DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

1. Motivación y aspectos de la programación paralela.
2. Tipos de sistemas paralelos. Paradigmas de programación paralela.
3. Conceptos básicos y medidas de paralelismo.
4. Diseño de programas paralelos.
5. La Interface de paso de mensaje: el estándar MPI.
   - Programación mediante paso de mensajes.
   - Introducción al estándar MPI (Message Passing Interface).
     - Evolución y origen.
     - ¿Qué es MPI?
     - Funciones básicas.
       - Inicialización.
       - Finalización.
       - Información.
       - Comunicación.
     - Comunicaciones colectivas.
     - Problemas de interbloqueo.
     - Mediciones de tiempo.
     - Caso de estudio. Cálculo de pi.
6. Paralelización de algoritmos: ejemplos y aplicaciones.

Evolución y origen

- Generaciones previas de computadores paralelos comerciales se han basado en la arquitectura de memoria distribuida, con lo que el paradigma natural es el modelo de paso de mensajes.
- Esto dio lugar al desarrollo de diferentes entornos de trabajo para paso de mensajes.
- Cada fabricante proporcionaba sus propias librerías con buenos rendimientos en sus máquinas pero incompatibles con las proporcionadas por otros fabricantes: con diferencias no sólo sintácticas sino también semánticas:
  - NX: librería propietario de Intel.
  - MPL: librería de paso de mensajes de IBM.
- Desarrollo de paquetes de dominio público: PICL, PARMACS, P4, PVM, …
- PVM: quizá la primera librería ampliamente adoptada y aceptada como librería de paso de mensajes.
- Esfuerzo por crear un estándar para la programación de paso de mensajes: MPI.

DSAP

Evolución y origen

- La primera versión del estándar MPI se finalizó y publicó en 1994, después de dos años de reuniones y discusiones.
- En 1995 se publicó una nueva especificación del estándar en la que se modificaron aspectos de orden menor de la primera versión.
- En 1997 se presentó la especificación MPI-2 que extiende MPI sin introducir cambios en la versión 1:
  - Se introdujeron características para incrementar la conveniencia y robustez de MPI, como operaciones en memoria remota, entrada/salida paralela, gestión dinámica de procesos, etc.

Todo ello ha permitido que MPI sea la primera librería estándar portable con buenos rendimientos.

DSAP

¿Qué es MPI?

- Message Passing Interface: Librería de funciones para el paso de mensajes entre procesadores.
- MPI NO ES un lenguaje!
- Puede ser utilizado en computadores paralelos o redes de computadores personales/estaciones de trabajo,…
- Aplicable desde Fortran o C.
- Todos los procesadores reciben la misma copia del programa a ejecutar.
- Requiere sus propios comandos de compilación / ejecución.

¿Qué es MPI?

MPI ofrece:

- **Estandarización** a muchos niveles, reemplazando virtualmente a todas las implementaciones de paso de mensajes utilizadas para producción.
- **Portabilidad** a los sistemas existentes y a los nuevos. La mayoría de las plataformas (si no todas) ofrecen, al menos, una implementación de MPI.
- **Rendimiento** comparable a las librerías propietarias de los vendedores. Incluso para arquitecturas de memoria compartida, las implementaciones MPI podrían no hacer uso de la red de interconexión, sino de la memoria compartida.
- **Riqueza.** Posee una extensa funcionalidad y muchas implementaciones de calidad.

Funciones básicas

- MPI incluye más de 125 funciones.
- Sin embargo, se puede efectuar trabajo productivo en paralelo con sólo 6.

Variables MPI predefinidas

- MPI predefine una serie de variables y de estructuras de datos inherentes a su funcionamiento.
- Se encuentran en un archivo de encabezado que debe ser incluido en todo código que use MPI.

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

Funciones básicas

- Necesitamos:
  - un método para crear procesos: estático / dinámico.
  - un método para enviar y recibir mensajes, punto a punto y de manera global.
- El objetivo de MPI es explicitar la comunicación entre procesos, es decir:
  - el movimiento de datos entre procesadores.
  - la sincronización de procesos.

Funciones básicas

- El modelo de paralelismo que implementa MPI es **SPMD** (Single Program Multiple Data).

```
if (pid==1)              ENVIAR_a_pid2
else if (pid==2)         RECIBIR_de_pid1
```

Recordamos que cada proceso dispone de su propio espacio de direcciones.

- También se puede trabajar con un modelo **MPMD** (Multiple Program Multiple Data): se ejecutan programas diferentes en los nodos.

Funciones básicas

- En todo caso, hay que tener en cuenta que la **eficiencia en la comunicación** va a ser determinante en el **rendimiento del sistema** paralelo, sobre todo en aquellas aplicaciones en las que la comunicación juega un papel importante (paralelismo de grano medio / fino).

- Además de implementaciones específicas, existen dos implementaciones libres de uso muy extendido: LAM/MPI (OpenMPI) y MPICH.

Nosotros vamos a usar **MPICH2**.

Funciones básicas: Inicialización

int MPI_Init(*argc, **argv)

- int **argc:** Puntero al número de argumentos.
- char **argv:** Puntero al vector de argumentos.

- Inicializa varias estructuras de datos inherentes al ambiente de trabajo MPI.
- Si el ambiente no se puede inicializar, el programa se detiene por completo.
- Debe ser invocada por todos los procesos antes de cualquier otra.
- El estándar MPI no indica nada sobre lo que un programa puede hacer antes de la inicialización de MPI. Sin embargo, en la implementación MPICH se debe hacer lo menos posible (evitar sobretodo: abrir archivos, lectura y escritura).

Funciones básicas: Inicialización

Todas las funciones MPI
(excepto MPI_Wtime y MPI_Wtick)
devuelven un valor de error.

Por defecto, el manipulador de errores de MPI aborta un trabajo cuando se produce un error.

Si no ha ocurrido ningún error, se devuelve **MPI_SUCCESS**.

Funciones básicas: Finalización

int MPI_Finalize(void)

- Cierra el ambiente de trabajo en paralelo una vez finalizado el trabajo.
- Debe ser llamada por todo proceso antes de salir.
- No se puede invocar a otras funciones MPI después de esta llamada.

Funciones básicas: Información

int MPI_Comm_size(comm, *size)

- MPI_Comm **comm:** variable de tipo MPI_Comm suministrado por el usuario indicando el comunicador utilizado. Por defecto, es MPI_COMM_WORLD que no hace falta declarar.
- int **size:** variable de tipo entero devuelto por la función indicando el número de procesos asignados al sistema.
- **Nota sobre MPI_Comm:** Este tipo de dato guarda toda la información relevante sobre un comunicador específico. Se utiliza para especificar el comunicador por el que se desea realizar las operaciónes de transmisión o recepción (como MPI_Send o MPI_Recv).

Determina el número total de procesos existentes dentro de un mismo comunicador. El usuario puede definir otros comunicadores para designar subconjuntos de procesos.

Funciones básicas: Información

int MPI_Comm_rank(comm, *rank)

- MPI_Comm **comm:** variable de tipo MPI_Comm suministrado por el usuario indicando el comunicador utilizado. Por defecto, es MPI_COMM_WORLD.
- int **rank:** variable de tipo entero devuelto por la función indicando el número lógico que corresponde a cada proceso.

Devuelve el número lógico que corresponde a cada proceso. Este valor siempre empieza en cero y alcanza un valor máximo igual al número de procesos menos uno.

Funciones básicas: Información

int MPI_Get_processor_name(*name, *resultlen)

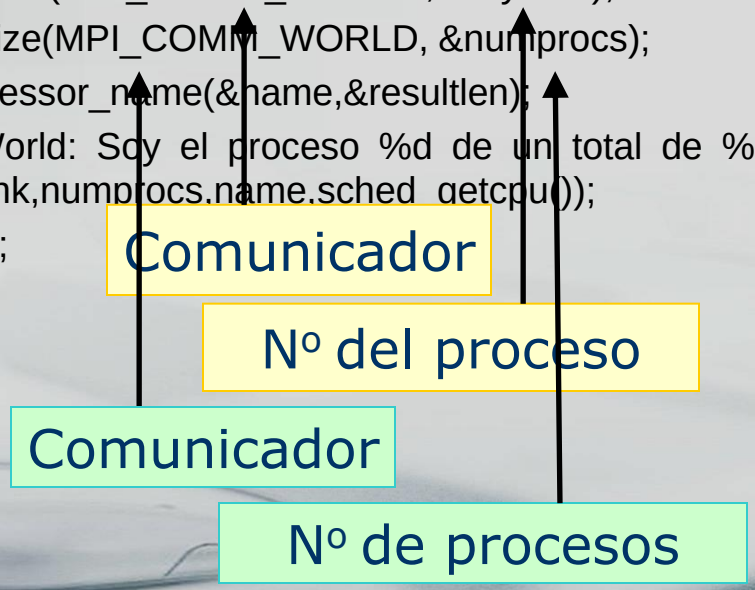
- char **name:** variable de tipo carácter devuelto por la función que indica el nodo en el que se está ejecutando la tarea. Debe ser un array de tamaño al menos MPI_MAX_PROCESSOR_NAME.
- int **resultlen:** variable de tipo entero devuelto por la función indicando la longitud (en caracteres) de name.

Obtiene el nombre del nodo en el que se está ejecutando la tarea que hace la llamada.

