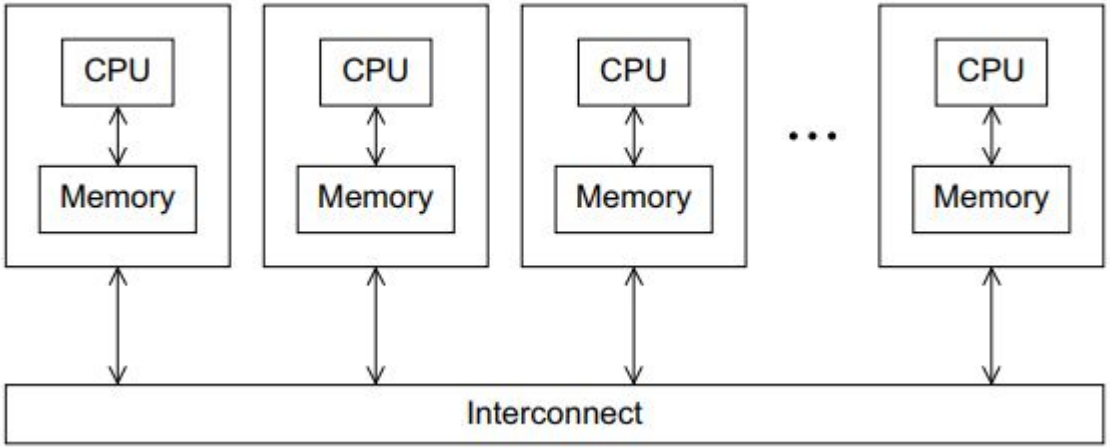
Parallel computing

MPI

Distributed memory

- Colección de **pares** CPU-Memory interconectados
- Como programar en estos sistemas enviando mensajes
  - **message-passing**



Message passing

- Un programa corriendo sobre un **par** es un **proceso**
- **Procesos** pueden **comunicar** entre ellos utilizando funciones
  - *send*
  - *receive*
- **MPI Message Passing Interface**
  - Libreria
  - funciones send/receive
  - funciones globales (**collective communication**) que involucran más de 2 procesos

Programar con MPI

- Queremos designar un proceso que imprima lo que recibe de los demás

```
#include <stdio.h>

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

   return 0;
}
```

Programar con MPI

- Compilar ***mpicc***
  - $ mpicc −g −Wall −o mpi_hello mpi_hello.c
  - wrapper C compiler
- Ejecutar
  - $ mpiexec −n <cuantos procesos> ./mpi_hello
  - $ 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!

Programar

```
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 */
```

Programar con MPI

- Todos las funciones empiezan con MPI_
- MPI_Init
  - inicializa la configuración de MPI
  - ninguna función de MPI puede llamarse antes del init

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

- MPI_Finalize(void)
  - Liberar los recursos utilizado para MPI

Programar con MPI

- Communicator
  - colección de procesos que pueden comunicar entre sí
  - MPI_COMM_WORLD
- Obtener información del communicator
  - 1er argumento -> communicator
  - 2do argumento -> el resultado de la función
    - Comm_size : cuantos procesos en el comm
    - Comm_rank : el rango del comm

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

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

Programar con MPI

- Un solo programa que realiza tareas distintas según el rango del proceso
  - Un proceso imprime mensajes
  - los otros procesos crean y mandan mensajes
  - SPMD (single program, multiple data)
    - if... else
- El programa puede ejecutarse sobre un cantidad cualquiera de procesos
  - se intenta que sea siempre el caso (escalabilidad)

Programar con MPI

- Communcation
  - los procesos otro que **0** crear mensajes sprintf y lo mandan al proceso **0**
  - el proceso 0 escribe en la consola los mensajes que recibe
    - Utiliza un for con comm_size-1 iteraciones

