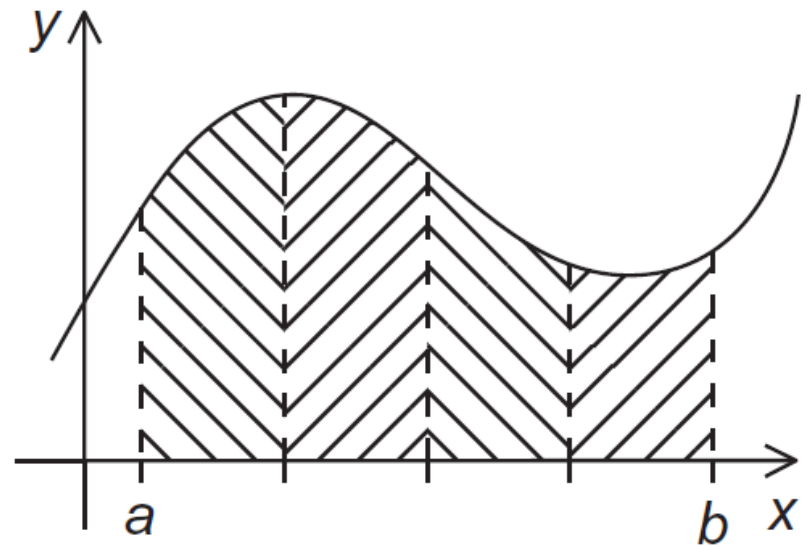- Know your implementation; don't make assumptions!

23

TRAPEZOIDAL RULE IN MPI
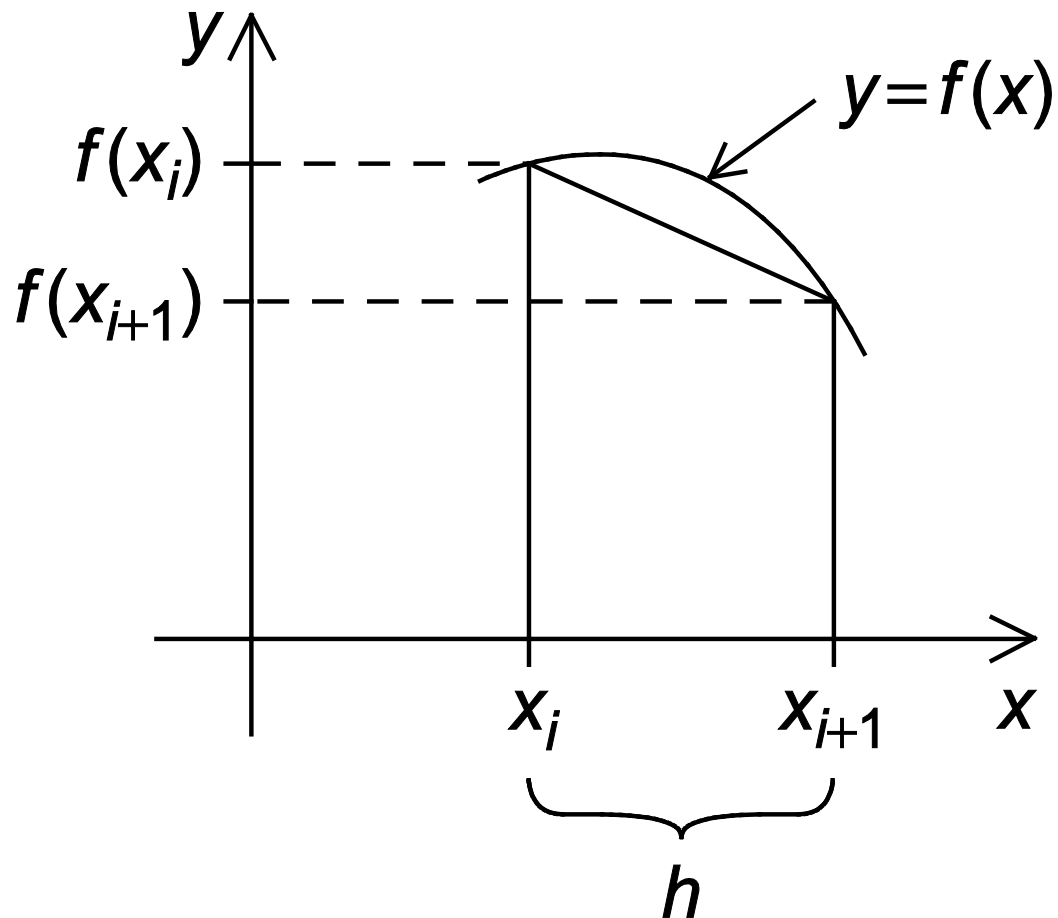
The Trapezoidal Rule

(a)

(b)

The Trapezoidal Rule

$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a, \; x_1 = a+h, \; x_2 = a+2h, \; \ldots, \; x_{n-1} = a+(n-1)h, \; x_n = b$$

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

One trapezoid

The figure shows axes labeled y and x. Points marked on the curve $y=f(x)$ at heights $f(x_i)$ and $f(x_{i+1})$, corresponding to x-positions x_i and x_{i+1}, with the spacing between them labeled h.

Pseudo-code for a serial program

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.