DSAP

Ejemplo 1

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv)
{
   int myrank, numprocs, resultlen;
   char name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Get_processor_name(&name,&resultlen);
   printf("Hello World: Soy el proceso %d de un total de %d. Estoy en el nodo %s y en el core
      %d\n",myrank,numprocs,name,sched_getcpu());
   MPI_Finalize();
}
```



DSAP

Salida del Ejemplo 1

- Después de la ejecución (mpirun –np 3 ejemplo1), la salida que produce el *ejemplo1* es:

```
Hello World: Soy el proceso  0 de un total de  3 procesos. Estoy en el nodo cluster1 y en el core 0
Hello World: Soy el proceso  2 de un total de  3 procesos. Estoy en el nodo cluster3 y en el core 2
Hello World: Soy el proceso  1 de un total de  3 procesos. Estoy en el nodo cluster2 y en el core 1
```

- Notar que los procesos se ejecutan asíncronamente.

DSAP

Compilación y ejecución

- Para la compilación se usan los scripts:

  mpif77, mpif90, mpicc

- De esta forma, la compilación del ejemplo 1 sería:

  mpicc –o ejemplo1 ejemplo1.c

- La ejecución de trabajos mpi se realiza con el script:

  mpirun

- La ejecución del ejemplo 1 sería:

  mpirun –np 3 ejemplo1

Le indicamos el número de procesos a utilizar: el número de copias del ejecutable, *ejemplo1*

Funciones básicas: Comunicación

- MPI ofrece dos (tres) tipos de comunicación:
  - **punto a punto**, del proceso i al j (participan ambos).
  - **en grupo** (colectiva): entre un grupo de procesos, de uno a todos, de todos a uno, o de todos a todos.
  - one-sided: del proceso i al j (participa uno solo).

- Además, básicamente en el caso de comunicación entre dos procesos, hay múltiples variantes en función de cómo se implementa el proceso de envío y de espera.

DSAP

Funciones básicas: Comunicación

- La comunicación entre procesos requiere (al menos) de dos participantes: emisor y receptor.

- El emisor ejecuta una función de **envío** y el receptor otra de **recepción**.

- La comunicación es un proceso **cooperativo**: si una de las dos funciones no se ejecuta, no se produce la comunicación (y podría generarse un **deadlock**).

Funciones básicas: Comunicación

Los modos de comunicación especificados por el send y received son:

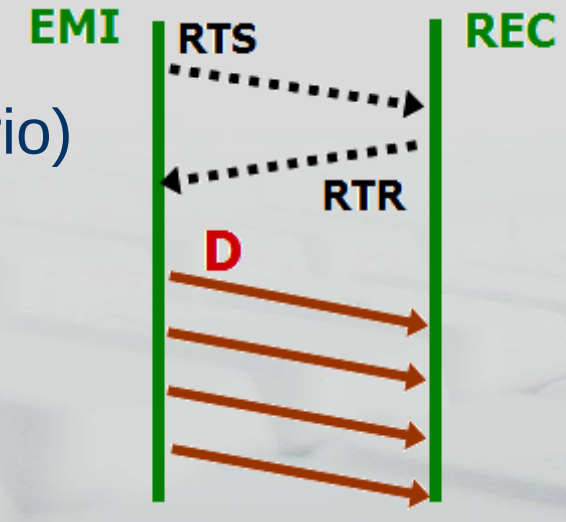
- **modo synchronous**.
- **modo ready**.
- **modo buffered**.
- **modo standard**.

DSAP

Funciones básicas: Comunicación

Modos de comunicación:

- **modo synchronous** (el más seguro y el más portable): la comunicación no se produce hasta que emisor y receptor se ponen de acuerdo (sin buffer intermedio).
  - petición de transmisión (espera)
  - aceptación de transmisión
  - envío de datos (de usuario a usuario)

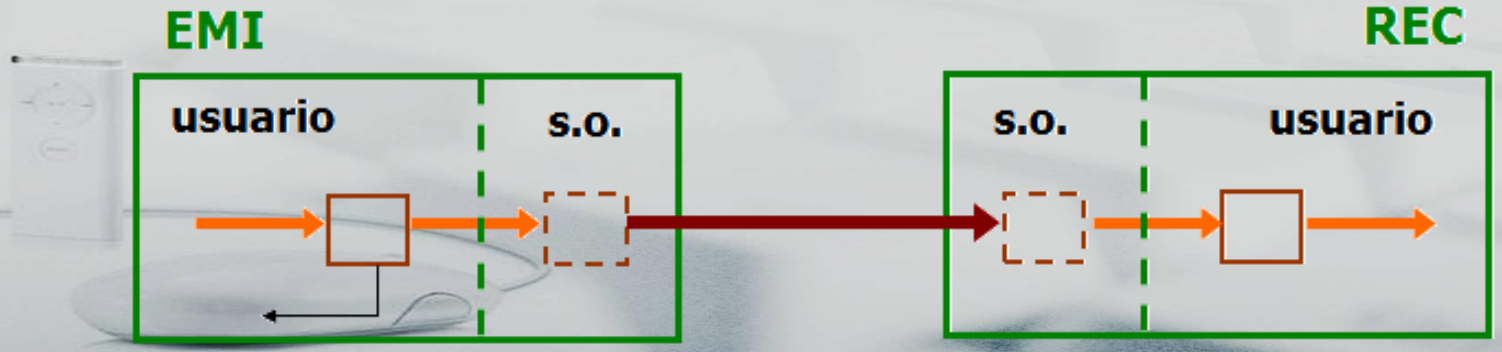


DSAP

Funciones básicas: Comunicación

Modos de comunicación:

- **modo ready** (el que ocasiona menor overhead al sistema).
- **modo buffered** (desacopla al que envía de quién recibe, permite que el usuario tenga el control): el emisor deja el mensaje en un buffer y retorna. La comunicación se produce cuando el receptor está dispuesto a ello. El buffer no se puede reutilizar hasta que se vacíe.



Funciones básicas: Comunicación

Modos de comunicación:

- **modo standard** (compromiso): utiliza la capacidad de buffering del sistema; es decir, retorna una vez copiado en el buffer el mensaje a enviar… ¡siempre que quepa!

Funciones básicas: Comunicación

Adicionalmente, los modos de comunicación pueden ser bloqueantes o no bloqueantes:

- **Bloqueante:** Se espera a que la "comunicación" se produzca, antes de continuar con la ejecución del programa. La comunicación síncrona es bloqueante. La comunicación con buffer también, si el mensaje no cabe en el buffer.
- **No bloqueante:** Se retorna "inmediatamente" de la función de comunicación, y se continúa con la ejecución. Se comprueba más tarde si la comunicación se ha efectuado.

Funciones básicas: Comunicación

Cada estrategia tiene sus ventajas e inconvenientes:

- **Síncrona**:
  - es más rápida si el receptor está dispuesto a recibir; nos ahorramos la copia en el buffer.
  - Además del intercambio de datos, sirve para sincronizar los procesos.
  - Ojo: al ser bloqueante es posible un **deadlock**!

- **Con buffer**:
  - el emisor no se bloquea si el receptor no está disponible,
  - pero hay que hacer copia(s) del mensaje,
  - más lento.

Funciones básicas: Comunicación

Los modos de comunicación especificados por el send y received son:

- modo synchronous (el más seguro y el más portable).
- modo ready (el que ocasiona menor overhead al sistema).
- modo buffered (desacopla al que envía de quién recibe, permite que el usuario tenga el control).
- modo standard (compromiso).

Blocking o non-blocking especificados por send y receive:

- blocking calls pueden estar en correspondencia con non-blocking calls.
- blocking suspende la ejecución hasta que el message buffer sea seguro de usar.
- non-blocking inicia la comunicación.

DSAP

Funciones básicas: Comunicación

Para enviar o recibir un mensaje es necesario especificar:

- **A quién** se envía (o **de quién** se recibe):
  - los **datos** a enviar (dirección de **comienzo** y **cantidad**).
  - el **tipo** de los datos.
  - la **clase** de mensaje (tag).

Todo lo que no son los datos forma el "sobre" del mensaje.

- Las dos funciones estándar para enviar y recibir mensajes son:

Funciones básicas:

Comunicación (Blocking Standard Send)

int MPI_Send(*buf, count, datatype, dest, tag, comm)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.

Un send/receive bloqueante suspende la ejecución del programa hasta que el buffer que se está enviando/recibiendo es seguro de usar. En el caso de un send bloqueante, esto significa que los datos que tienen que ser enviados han sido copiados en el buffer de envío (no necesariamente han sido recibidos por la tarea que recibe).

DSAP

Funciones básicas:

Comunicación (Blocking Standard Send)

int MPI_Send(*buf, count, datatype, dest, tag, comm)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Ca
- MPI_Datatyp
- int **dest:** Núm
- int **tag:** Ident

ha de comunica

- MPI_Comm

Un send/rece
hasta que el
usar. En el ca
que tienen qu
(no necesarian

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

Funciones básicas:

Comunicación (Blocking Standard Receive)

int MPI_Recv(*buf, count, datatype, source, tag, comm, *status)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **source:** Número lógico del proceso desde el cual se espera recibir información. MPI permite no especificar explícitamente el proceso desde el que se espera recibir; en este caso se indicará *source* como MPI_ANY_SOURCE.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío. MPI_ANY_TAG permite aceptar mensajes con cualquier identificador.
- MPI_Comm **comm:** Comunicador.
- MPI_Status **status:** Estructura interna que contiene los campos MPI_SOURCE, MPI_TAG y count.

Funciones básicas:

Comunicación (Blocking Standard Receive)

int MPI_Recv(*buf, count, datatype, source, tag, comm, *status)

- **buf:** Varia
- int **count**
- MPI_Data
- int **sourc**

información, que se e MPI_ANY_S

- int **tag:** Id

ha de comu con cualquie

- MPI_Com
- MPI_Status **status:** Estructura interna que contiene los campos MPI_SOURCE, MPI_TAG y count.

Sobre el tipo de dato MPI_Status

Se utiliza para guardar información sobre operaciones de recepción de mensajes (como por ejemplo MPI_Recv). Este tipo de dato guarda una estructura interna con los siguientes campos:

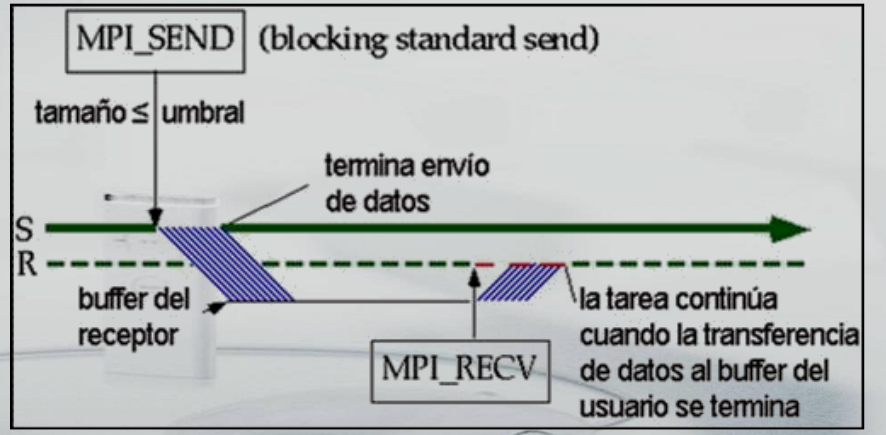
- **MPI_SOURCE**: Indica el rango del proceso origen en el comunicador en el que se realizó la transmisión (int).
- **MPI_TAG**: Valor de la etiqueta que tenía el mensaje (int).
- **count**: Tamaño del mensaje en bytes (para obtener el número de elementos, por ejemplo si fuera un entero, sería "status.count/sizeof(int)").

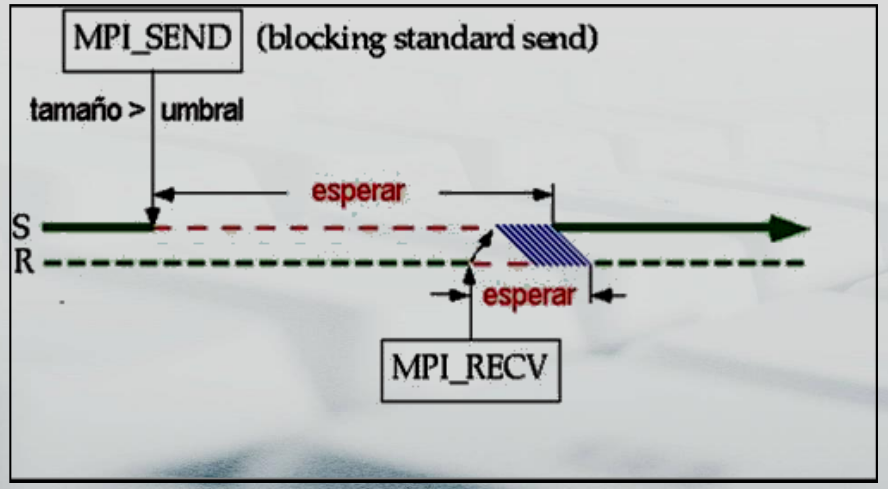
DSAP

Funciones básicas:

Comunicación (Blocking Standard Send/Recv)

- En el blocking standard send (MPI_Send (...)) se copia el mensaje sobre la red en el buffer del sistema del nodo que recibe, entonces la tarea que ejecuta el send continúa con su ejecución.
- El buffer del sistema se crea cuando comienza el programa (el usuario no necesita utilizarlo de ninguna manera). Hay un buffer del sistema por cada tarea que se encargará de manejar múltiples mensajes. El mensaje será copiado del buffer del sistema a la tarea que invoca el receive cuando se ejecute el receive.
- En ocasiones el buffer proporcionado por el sistema puede ser insuficiente pudiendo ocasionar problemas de interbloqueo.





DSAP

Ejemplo 2