Programar con MPI - MPI_Send

- los primeros 3 argumentos determinan el contenido del mensaje
  - msg_buf_p : puntero hacia el mensaje
  - msg_size, msg_type : cantidad de datos para mandar
    - msg_type : strlen(greeting)+1 (‘\0’)
    - msg_type : MPI_CHAR (MPI_Datatype)

```
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 */);
```

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	

Programar con MPI - MPI_Send

- los 3 siguientes determinan el destinatario
  - dest : rango del destinatario
  - tag : entero positivo para mandar informacion complementaria
  - communicator: se puede tener varios communicators

```
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 */);
```

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	

Programar con MPI - MPI_Recv

```
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 */);
```

- Los primeros 3 argumentos representan la memoria disponible para recibir el msg
  - msg_buf_p : puntero hacia los datos asignados
  - buf_size : tamaño del mensaje
  - buf_type : el tipo de datos del mensaje
- Los tres siguientes representan el msg
  - source : el proceso que mandó el msg
  - tag : tiene que corresponder al tag del msg enviado
  - communicator : mismo communicator que el send
  - status_p : ? MPI_STATUS_IGNORE

Programar con MPI - Message matching

- proceso **q** llama
  - MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);
- proceso **r** recibe
  - MPI_Recv(rec_buf_p, recv _buf sz, recv_type, src, recv_tag, recv_comm, &status);
- Para que **r** reciba el mensaje de **q**
  - recv_comm==send_comm
  - recv_tag==send_tag
  - dest==r
  - src==q
  - También debe de haber un compatibilidad entre msg enviado y msg esperado
    - recv_type==send_type
    - recv_buf_sz >= send_buf_sz

Programar con MPI - Message matching

- Entonces si mis procesos que mandan mensajes terminen en momentos distintos
  - El proceso 0 recibirá los mensajes en orden de todas formas
  - Significa que si proceso comm_sz-1 termina primero tendrá que esperar a los otros procesos que terminen
- Solución ? Solamente para MPI_Recv
  - MPI_ANY_SOURCE

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

  - MPI_ANY_TAG
  - No existe algo similar para el comunicator

Programar con MPI - status_p

- Un proceso puede recibir datos que desconoce
  - quien lo mandó
  - cuantos datos se mandaron
  - el tag
- MPI_Status
  - estructura de datos
  - MPI_SOURCE
  - MPI_TAG
  - MPI_ERROR
  - ej. MPI_Status status; status.MPI_SOURCE; status.MPI_TAG;
  - Cuantos datos ?
    - MPI_Get_count(&status, recv_type, &count)
    - no puede ser directamente accesible por status porque depende del tipo de datos

Programar con MPI - Send y Receive

- Que pasa cuando se manda un mensaje
  - El proceso que manda agrega la envoltura
    - rank, tag, etc
  - El proceso puede **buffer**
    - agrega la envoltura y se termina el send
  - El proceso puede **block**
    - agrega la envoltura y espera hasta poder empezar el envío
  - Si necesitamos saber que el mensaje fue transmitido exitosamente, MPI provee alternativas al send
- block o buffer ?
  - depende del tamaño de los datos si es superior > max entonces **block** sino **buffer**

Programar con MPI - Send y Receive

- Que pasa cuando se recibe un mensaje ?
  - al contrario del MPI_Send, MPI_Recv siempre bloquea hasta recibir un mensaje
- Si un proceso q manda dos mensajes a un proceso r
  - el primer mensaje siempre llegara antes del segundo
  - esto no está asegurado cuando los dos mensajes vienen de un proceso diferente.