2. Identify communication channels between tasks.

3. Aggregate tasks into composite tasks.

4. Map composite tasks to cores.

Parallel pseudo-code

```
1     Get a, b, n;
2     h = (b−a)/n;
3     local_n = n/comm_sz;

4     local_a = a + my_rank*local_n*h;
5     local_b = local_a + local_n*h;
6     local_integral = Trap(local_a, local_b, local_n, h);
7     if (my_rank != 0)
8         Send local_integral to process 0;
9     else /* my_rank == 0 */
10        total_integral = local_integral;
11        for (proc = 1; proc < comm_sz; proc++) {
12            Receive local_integral from proc;
13            total_integral += local_integral;
14        }
15    }
16    if (my_rank == 0)
17        print result;
```

Tasks and communications for Trapezoidal Rule
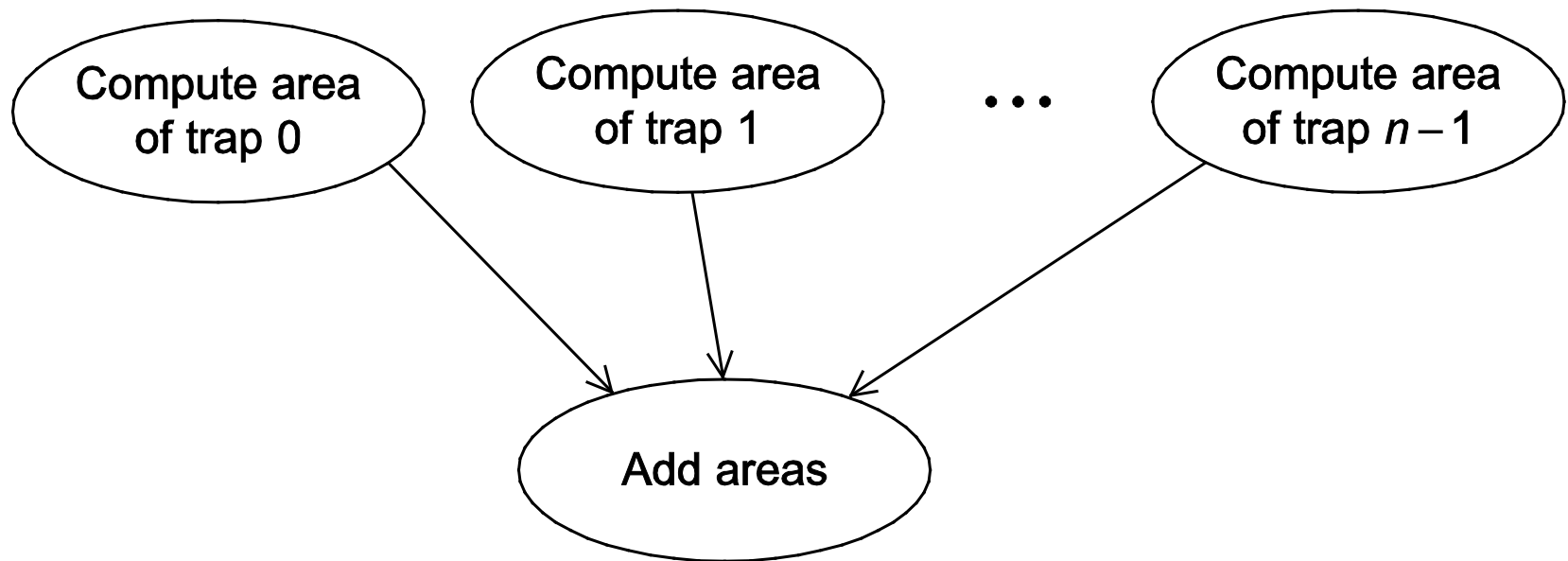
First version (1)

```
1  int main(void) {
2      int my_rank, comm_sz, n = 1024, local_n;
3      double a = 0.0, b = 3.0, h, local_a, local_b;
4      double local_int, total_int;
5      int source;
6
7      MPI_Init(NULL, NULL);
8      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11     h = (b-a)/n;              /* h is the same for all processes */
12     local_n = n/comm_sz;      /* So is the number of trapezoids  */
13
14     local_a = a + my_rank*local_n*h;
15     local_b = local_a + local_n*h;
16     local_int = Trap(local_a, local_b, local_n, h);
17
18     if (my_rank != 0) {
19         MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
```

First version (2)

```
21      } else {
22          total_int = local_int;
23          for (source = 1; source < comm_sz; source++) {
24              MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26              total_int += local_int;
27          }
28      }
29
30      if (my_rank == 0) {
31          printf("With n = %d trapezoids, our estimate\n", n);
32          printf("of the integral from %f to %f = %.15e\n",
33              a, b, total_int);
34      }
35      MPI_Finalize();
36      return 0;
37  } /*   main   */
```

First version (3)

```
 1  double Trap(
 2        double left_endpt   /* in */,
 3        double right_endpt  /* in */,
 4        int    trap_count   /* in */,
 5        double base_len     /* in */) {
 6     double estimate, x;
 7     int i;
 8
 9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10     for (i = 1; i <= trap_count -1; i++) {
11        x = left_endpt + i*base_len;
12        estimate += f(x);
13     }
14     estimate = estimate*base_len;
15
16     return estimate;
17  } /*  Trap  */
```

Dealing with I/O

```c
#include <stdio.h>
#include <mpi.h>

int main(void) {
    int my_rank, comm_sz;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Proc %d of %d > Does anyone have a toothpick?\n",
            my_rank, comm_sz);

    MPI_Finalize();
    return 0;
}  /* main */
```

Each process just prints a message.

Running with 6 processes

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

unpredictable output

Input

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.

- Process 0 must read the data (scanf) and send to the other processes.

```
.  .  .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b−a)/n;
.  .  .
```

Function for reading user input

```c
void Get_input(
        int       my_rank    /*  in   */,
        int       comm_sz    /*  in   */,
        double*   a_p        /*  out  */,
        double*   b_p        /*  out  */,
        int*      n_p        /*  out  */) {
    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    }
} /* Get_input */
```