```
main(int argc, char **argv) {
  int myrank, numprocs, i;
  MPI_Status estado;
  float dato=7.0, res=0.0;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  printf("Soy el proceso %d de un total de %d.\n",myrank,numprocs);
  if (myrank > 0) {
        res = dato * myrank;
        MPI_Send(&res,1, MPI_FLOAT, 0, 8, MPI_COMM_WORLD); }
  else {
       printf("Antes de recibir, el valor de res es %f\n",res);
       for ( i=1 ; i<numprocs ; i++ ) {
           MPI_Recv(&res,1, MPI_FLOAT, i, 8, MPI_COMM_WORLD, &estado);
           printf("Despues de recibir de %d, el valor de res es %f\n",i,res);
           printf("estado.MPI_SOURCE = %d, estado.MPI_TAG = %d, estado.count =
    %d\n", estado.MPI_SOURCE, estado.MPI_TAG, estado.count_lo);
       }
  }
  MPI_Finalize();
}
```

En este caso podríamos haber utilizado MPI_ANY_SOURCE

Salida del Ejemplo 2

- Después de la ejecución (mpirun –np 3 ejemplo2), la salida que produce el *ejemplo2* es:

```
Soy el proceso 0 de un total de 3.
Antes de recibir, el valor de res es 0.000000
Despues de recibir de 1, el valor de res es 7.000000
estado.MPI_SOURCE = 1, estado.MPI_TAG = 8, estado.count = 4

Despues de recibir de 2, el valor de res es 14.000000
estado.MPI_SOURCE = 2, estado.MPI_TAG = 8, estado.count = 4

Soy el proceso 1 de un total de 3.
Soy el proceso 2 de un total de 3.
```

Funciones básicas:

Comunicación

COMENTARIOS

- Si un proceso tiene varios mensajes pendientes de recibir, no se reciben en el orden en que se enviaron sino en el que se indica en la recepción mediante los parámetros de *origen* y *tag* del mensaje.

- Si el *tag* del mensaje que se recibe puede ser cualquiera, los mensajes que provienen del mismo *origen* se reciben en el orden en que se enviaron.

Funciones básicas:

Comunicación (Blocking Standard Send)

Envío de una columna en una matriz (caso C)

double A[100][200];

- **Solución 1:** Enviar uno a uno los elementos de la columna.

```
for (i=0; i<100; i++)
    MPI_SEND(&A[i][0], 1, MPI_DOUBLE, dest, tag, comm);
```

- **Solución 2:** Empaquetar los datos en un vector.

```
double c[100];
for (i=0; i<100; i++)  c[i]=A[i][0];
    MPI_SEND(&c, 100, MPI_DOUBLE, dest, tag, comm);
```

DSAP

Funciones básicas:

Comunicación (Blocking Standard Send)

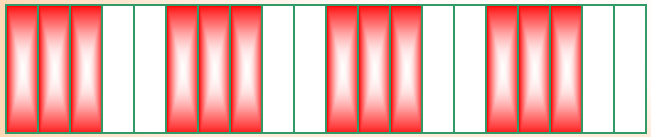
Envío de una columna en una matriz (caso C)

❑ **Solución 3:** Definir un nuevo tipo de dato.

```
double A[maxm][maxn]; → A[m][n]
int mitipo_col;
MPI_Type_vector(m, 1, maxn, MPI_DOUBLE, &mitipo_col);
MPI_TYPE_COMMIT(&mitipo_col);

MPI_SEND(&A[0][0], 1, mitipo_col, dest, tag, comm);
```

MPI_Type_vector(count, blocklength, stride, old_type, *new_type)



count = 4
blocklength = 3
stride = 5

Ejemplo 3
(envío de un bloque consecutivo de columnas)

```
#define maxm 1000
#define maxn 900
main(int argc, char **argv)
{
   int myrank,numprocs;
   int i,j,k,indice,columna_ini;
   int m,col_por_proceso;
   double A[maxm][maxn], aux;
   double start_time,end_time,total_time;
   MPI_Status estado;
   int request;
   int MPI_COL;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

Ejemplo 3
(envío de un bloque consecutivo de columnas)

```
if (myrank == 0) {
      printf("Numero de columnas a cada proceso (1-%d):“, maxn/numprocs);
      scanf("%d", &col_por_proceso);
      m = col_por_proceso*numprocs;
      printf("El orden de la matriz es: %d\n", m);
      for (i=0; i<m; i++) {
          for (j=0; j<m; j++) {
              A[i][j]=i+j;
          }
      }
      for (i=1; i<numprocs; i++) {
          MPI_Send(&m, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
          MPI_Send(&col_por_proceso, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      }
  }
  else {
      MPI_Recv(&m, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &estado);
      MPI_Recv(&col_por_proceso, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &estado);
  }
```

Ejemplo 3
(envío de un bloque consecutivo de columnas)

```
if (myrank == 0) {
     start_time = MPI_Wtime();
     columna_ini=col_por_proceso;                    //  OPCION 1 (elemento a elemento)
     for (k=1; k<numprocs; k++) {
       for (i=0; i<m; i++) {
         for (j=columna_ini; j<columna_ini+col_por_proceso; j++) {
           MPI_Send(&A[i][j],1,MPI_DOUBLE,k,9,MPI_COMM_WORLD);
       }}
       MPI_Recv(&m,1,MPI_INT,k,7,MPI_COMM_WORLD,&estado);
       columna_ini = columna_ini + col_por_proceso;
     }
     end_time = MPI_Wtime();
     total_time=end_time-start_time;
     printf("TIEMPO OPCION 1: %f\n",total_time);  }
else {
     for (i=0; i<m; i++)
       for (j=0; j<col_por_proceso; j++)
         MPI_Recv(&A[i][j],1,MPI_DOUBLE,0,9,MPI_COMM_WORLD,&estado);
     MPI_Send(&m,1,MPI_INT,0,7,MPI_COMM_WORLD);
}
```

Ejemplo 3
(envío de un bloque consecutivo de columnas)

```
if (myrank == 0) {
      start_time = MPI_Wtime();
      columna_ini=col_por_proceso;                                  //  OPCION 2 (fila a fila)
      for (k=1; k<numprocs; k++) {
        for (i=0; i<m; i++)
            MPI_Send(&A[i][columna_ini],col_por_proceso,MPI_DOUBLE,k,9,
                          MPI_COMM_WORLD);
        MPI_Recv(&m,1,MPI_INT,k,7,MPI_COMM_WORLD,&estado);
        columna_ini = columna_ini + col_por_proceso;
      }
      end_time = MPI_Wtime();
      total_time=end_time-start_time;
      printf("TIEMPO OPCION 2: %f\n",total_time);
  }
  else {
      for (i=0; i<m; i++)
         MPI_Recv(&A[i][0],col_por_proceso,MPI_DOUBLE,0,9,MPI_COMM_WORLD,&estado);
      MPI_Send(&m,1,MPI_INT,0,7,MPI_COMM_WORLD);
  }
```