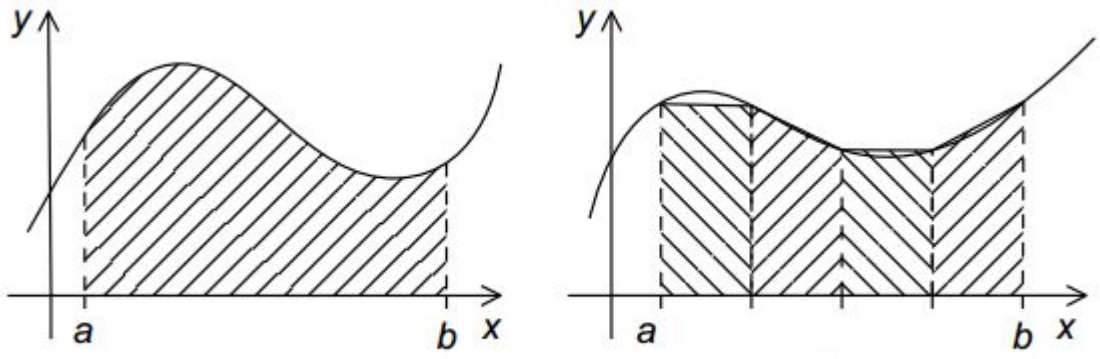
Programar con MPI - Advertencia

- Un proceso llama a MPI_Recv
  - no recibe ningún mensaje entonces se queda esperando
- Tener cuidado del matching
- MPI_Send en modo block también puede bloquear por un problema de matching
  - Si está en modo buffer, el mensaje será perdido

Ejemplo: la regla de trapecio

- y=f(x), a < b queremos integrar esta funcion con la regla de trapecio
- Subdividir el espacio entre a y b en n partes
- el ***h = (b − a)/n***, ***xi = a + ih***, ***i = 0, 1,...,n***

$$T = (b-a)\frac{f(a)+f(b)}{2}.$$



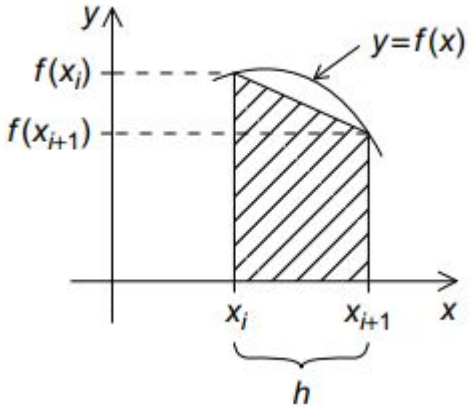
Ejemplo: la regla de trapecio

- el $h = (b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$
- Approx : $h[\, f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$.

Ejemplo: la regla de trapecio

- el ***h = (b − a)/n***, ***xi = a + ih***, ***i = 0, 1,...,n***
- Approx : **h[f(x0)/2 + f(x1) + f(x2) + ⋯ + f(xn−1) + f(xn)/2].**

```
/* 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;
```



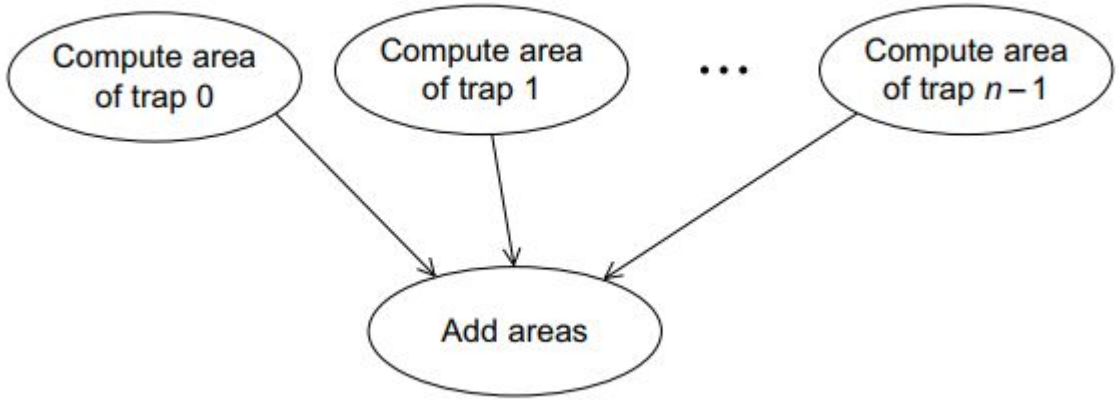
Ejemplo: la regla de trapecio

- Determinar dos tipos de **tareas**
  - Calcular el área de un trapecio
  - Calcular la suma de las áreas
- Repartir las tareas ?
  - n/comm_sz

```
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;
```