Ejemplo 3
(envío de un bloque consecutivo de columnas)

```
MPI_Type_vector(m,1,maxn,MPI_DOUBLE,&MPI_COL);
MPI_Type_commit(&MPI_COL);
if (myrank == 0) {
      start_time = MPI_Wtime();                    // OPCION 3 (nuevo tipo)
      columna_ini=col_por_proceso;
      for (k=1; k<numprocs; k++) {
        for (j=columna_ini; j<columna_ini+col_por_proceso; j++)
             MPI_Send(&A[0][j],1,MPI_COL,k,9,MPI_COMM_WORLD);
        MPI_Recv(&m,1,MPI_INT,k,7,MPI_COMM_WORLD,&estado);
        columna_ini = columna_ini + col_por_proceso;
      }
      end_time = MPI_Wtime();
      total_time=end_time-start_time;
      printf("TIEMPO OPCION 1: %f\n",total_time);
 }
 else {
      for (j=0; j<col_por_proceso; j++)
          MPI_Recv(&A[0][j],1,MPI_COL,0,9,MPI_COMM_WORLD,&estado);
      MPI_Send(&m,1,MPI_INT,0,7,MPI_COMM_WORLD);
 }
MPI_Finalize();
}
```

Comunicaciones colectivas: Barreras

int MPI_Barrier(comm)

- MPI_Comm **comm:** comunicador.

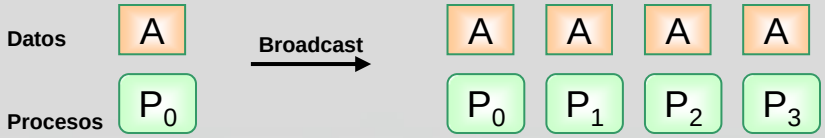
Los procesos incluidos en comm detienen su progreso al llamar a esta función y sólo continuarán hasta que todos los procesos hayan llamado a esta función.

DSAP

Comunicaciones colectivas: Broadcast uno-a-todos

int MPI_Bcast(*buf, count, datatype, root, comm)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **root:** Número lógico del proceso que hace el envío y desde el cual se espera recibir información.
- MPI_Comm **comm:** Comunicador.



Uno de los procesadores (el *root*) envía un mensaje a todos los procesadores incluídos en *comm*. *count* y *datatype* deben coincidir en todas las llamadas a esta función con objeto de que la cantidad de datos enviados y recibidos sea la misma.

Ejemplo 4

```
main(int argc, char **argv)
{
   int myrank, numprocs;
   float dato, res;
   dato = 7.0;
   res = 0.0;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   printf("Soy el proceso %d de un total de %d\n",myrank,numprocs);
   if (myrank == 1)   res = dato * dato;

   printf("Antes de recibir, el valor de res es %f\n",res);
   MPI_Bcast(&res, 1, MPI_FLOAT, 1, MPI_COMM_WORLD);
   printf("Despues de recibir, el valor de es %f\n",res);

   MPI_Finalize();
}
```

Salida del Ejemplo 4

- Después de la ejecución (mpirun –np 3 ejemplo4), la salida que produce el *ejemplo4* es:

```
Soy el proceso 0 de un total de 3
Antes de recibir, el valor de res es 0.000000
Despues de recibir, el valor de es 49.000000

Soy el proceso 2 de un total de 3
Antes de recibir, el valor de res es 0.000000
Despues de recibir, el valor de es 49.000000

Soy el proceso 1 de un total de 3
Antes de recibir, el valor de res es 49.000000
Despues de recibir, el valor de es 49.000000
```

Comunicaciones colectivas: MPI_IN_PLACE

Las operaciones colectivas pueden ejecutarse con la opción “in place”, cuando el buffer de salida es el mismo que el buffer de entrada. La forma de especificar esta situación es a través de un valor especial en uno de sus argumentos, MPI_IN_PLACE, en lugar del buffer de salida o del buffer de entrada, según el caso.

La operación “in place” reduce movimientos de memoria no necesarios tanto por la implementación de MPI como por el usuario.

Con la opción “in place”, el buffer de recepción se transforma en un buffer de envío y recepción, en muchas comunicaciones colectivas.

Comunicaciones colectivas: Reducción todos-a-uno

int MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, dest, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- **recvbuf:** Variable que contiene la información a recibir.
- int **count:** Cantidad de elementos contenidos en sendbuf.
- MPI_Datatype **datatype:** Tipo de la variable sendbuf y recvbuf.
- MPI_Op **op:** Operación a ejecutar.
- int **dest:** Número lógico del proceso al cual se ha transferido información.
- MPI_Comm **comm:** Comunicador.

MPI_REDUCE Combina los elementos almacenados en sendbuf de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el resultado en recvbuf del proceso dest. Notar que tanto sendbuf como recvbuf deben tener el mismo número de elementos de tipo datatype; asimismo, todos los procesos involucrados en la operación deben llamar a esta función con el mismo valor de count, datatype, op y dest.
La opción “in place" puede ser especificada con el valor MPI_IN_PLACE en el argumento sendbuf en el proceso root. En este caso, los datos de entrada en el proceso root, se toman del buffer de recepción recvbuf, donde serán reemplazados con los datos de salida.

Comunicaciones colectivas: Reducción todos-a-uno

int MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op, dest, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- **recvbuf:** Variable que contiene la información a recibir.
- int **count:** Cantidad de elementos contenidos en sendbuf.
- MPI_Datatype **datatype:** Tipo de la variable sendbuf y rec
- MPI_Op **op:** Operación a ejecutar.
- int **dest:** Número lógico del proceso al cual se ha transferid
- MPI_Comm **comm:** Comunicador.

Operación	Significado
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
…	…

MPI_REDUCE Combina los elementos almacenados en sendbuf de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el resultado en recvbuf del proceso dest. Notar que tanto sendbuf como recvbuf deben tener el mismo número de elementos de tipo datatype; asimismo, todos los procesos involucrados en la operación deben llamar a esta función con el mismo valor de count, datatype, op y dest.

La opción “in place" puede ser especificada con el valor MPI_IN_PLACE en el argumento sendbuf en el proceso root. En este caso, los datos de entrada en el proceso root, se toman del buffer de recepción recvbuf, donde serán reemplazados con los datos de salida.

Ejemplo 5

```
main(int argc, char **argv)
{
   int myrank, numprocs, i;
   MPI_Status estado;
   double x[10], y[10];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   for (i=0; i<10; i++)
   {
        x[i]=(myrank+1) * i;
        y[i]=0.0;
   }
   printf("Soy %d. Antes de recibir el valor de x es:  ",myrank);
   for (i=0; i<10; i++) printf("%4.1f ",x[i]);
   printf("\n");
   MPI_Reduce(&x,&y,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
   printf("Soy %d. Despues de recibir el valor de y es:",myrank);
   for (i=0; i<10; i++) printf("%4.1f ",y[i]);
   printf("\n");
   MPI_Finalize();
}
```

Si quisiéramos usar la misma variable x para la entrada-salida:

```
MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD;
```

Provocaría un error:

Buffers must not be aliased

SOLUCIÓN: Usar MPI_IN_PLACE en el root

```
if (myrank != 0)
   MPI_Reduce(&x,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
else
   MPI_Reduce(MPI_IN_PLACE,&x,10,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

Salida del Ejemplo 5

- Después de la ejecución (mpirun –np 2 ejemplo5), la salida que produce el *ejemplo5* es:

```
Soy 0. Antes de recibir el valor de x es:     0.0 1.0 2.0 3.0  4.0  5.0  6.0  7.0  8.0  9.0
Soy 0. Despues de recibir el valor de y es: 0.0 3.0 6.0 9.0 12.0 15.0 18.0 21.0 24.0 27.0
Soy 1. Antes de recibir el valor de x es:     0.0 2.0 4.0 6.0  8.0 10.0 12.0 14.0 16.0 18.0
Soy 1. Despues de recibir el valor de y es: 0.0 0.0 0.0 0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Comunicaciones colectivas: Reducción todos-a-todos

int MPI_Allreduce (*sendbuf, *recvbuf, count, datatype, op, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- **recvbuf:** Variable que contiene la información a recibir.
- int **count:** Cantidad de elementos contenidos en sendbuf.
- MPI_Datatype **datatype:** Tipo de la variable sendbuf y recvbuf.
- MPI_Op **op:** Operación a ejecutar.
- MPI_Comm **comm:** Comunicador.

MPI_ALLREDUCE combina los elementos almacenados en sendbuf de cada proceso definido en el comunicador comm, utilizando la operación op, y regresa el resultado en recvbuf de cada proceso. Notar que tanto sendbuf como recvbuf deben tener el mismo número de elementos de tipo datatype; asimismo, todos los procesos involucrados en la operación deben llamar a esta función con el mismo valor de count, datatype, op.

La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento sendbuf en todos los procesos. En este caso, los datos de entrada se toman del buffer de recepción, recvbuf, donde serán reemplazados por los datos de salida.

DSAP

Ejemplo 6

```
main(int argc, char **argv)  {
   int myrank, numprocs, i, m;
   MPI_Status estado;
   double *x, *y;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (myrank == 0) {
         printf("Longitud de los vectores: "); scanf("%i",&m); }
   MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
   x = (double *)malloc(m * sizeof(double));
   y = (double *)malloc(m * sizeof(double));
   for (i=0; i<m-1; i++) {
        x[i]=(myrank+1) * i; y[i]=0.0;  }
   printf("Soy %d. Antes de recibir el valor de x es:  ",myrank);
   for (i=0; i<10; i++) printf("%4.1f ",x[i]);   printf("\n");
   MPI_Allreduce(x,y,m,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
   printf("Soy %d. Despues de recibir el valor de y es:",myrank);
   for (i=0; i<9; i++) printf("%4.1f ",y[i]);   printf("\n");
   free(x);  free(y);
MPI_Finalize(); }
```

Podríamos usar la misma variable x para la entrada-salida con la opción MPI_IN_PLACE:

```
MPI_Allreduce(MPI_IN_PLACE,x,m,MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
```

Salida del Ejemplo 6

- Después de la ejecución (mpirun –np 3 ejemplo6), la salida que produce el *ejemplo6* es:

```
Soy 0. Antes de recibir el valor de x es:      0.0  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0
Soy 0. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0

Soy 2. Antes de recibir el valor de x es:      0.0  3.0   6.0   9.0 12.0 15.0 18.0 21.0 24.0
Soy 2. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0

Soy 1. Antes de recibir el valor de x es:      0.0  2.0   4.0   6.0   8.0 10.0 12.0 14.0 16.0
Soy 1. Despues de recibir el valor de y es: 0.0  6.0 12.0 18.0 24.0 30.0 36.0 42.0 48.0
```

DSAP

Comunicaciones colectivas: Scatter/Gather

int MPI_Scatter(*sendbuf, sendcnt, sendtype, *recvbuf, recvcnt, recvtype, root, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- int **sendcnt:** Tamaño de los segmentos a comunicar.
- MPI_Datatype **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- int **recvcnt:** Cantidad de elementos contenidos en recvbuf.
- MPI_Datatype **recvtype:** Tipo de la variable recvbuf.
- int **root:** Número lógico del proceso que hace el envío y desde el cual se espera recibir información.
- MPI_Comm **comm:** Comunicador.

El proceso raíz (root) dispone del mensaje que es dividido en segmentos de igual tamaño (sendcnt). El i-ésimo segmento se envía al i-ésimo proceso del grupo (recvbuf). La cantidad de datos enviados tiene que ser igual a la cantidad de datos recibidos y debe coincidir en todos los procesos.

La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento recvbuf del proceso root. En este caso, recvcnt y recvtype se ignoran, y el proceso root no envía nada a sí mismo.

DSAP

Ejemplo 7

```
main(int argc, char **argv)   {
   int myrank, numprocs, i, lm, m, root;
   MPI_Status estado;
   double *x, *y, ldot, gdot;
   double double sdot(int, double *, double *);
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   printf("Soy el proceso %d de un total de %d\n",myrank,numprocs);
   root = 0;
   if (myrank == root)  {
        printf("Longitud de los vectores en cada proceso: ");
        scanf("%i",&lm);
        m = lm * numprocs;
        printf("Longitud de los vectores a multiplicar: %i\n",m);
        x = (double *)malloc(m * sizeof(double));
        y = (double *)malloc(m * sizeof(double));
        for (i=0; i<m; i++)  {
             x[i]= i+1; y[i]=2*(i+1);  } }
   MPI_Bcast(&lm, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (myrank != root)   {
        x = (double *)malloc(lm * sizeof(double));
        y = (double *)malloc(lm * sizeof(double));   }
  ...
  ...
   free(x); free(y);
   MPI_Finalize();
}
```

```
double sdot(int lm, double *x, double *y) {
        int k;
        double valor;
        valor = 0.0;
        for (k=0;k<lm;k++)
           valor = valor + x[k]*y[k];
        return valor; }
```

```
if (myrank != root) {
MPI_Scatter(x,lm,MPI_DOUBLE,
                x,lm,MPI_DOUBLE, root,MPI_COMM_WORLD);
MPI_Scatter(y,lm,MPI_DOUBLE,
                y,lm,MPI_DOUBLE, root,MPI_COMM_WORLD); }
else {
MPI_Scatter(x,lm,MPI_DOUBLE,
    MPI_IN_PLACE,lm,MPI_DOUBLE, root,MPI_COMM_WORLD);
MPI_Scatter(y,lm,MPI_DOUBLE,
    MPI_IN_PLACE,lm,MPI_DOUBLE, root,MPI_COMM_WORLD);}
printf("Soy %d. Despues de scatter el valor de x es:",myrank);
for (i=0; i<5; i++) printf("%4.1f ",x[i]);  printf("\n");
printf("Soy %d. Despues de scatter el valor de y es:",myrank);
for (i=0; i<5; i++) printf("%4.1f ",y[i]);  printf("\n");
 ldot = sdot(lm,x,y);
printf("Producto interno parcial del proceso %d: %f\n",myrank, ldot);
 MPI_Reduce(&ldot,&gdot,1,MPI_DOUBLE,
                MPI_SUM,root,MPI_COMM_WORLD);
if (myrank == root) printf("Producto interno global: %f\n",gdot);
```

Salida del Ejemplo 7

- Después de la ejecución (mpirun –np 3 ejemplo7), la salida que produce el *ejemplo7* es:

```
Soy el proceso 0 de un total de 3
Longitud de los vectores en cada proceso: 1000000
Longitud de los vectores a multiplicar: 3000000
Soy 0. Despues de scatter el valor de x es: 1.0  2.0  3.0  4.0  5.0
Soy 0. Despues de scatter el valor de y es: 2.0  4.0  6.0  8.0 10.0
Producto interno parcial del proceso 0: 666667666666255104.000000
Producto interno global: 18000009000017612800.000000

Soy el proceso 1 de un total de 3
Soy 1. Despues de scatter el valor de x es:1000001.0 1000002.0 1000003.0 1000004.0 1000005.0
Soy 1. Despues de scatter el valor de y es:2000002.0 2000004.0 2000006.0 2000008.0 2000010.0
Producto interno parcial del proceso 1: 4666669666673246208.000000

Soy el proceso 2 de un total de 3
Soy 2. Despues de scatter el valor de x es:2000001.0 2000002.0 2000003.0 2000004.0 2000005.0
Soy 2. Despues de scatter el valor de y es:4000002.0 4000004.0 4000006.0 4000008.0 4000010.0
Producto interno parcial del proceso 2: 126666716666678110208.000000
```

DSAP

Comunicaciones colectivas: Scatter/Gather

int MPI_Gather(*sendbuf, sendcnt, sendtype, *recvbuf, recvcnt, recvtype, root, comm)

- **sendbuf:** Variable que contiene la información a comunicar.
- int **sendcnt:** Tamaño de los segmentos a comunicar.
- MPI_Datatype **sendtype:** Tipo de la variable sendbuf.
- **recvbuf:** Variable que contiene la información a recibir.
- int **recvcnt:** Cantidad de elementos contenidos en recvbuf.
- MPI_Datatype **recvtype:** Tipo de la variable recvbuf.
- int **root:** Número lógico del proceso que espera recibir la información.
- MPI_Comm **comm:** Comunicador.

Cada proceso (incluido el root) envía el contenido de su buffer de envío al proceso root. El proceso raíz recibe los mensajes y los almacena por orden del número de proceso. El buffer de recepción (recvbuf) es ignorado en todos los procesos distintos del root. La cantidad de datos enviados tiene que ser igual a la cantidad de datos recibidos y debe coincidir en todos los procesos.

La opción “in place” se especifica con el valor MPI_IN_PLACE en el argumento sendbuf del proceso root. En este caso, sendcnt y sendtype se ignoran, y la contribución del proceso root al vector de salida se asume que está en su lugar correcto del buffer de recepción recvbuf.

Comunicaciones colectivas:

- Otras versiones de estas funciones son:
  - **MPI_Allgather**(…); al final, todos los procesos disponen de todos los datos.
  - **MPI_Gatherv**(…); la información que se recolecta es de tamaño variable.
  - **MPI_Allgatherv**(…); "suma" de las dos anteriores.
  - **MPI_Alltoall**(…); todos los procesos distribuyen datos a todos los procesos.
  - **MPI_Scatterv**(…); la información que se distribuye es de tamaño variable.
  - **MPI_Alltoallv**(…); "suma" de las dos anteriores.