Ejemplo: la regla de trapecio

- Comunicación entre procesos



Ejemplo: la regla de trapecio

```
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);
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  */
```

Ejemplo: la regla de trapecio

```
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);
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  */
```

Input/Output

- Output
  - autorizado a todo los procesos, non-deterministico
- Input generalmente autorizado solamente al proceso 0 de los communicators
  - proceso 0 pide un dato y lo manda al proceso correspondiente
  - Ej. pedir a,b,n para la regla del trapecio y mandar esto a todos los procesos

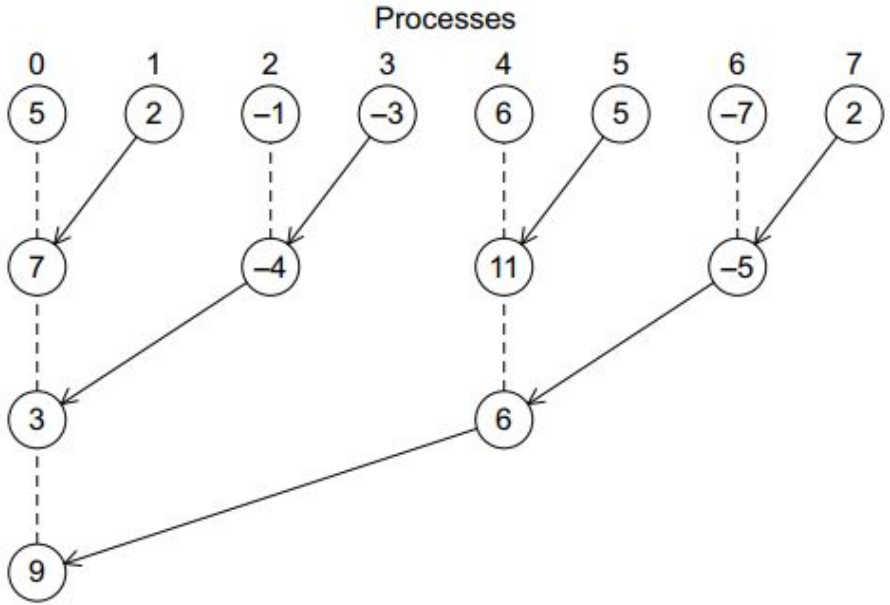
```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

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

h = (b-a)/n;
. . .
```