Ejemplo 8

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#define m 7
#define n 10
main(int argc, char **argv)
{
   void vermatriz(int max, double a[][max], int , int , char []);
   int myrank, numprocs, i, j, lm, root, resto, slice;
   double A[m][n], B[m][n];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   root = 0;
   if (myrank == root) {
      resto = m % numprocs;
      for (i=0; i<m; i++)
         for (j=0; j<n; j++)
            A[i][j] = i-j;  }
   else {
      resto = 0;
   }
   slice = m / numprocs;
   lm = slice + resto;
   if (myrank != root) {
      MPI_Scatter(&A[resto][0],slice*n,MPI_DOUBLE,
            &A[resto][0],slice*n,MPI_DOUBLE,
            root,MPI_COMM_WORLD); }
   else {
      MPI_Scatter(&A[resto][0],slice*n,MPI_DOUBLE,
            MPI_IN_PLACE,slice*n,MPI_DOUBLE,
            root,MPI_COMM_WORLD); }
   printf("Proceso %d\n",myrank);
   vermatriz(n,A,lm,n,"A");
   for (i=0; i<lm; i++)
      for (j=0; j<n; j++)
         B[i][j] = 2*A[i][j];
   if (myrank != root) {
      MPI_Gather(&B[resto][0],slice*n,MPI_DOUBLE,
            &B[resto][0],slice*n,MPI_DOUBLE,
            0, MPI_COMM_WORLD); }
   else {
      MPI_Gather(MPI_IN_PLACE,slice*n,MPI_DOUBLE,
            &B[resto][0],slice*n,MPI_DOUBLE,
            0, MPI_COMM_WORLD); }
   if (myrank == 0){
      vermatriz(n,B,m,n,"B"); }
   MPI_Finalize();
}
```

DSAP

Salida del Ejemplo 8

- Después de la ejecución (mpirun –np 3 ejemplo8), la salida que produce el *ejemplo8* es:

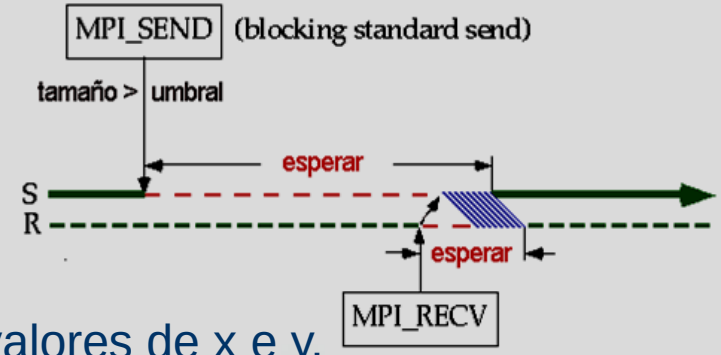
```
Proceso 0
   A=       0       1       2       3       4       5       6       7       8       9
      0:  0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000  -9.000
      1:  1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000  -8.000
      2:  2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000  -7.000
   B=       0       1       2       3       4       5       6       7       8       9
      0:  0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000 -18.000
      1:  2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000 -16.000
      2:  4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000 -14.000
      3:  6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000 -12.000
      4:  8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000 -10.000
      5: 10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000  -8.000
      6: 12.000  10.000   8.000   6.000   4.000   2.000   0.000  -2.000  -4.000  -6.000
Proceso 1
   A=       0       1       2       3       4       5       6       7       8       9
      0:  3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000  -6.000
      1:  4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000  -5.000
Proceso 2
   A=       0       1       2       3       4       5       6       7       8       9
      0:  5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000  -4.000
      1:  6.000   5.000   4.000   3.000   2.000   1.000   0.000  -1.000  -2.000  -3.000
```

DSAP

Problemas de interbloqueo

- El uso de blocking standard send (MPI_Send (...)) puede en ocasiones generar un interbloqueo en los procesos cuando se sobrepasa un deteminado umbral en el tamaño de los mensajes enviados.

Proceso 0	Proceso 1
send(x, 1)	send(y, 0)
receive(y, 1)	receive(x, 0)



Ambos procesos pretenden intercambiar sus valores de x e y.

….pero

- La operación send (síncrona) de cada proceso está esperando el correspondiente receive del segundo proceso implicado en la operación.
- Asimismo, la operación receive de ambos procesos no se ejecuta nunca ya que la operación de envío no finaliza.

Como consecuencia, ninguno de los procesos puede proceder con su ejecución, es decir, se encuentran interbloqueados.

Ejemplo 10

```
main(int argc, char **argv)   {
   double *mensaje1, *mensaje2;
   int rank,dest,ori,numprocs, msglen;
   int send_eti, recv_eti, i;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (numprocs != 2) {
        if (rank == 0) printf("Ejecuta con solo 2 procesos!!\n");
        MPI_Finalize ();
        return 0;  }
   if (rank == 0) {
        printf("Longitud de los vectores a enviar: ");
        scanf("%i",&msglen);  }
   MPI_Bcast(&msglen,1,MPI_INT,0,
                  MPI_COMM_WORLD);
   mensaje1 = (double *)malloc(msglen * sizeof(double));
   mensaje2 = (double *)malloc(msglen * sizeof(double));
   for (i=0; i<msglen; i++)   {
          mensaje1[i]= 100;
          mensaje2[i]=-100;     }

   if ( rank == 0 ) {
          dest = 1;
          ori = 1;
          send_eti = 10;
          recv_eti = 20;
     } else {
          dest = 0;
          ori = 0;
          send_eti = 20;
          recv_eti = 10;
     }
     printf(" Tarea %d esta enviando el mensaje\n", rank);
     MPI_Send(mensaje1,msglen,MPI_DOUBLE,dest,send_eti,
                    MPI_COMM_WORLD);
     MPI_Recv(mensaje2,msglen,MPI_DOUBLE,ori,recv_eti,
                    MPI_COMM_WORLD, &status);
     printf(" Tarea %d ha recibido el mensaje\n", rank);
     free(mensaje1);
     free(mensaje2);
     MPI_Finalize();
}
```

Este programa entra en interbloqueo cuando se ejecuta con un valor elevado de *msglen*

Problemas de interbloqueo

MPI proporciona varias alternativas con las que resolver estos problemas de interbloqueo:

- Utilizar la función **MPI_Bsend** que permite gestionar su propio buffer para la comunicación y garantizar que la función de envío hace la copia del mensaje de forma correcta. Debe utilizarse con las funciones de gestión del buffer.
- Usar la función **MPI_Sendrecv** que combina, en una sola llamada, el envío y la recepción. Es una función bloqueante que permite prevenir interbloqueos como consecuencia de situaciones de espera circular.
- Hacer uso de las operaciones punto-a-punto no bloqueantes **MPI_Isend** y **MPI_Irecv** en combinación con las funciones **MPI_Test** y **MPI_Wait** que permiten validar o esperar el resultado de una operación no bloqueante.

Problemas de interbloqueo:
Nonblocking Standard Send

int MPI_Isend(*buf, count, datatype, dest, tag, comm, *request)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.
- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_ISEND.

Envía un mensaje a otro proceso. El proceso origen continúa su trabajo sin esperar a que el proceso destinatario haya recibido el mensaje.

Problemas de interbloqueo:
Nonblocking Standard Receive

int MPI_Irecv(*buf, count, datatype, source, tag, comm, *request)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **source:** Número lógico del proceso desde el cual se espera recibir información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.
- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_IRECV.

Se dispone a recibir un mensaje de parte de otro proceso y continua su trabajo sin esperar a recibirlo por completo.

Problemas de interbloqueo:
Nonblocking Standard Send/Receive

int MPI_Wait(*request, *status)

- MPI_Request **request:** En combinación con las funciones MPI_TEST y MPI_WAIT proporciona información sobre el estado de la función MPI_ISEND y MPI_IRECV.
- MPI_Status **status:** Auxiliar necesario para conocer el estado de ejecución de una función MPI.

Una llamada a la función MPI_WAIT regresa cuando la operación no bloqueada identificada por **request** ha concluido.


```
main(int argc, char **argv)   {
   double *mensaje1, *mensaje2;
   int rank,dest,ori,numprocs, msglen;
   int send_eti, recv_eti, i;
   MPI_Status status;
   MPI_Request request1, request2;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,
   MPI_Comm_size(MPI_COMM_WORLD,
   if (numprocs != 2) {
        if (rank == 0) printf("Ejecuta con solo
        MPI_Finalize ();
        return 0;  }
   if (rank == 0) {
        printf("Longitud de los vectores a enviar: ");
        scanf("%i",&msglen);
        printf("%d\n",msglen); }
   MPI_Bcast(&msglen,1,MPI_INT,0,
                MPI_COMM_WORLD);
   mensaje1 = (double *)malloc(msglen * sizeof(double));
   mensaje2 = (double *)malloc(msglen * sizeof(double));
   for (i=0; i<msglen; i++)   {
          mensaje1[i]= 100;
          mensaje2[i]=-100;      }
```

Distintas ejecuciones:

- Tare
- Tare
- Tare
- Tare

==========
BAD TERM

YOU
==========

Distintas ejecuciones:

- Longitud de los vectores a enviar: 100000

Tarea 0: m2[0]= 100.0 m2[1]= 100.0 m2[2]= 100.0

Tarea 1: m2[0]= 100.0 m2[1]= 100.0 m2[2]= 100.0

CORRECTO!!

```
                MPI_COMM_WORLD,&request2);
   MPI_Wait ( &request1, &status );
   MPI_Wait ( &request2, &status );
   printf("Tarea %d: ", rank);
   for (i=0; i<3; i++) printf("m2[%d]=%6.1f ", i, mensaje2[i]);
   printf("\n");
   free(mensaje1);
   free(mensaje2);
   MPI_Finalize();
}
```

Ejemplo 12

BLOCKING CALLS
PUEDEN ESTAR EN CORRESPONDENCIA
CON
NON-BLOCKING CALLS

Ejemplo 12

```
main(int argc, char **argv)   {
   double *mensaje1, *mensaje2;
   int rank,dest,ori,numprocs, msglen;
   int send_eti, recv_eti, i;
   MPI_Status status;
   MPI_Request request;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   if (numprocs != 2) {
        if (rank == 0) printf("Ejecuta con solo 2 procesos!!\n");
        MPI_Finalize ();
        return 0;  }
   if (rank == 0) {
        printf("Longitud de los vectores a enviar: ");
        scanf("%i",&msglen);
        printf("%d\n",msglen); }
   MPI_Bcast(&msglen,1,MPI_INT,0,
                  MPI_COMM_WORLD);
   mensaje1 = (double *)malloc(msglen * sizeof(double));
   mensaje2 = (double *)malloc(msglen * sizeof(double));
   for (i=0; i<msglen; i++)   {
          mensaje1[i]= 100;
          mensaje2[i]=-100;      }