Input/Output - Get_input example

```
1  void Get_input(
2        int      my_rank   /* in  */,
3        int      comm_sz   /* in  */,
4        double*  a_p       /* out */,
5        double*  b_p       /* out */,
6        int*     n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

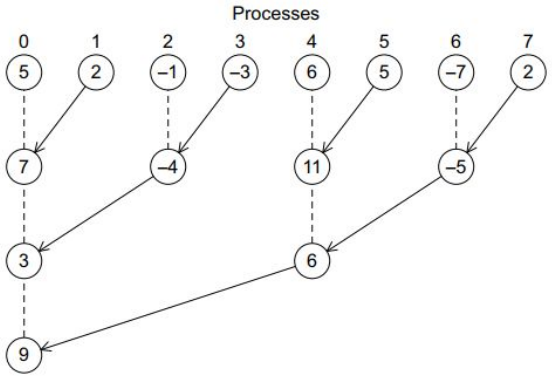
Comunicación

- Problema de suma global de la regla del trapecio !
  - el proceso 0 termina haciendo toda la suma

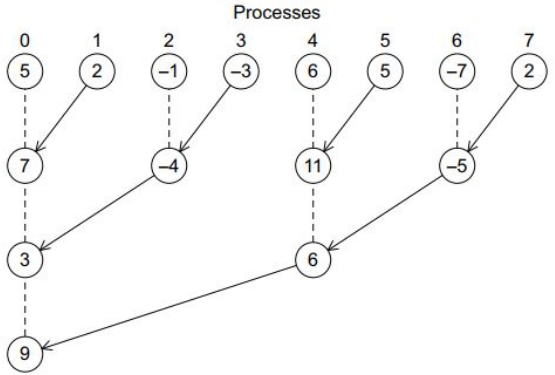


Comunicación

- Comunicación estructurada
  - la estructura original era **comm_sz-1 recepciónes**, y **comm_sz-1 sumas** por proceso 0
  - ahora solamente necesita **3**
    - **procesos paralelos (0,2,4,6)**
    - **50 % de ganancia**
  - **Si tendriamos 1024 procesos**
    - **proceso 0 solamente haria 10 suma**
    - **99% de ganancia**



Comunicación



- Comunicación estructurada
  - Como haremos esto ?
- **MPI_Reduce**
  - **Evitar una mala implementación**
  - **collective communicación (vs point-to-point como MPI_Recv/Send)**
    - función que involucra todos los procesos
  - Flexible (suma,min, max, etc)

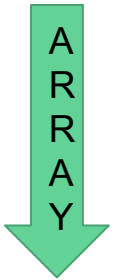
```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

Comunicación

- **MPI_Reduce**
  - Operator
    - MPI_Op

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```



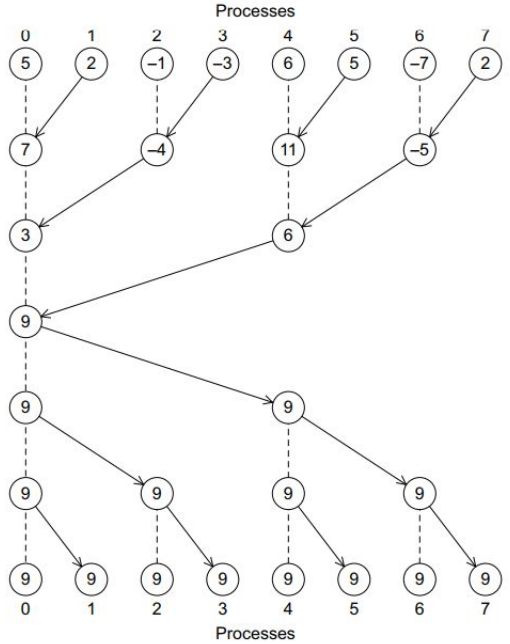

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Comunicación

```
int MPI_Allreduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      MPI_Comm      comm           /* in  */);
```

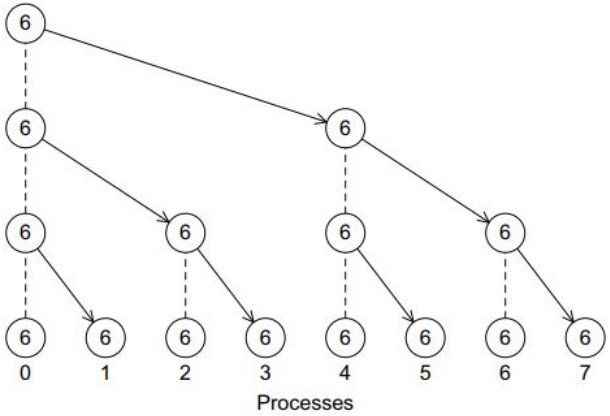
- MPI_Allreduce (comunición en mariposa)
  - Permite distribuir el resultado a todos los procesos
    - seguir calculando
  - No tiene dest_process ya que todos recibirán el resultado



Broadcast

- Comunicación colectiva que envía un dato a todos
  - todos los procesos llaman al broadcast
  - no hay destinatario

```
int MPI_Bcast(
        void*         data_p          /* in/out */,
        int           count           /* in     */,
        MPI_Datatype  datatype        /* in     */,
        int           source_proc     /* in     */,
        MPI_Comm      comm            /* in     */);
```

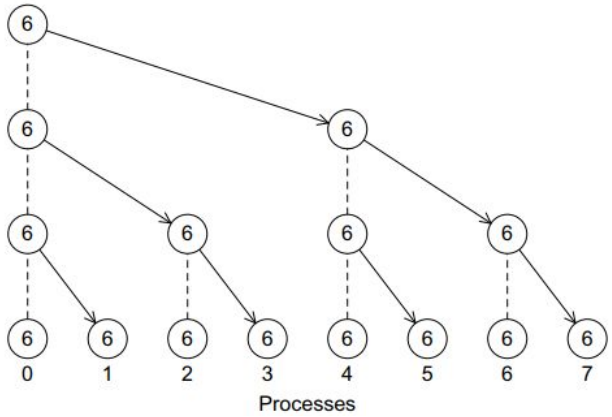


Broadcast

```
int MPI_Bcast(
      void*         data_p          /* in/out */,
      int           count           /* in     */,
      MPI_Datatype  datatype        /* in     */,
      int           source_proc     /* in     */,
      MPI_Comm      comm            /* in     */);
```

- Comunicación colectiva que envía un dato a todos
  - todos los procesos llaman al broadcast
  - no hay destinatario

```
1  void Get_input(
2        int      my_rank  /* in  */,
3        int      comm_sz  /* in  */,
4        double*  a_p      /* out */,
5        double*  b_p      /* out */,
6        int*     n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```



Distribución de datos

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

- Suma de vectores
  - En secuencial es muy simple
- Como hacerlo con MPI ?
  - Cada proceso hara un cierta cantidad de sumas
    - local_n=n/comm_sz
  - Enviar un bloque de local_n a cada proceso
    - **block partition**
    - **cyclic partition**
    - **block-cyclic**

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Process	Components											
	Block				Cyclic				Block-Cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Distribución de datos

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

- Como hacerlo con MPI ?
  - Cada proceso hará un cierta cantidad de sumas

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

```
1  void Parallel_vector_sum(
2        double  local_x[]  /* in  */,
3        double  local_y[]  /* in  */,
4        double  local_z[]  /* out */,
5        int     local_n    /* in  */) {
6     int local_i;
7
8     for (local_i = 0; local_i < local_n; local_i++)
9        local_z[local_i] = local_x[local_i] + local_y[local_i];
10 }  /* Parallel_vector_sum */
```

Dispersion de datos

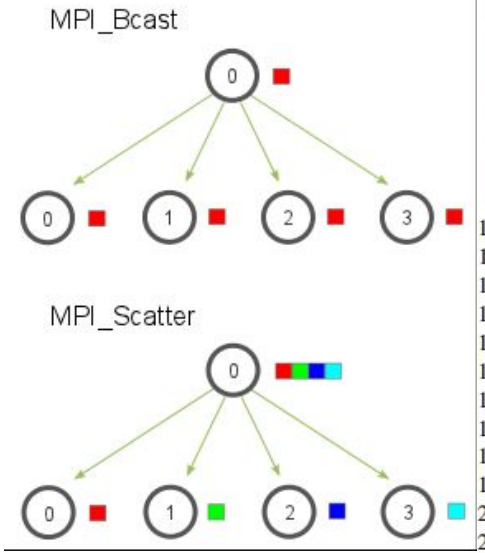
- Tenemos un vector de 10k datos
  - cada proceso necesita datos de este vector
  - mandamos a cada proceso el vector
    - cada proceso asignará su memoria para 10k mientras utiliza 1k.
- Scatter (comunicación colectiva)

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

Dispersion de datos

- Scatter (comunicación colectiva)

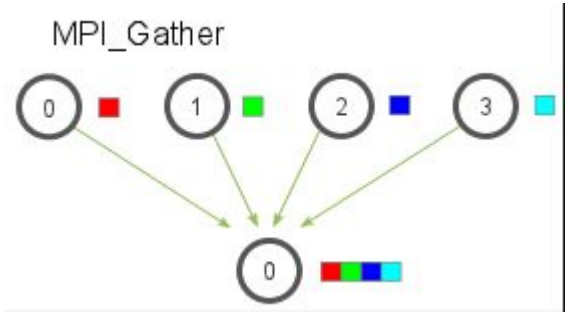
```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```




```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

Reunión de datos

- Una vez el cálculo de la suma de vectores está hecha por todos los procesos
  - reunir estos vectores
- MPI_Gather