   if ( rank == 0 ) {
          dest = 1;
          ori = 1;
          send_eti = 10;
          recv_eti = 20;
     } else {
          dest = 0;
          ori = 0;
          send_eti = 20;
          recv_eti = 10;
     }
     MPI_Isend(mensaje1,msglen,MPI_DOUBLE,dest,send_eti,
                  MPI_COMM_WORLD,&request);
     MPI_Recv(mensaje2,msglen,MPI_DOUBLE,ori,recv_eti,
                  MPI_COMM_WORLD, &status);
     MPI_Wait ( &request, &status );
     printf("Tarea %d: ", rank);
     for (i=0; i<3; i++) printf("m2[%d]=%6.1f ", i, mensaje2[i]);
     printf("\n");
     free(mensaje1);
     free(mensaje2);
     MPI_Finalize();
}
```

Problemas de interbloqueo:
Blocking Buffered Send

int MPI_Bsend(*buf, count, datatype, dest, tag, comm)

- **buf:** Variable que contiene la información a comunicar.
- int **count:** Cantidad de elementos contenidos en buf.
- MPI_Datatype **datatype:** Tipo de la variable buf.
- int **dest:** Número lógico del proceso al cual se ha transferido información.
- int **tag:** Identifica el envío. Generalmente es cero y sólo cambia cuando se ha de comunicar más de un envío.
- MPI_Comm **comm:** Comunicador.

Realiza un envío en modo blocking buffered. No depende de una operación de recepción para finalizar. Si no existe recepción, el mensaje se dirige al buffer para completar la llamada. **MPI_Bsend** no garantiza que el mensaje se ha enviado, sino que queda en el buffer hasta que su correspondiente recepción. Necesita del uso de la función **MPI_Buffer_attach**.

Problemas de interbloqueo:
Blocking Buffered Send (espacio en buffer)

int MPI_Buffer_attach(*buffer, size)

- **buffer:** Debe apuntar a un array existente que no debe ser usado por el programador (input).
- int **size:** Tamaño en bytes del buffer (input).

Esta función provee a MPI de un buffer en el espacio de memoria del usuario que se utiliza para el envío de mensajes en modo buffered. Sólo un buffer puede ser declarado para una tarea en cada momento.

El buffer puede ser liberado con
int MPI_Buffer_detach(void *buffer, int *size)

Problemas de interbloqueo:
Blocking Buffered Send (espacio en buffer)

int MPI_Pack_size(incount, datatype, comm, *size)

- int **incount:** Número de elementos del tipo datatype (input).
- MPI_Datatype **datatype:** Tipo de la variable (input).
- MPI_Comm **comm:** Comunicador (input).
- int **size:** Cota superior del tamaño de mensaje (output, en bytes).

Devuelve una cota superior de la cantidad de espacio (en bytes) requerido para un mensaje de tamaño *incount* y tipo *datatype.*

Adicionalmente, un send utiliza algo más de espacio determinado por MPI_BSEND_OVERHEAD, con lo que la cantidad de espacio requerido para un mensaje de tamaño *incount* y tipo *datatype* es:

size + MPI_BSEND_OVERHEAD

Ejemplo 14

- Se generan un total de *ntask* tareas.
- Cada una define un vector de tamaño *size.*
- Cada tarea *i* envía el vector a la tarea *i-1*. La tarea *0* envía a la tarea *ntask-1*. Este proceso se realiza *ntask* veces, con lo que al final cada proceso debe almacenar el mismo vector que el inicial.
- Uso de Nonblocking Standard Send:
  - Al finalizar, los procesos no almacenan el mismo vector inicial.
  - Comportamiento no determinista debido al exceso de mensajes entre procesos.
- Uso de Blocking Buffered Send:
  - Al finalizar, los procesos SÍ almacenan el mismo vector.
  - El uso del buffer garantiza que el dato enviado será recibido aunque se modifique en el proceso origen.

Mediciones de tiempo

- Los sistemas operativos generalmente proporcionan comandos de línea que permiten cronometrar la ejecución de un código de principio a fin.
- Aún cuando esta opción es valiosa para el desarrollador, en ocasiones es necesario cronometrar la ejecución de segmentos de código para estimar su eficiencia.

```
double MPI_Wtime()

double MPI_Wtick()
```

Mediciones de tiempo

double MPI_Wtime()

Devuelve un número en coma flotante, de segundos que representan un cierto lapso de tiempo con respecto a un tiempo pasado.

double MPI_Wtick()

Regresa la resolución de reloj, o cantidad de segundos entre cuentas sucesivas de reloj, asociada a MPI_Wtime. El valor se reporta en segundos y como un número de doble precisión. De esta manera, una resolución de 0.001 indica que el sistema incrementa el contador del reloj cada milisegundo.

Mediciones de tiempo: uso

```
…
double start_time, end_time, clock_res;
…
start_time = MPI_Wtime();
…
… cálculos
…
end_time = MPI_Wtime();
clock_res = MPI_Wtick();
printf("El sistema tiene una resolucion de reloj = %f\n',clock_res);
Printf("Tiempo de ejecucion = %f seg\n", end_time - start_time);
```

Ejemplo 13

```
main(int argc, char **argv)   {
  …. //definicion de variables
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   printf("Soy el proceso %d de un total de %d\n",myrank,numprocs);
   root = 0;
   if (myrank == root)  {
       printf("Longitud de los vectores en cada proceso: ");  scanf("%i",&lm);
       m = lm * numprocs;
       printf("Longitud de los vectores a multiplicar: %i\n",m);
       x = (double *)malloc(m * sizeof(double));
       y = (double *)malloc(m * sizeof(double));
       for (i=0; i<m; i++)  {
            x[i]= i+1; y[i]=1.0/(i+1);  } }
   MPI_Bcast(&lm, 1, MPI_INT, 0,
          MPI_COMM_WORLD);
   if (myrank != root)   {
       x = (double *)malloc(lm * sizeof(double));
       y = (double *)malloc(lm * sizeof(double));   }
  …
  …
   free(x); free(y);
   MPI_Finalize();  }
```

```
start_time = MPI_Wtime();
 if (myrank != root) {
    MPI_Scatter(x,lm,MPI_DOUBLE,
                x,lm,MPI_DOUBLE,
                root,MPI_COMM_WORLD);
    MPI_Scatter(y,lm,MPI_DOUBLE,
                 y,lm,MPI_DOUBLE,
                 root,MPI_COMM_WORLD); }
 else {
    MPI_Scatter(x,lm,MPI_DOUBLE,
                MPI_IN_PLACE,lm,MPI_DOUBLE,
                root,MPI_COMM_WORLD);
    MPI_Scatter(y,lm,MPI_DOUBLE,
                MPI_IN_PLACE,lm,MPI_DOUBLE,
                root,MPI_COMM_WORLD);  }
ldot = sdot(lm,x,y);
MPI_Reduce(&ldot,&gdot,1,MPI_DOUBLE,
              MPI_SUM,root,MPI_COMM_WORLD);
end_time = MPI_Wtime();
if(myrank==root)
 printf("Producto interno global: %f\nTiempo %f:\n",
          gdot,end_time-start_time);
```

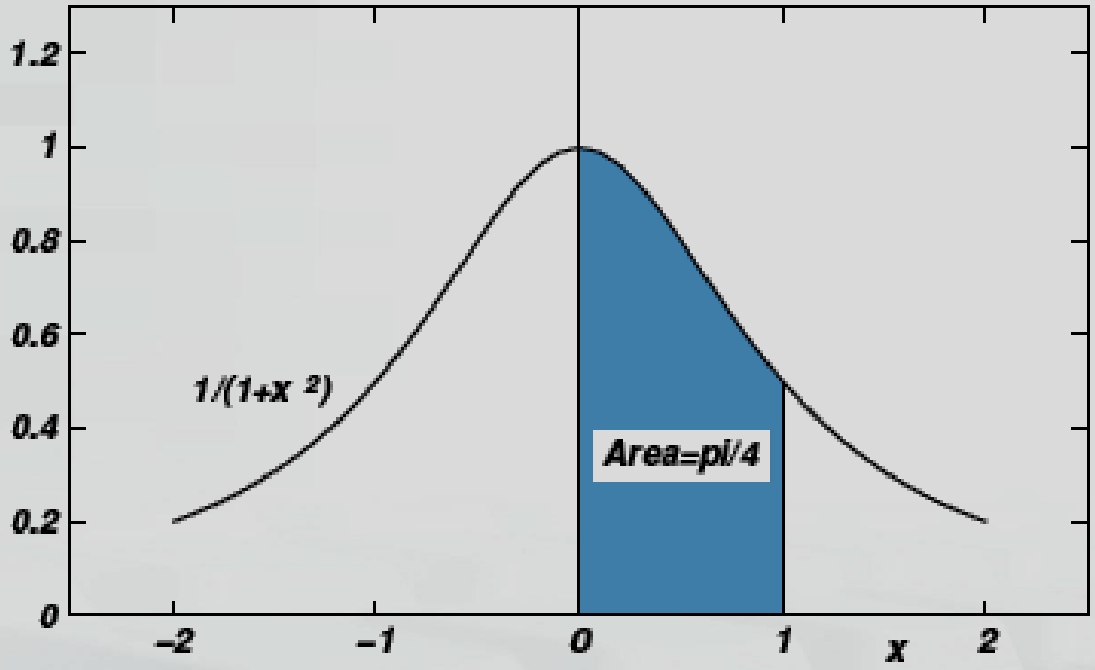
Salida del Ejemplo 13

- Después de la ejecución (mpirun –np 6 ejemplo13), la salida que produce el *ejemplo13* es:

```
Longitud de los vectores en cada proceso: 10000000
Longitud de los vectores a multiplicar: 60000000
Producto interno global: 60000000.000000
Tiempo: 3.683594
```

DSAP

Caso de estudio. Cálculo de PI



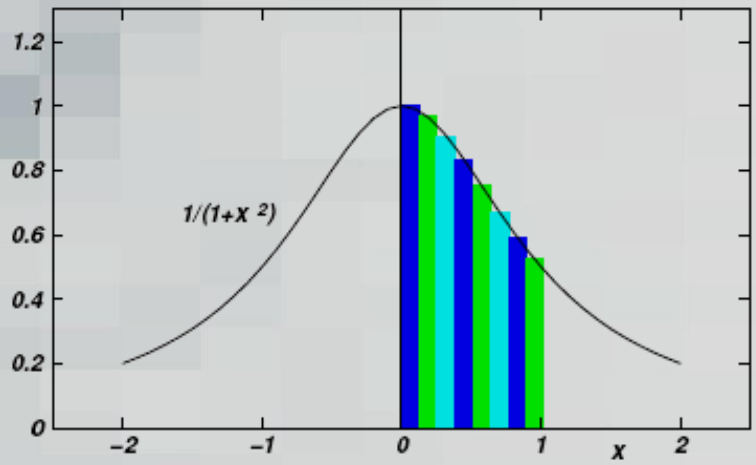
$$\operatorname{atan}(1) = \pi/4$$

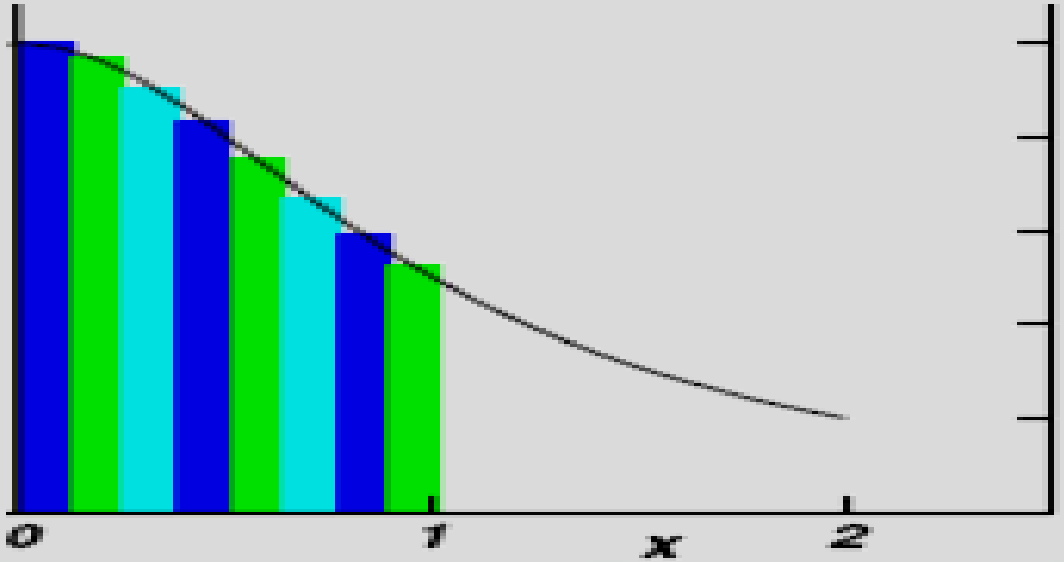
$$\frac{\mathrm{d}\operatorname{atan}(x)}{\mathrm{d}x} = \frac{1}{1+x^2}$$

$$\pi/4 = \operatorname{atan}(1) - \operatorname{atan}(0) = \int_0^1 \frac{1}{1+x^2}\,\mathrm{d}x$$

DSAP

Caso de estudio. Cálculo de PI

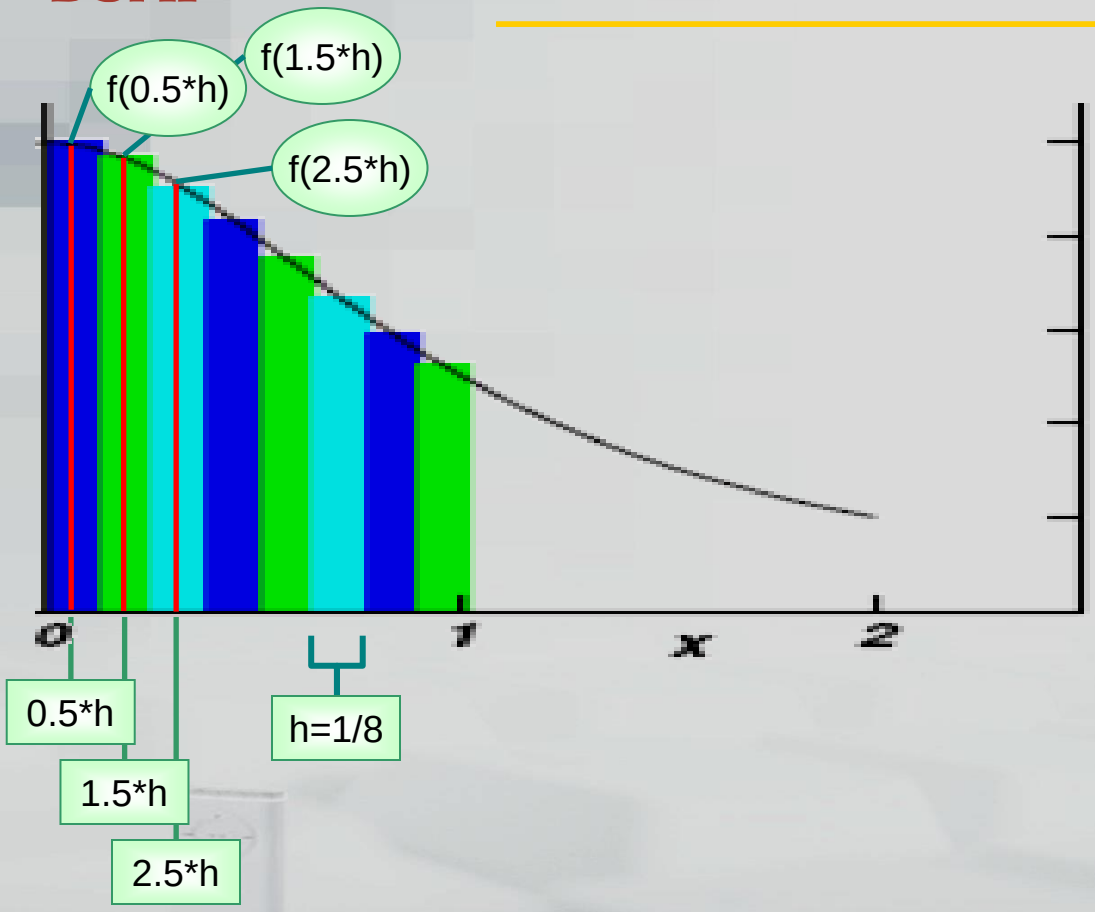




Dividimos el área en rectángulos para realizar la integración numérica

DSAP

Caso de estudio. Cálculo de PI




```
h = 1.d0 / n;
pi = 0;
for (i=1,i<=n;i++) {
    x = (i-0.5)*h;
    pi = pi + f(x);
}
pi = pi * h;
```

n=8 → h=1/8
Área 1er rectángulo: f(0.5*h) * h
Área 2º rectángulo: f(1.5*h) * h
Área 3er rectángulo: f(2.5*h) * h
...
Área 8º rectángulo: f(7.5*h) * h

pi = h * (f(0.5*h) + f(1.5*h) + f(2.5*h) + f(3.5*h) + f(4.5*h) + f(5.5*h) + f(6.5*h) + f(7.5*h))

Caso de estudio. Cálculo de PI

```
#define PI25DT 3.14159265358979323846264338
main(int argc, char **argv)
{
   int myrank, nprocs, i, n=500000000;
   double start_time, end_time, totalsec, total;
   double h, pi, x, parte;
   MPI_Status estado;
   double f(double x);

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   if (myrank == 0 && argc > 1)
       sscanf(argv[1], "%i", &n);
   if (myrank == 0)
       printf("Numero de intervalos: %d\n",n);
```

```
// Calculo secuencial
if (myrank == 0) {
   start_time = MPI_Wtime();
   h=1.0 / n;
   pi=0;
   for (i=1; i<=n; i++) {
         x   = (i-0.5)*h;
         pi  = pi + f(x);
   }
   pi = pi * h;
   end_time = MPI_Wtime();
   totalsec=end_time-start_time;
   printf("Calculo secuencial, PI = %26.24f\n", pi);
   printf("Calculo secuencial, Tiempo: %f\n", totalsec);
   printf("Calculo secuencial, Error: %26.24f\n\n",
              fabs(pi- PI25DT));
   pi=0;
}
```

Caso de estudio. Cálculo de PI

```
#define PI25DT 3.14159265358979323846264338
main(int argc, char **argv)
{
   int myrank, nprocs, i, n=500000000;
   double start_time, end_time, totalsec, total;
   double h, pi, x, parte;
   MPI_Status estado;
   double f(double x);

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   if (myrank == 0 && argc > 1)
       sscanf(argv[1], "%i", &n);
   if (myrank == 0)
       printf("Numero de intervalos: %d\n",n);
// Calculo paralelo
   start_time = MPI_Wtime();
   MPI_Bcast(&n,1,MPI_INT,0,
                    MPI_COMM_WORLD);
   h = 1.0 / n;
   parte = 0;
   for (i=myrank+1; i<=n; i=i+nprocs) {
       x     = (i-0.5)*h;
       parte = parte + f(x);
   }
   parte = parte*h;

   MPI_Reduce(&parte,&pi,1,MPI_DOUBLE,
              MPI_SUM,0,MPI_COMM_WORLD);
   end_time = MPI_Wtime();
   total=end_time-start_time;
   if (myrank == 0) {
       printf("Calculo paralelo, PI = %26.24f\n", pi);
       printf("Calculo paralelo, Tiempo: %f\n", total);
       printf("Calculo paralelo, Error: %26.24f\n",
                                    fabs(pi- PI25DT));
       printf("Speed-up: %5.2f,
              Eficiencia: %6.2f%\n",
              totalsec/total,
              ((totalsec/total)/nprocs)*100);
   }
   MPI_Finalize();
}
```

Salida del cálculo de PI

- Después de la ejecución (mpirun –np 4 pi), la salida que produce el ejemplo *pi* es:

```
Numero de intervalos: 500000000
Calculo secuencial, PI = 3.14159265358981398819082 6
Calculo secuencial, Tiempo: 8.332031
Calculo secuencial, Error: 0.000000000000020872192863

Calculo paralelo, PI = 3.14159265359001249606762 9
Calculo paralelo, Tiempo: 2.085938
Calculo paralelo, Error: 0.000000000000219380069666
Speed-up:  3.99, Eficiencia:  99.86%
```