```
int MPI_Gather(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           dest_proc   /* in  */,
      MPI_Comm      comm        /* in  */);
```

Reunión de datos

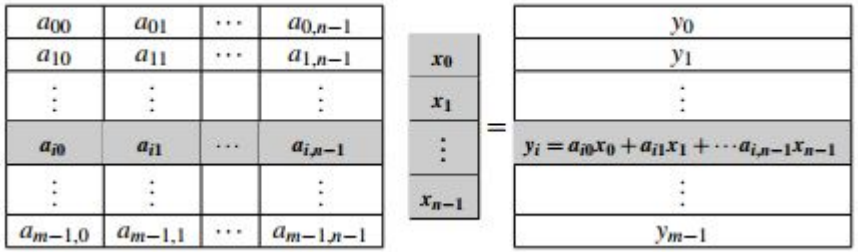
- MPI_Gather

```
int MPI_Gather(
      void*         send_buf_p   /* in  */,
      int           send_count   /* in  */,
      MPI_Datatype  send_type    /* in  */,
      void*         recv_buf_p   /* out */,
      int           recv_count   /* in  */,
      MPI_Datatype  recv_type    /* in  */,
      int           dest_proc    /* in  */,
      MPI_Comm      comm         /* in  */);
```

```
1  void Print_vector(
2        double    local_b[]  /* in */
3        int       local_n    /* in */,
4        int       n          /* in */,
5        char      title[]    /* in */,
6        int       my_rank    /* in */,
7        MPI_Comm  comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 }  /* Print_vector */
```

Reunión de datos

- Ejemplo multiplicación matriz por un vector




```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

```
y[i] += A[i*n+j]*x[j];
```

Matriz almacenada en 1D

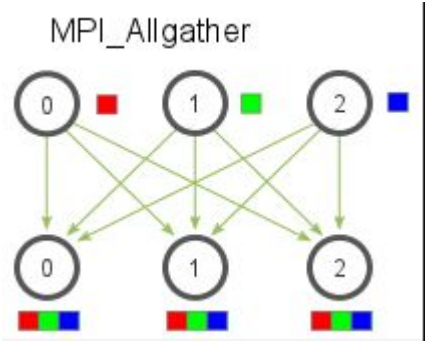
Reunión de datos

- Ejemplo multiplicación matriz por un vector

```
1  void Mat_vect_mult(
2        double  A[]  /* in  */,
3        double  x[]  /* in  */,
4        double  y[]  /* out */,
5        int     m    /* in  */,
6        int     n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

Reunión de datos

- Ejemplo multiplicación matriz por un vector
  - Una tarea individual será la multiplicación de un elemento de A con X y su adición en Y.
  - Proceso *q* hará el proceso par Y[i] entonces este proceso necesita la fila *i* de A
  - Utilizar una distribución en bloque de A en filas
  - X sera enviado cada proceso
- A veces Y es utilizado como X en una siguiente iteración
  - MPI_Gather + MPI_Bcast
  - Solución MPI_Allgather




```
int MPI_Allgather(
      void*          send_buf_p  /* in */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

Reunión de datos

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```