Algoritmos Paralelos

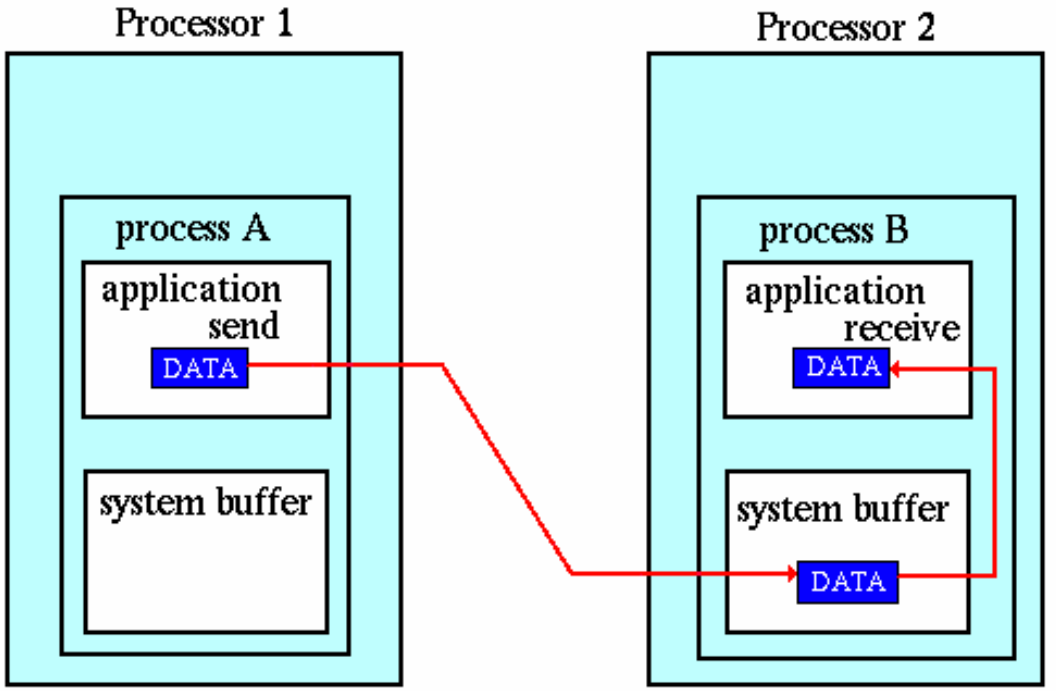
(clase 06.10.15)

Prof. J.Fiestas

Buffer (memoria):

Variables o vectores que se usan como argumentos de contenedores de mensajes en rutinas MPI

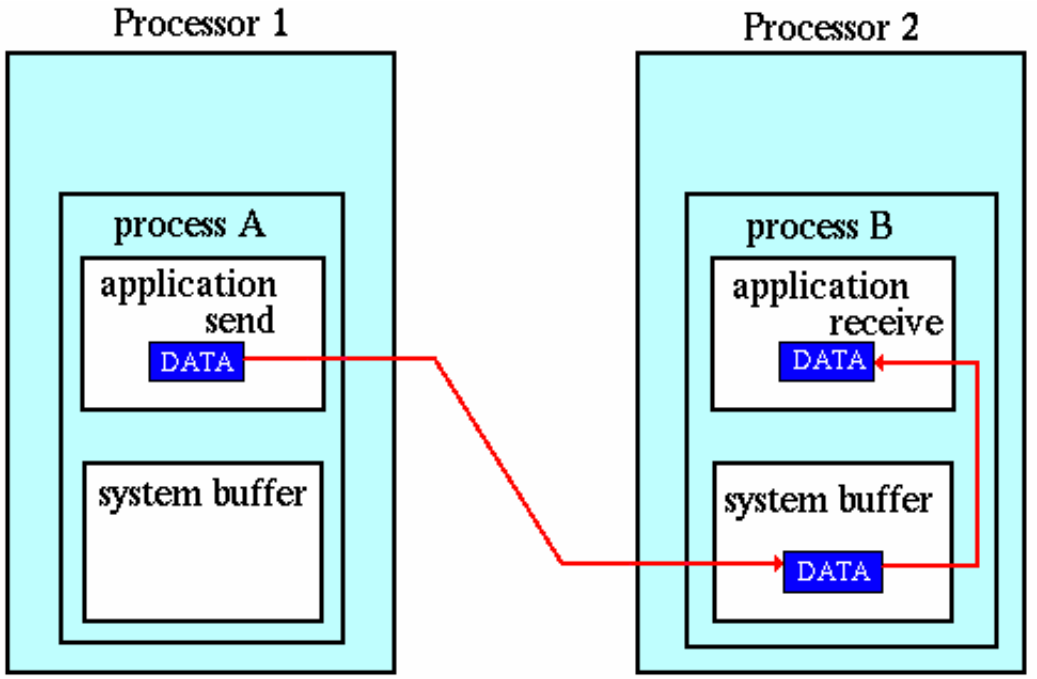
Camino de un mensaje grabado (buffered) en el proceso de recibo



Buffer (memoria):

Hay formas de especificar un espacio de buffer, i.e. MPI_Buffer_attach()
El espacio ofrecido por el sistema

Camino de un mensaje grabado (buffered) en el proceso de recibo



Modos de comunicación:

MPI_Send es un método de comunicación bloqueado, i.e. no retorna hasta que el mensaje ha sido recibido, tal que el enviante es libre de modificar el buffer.

Este usa el método de comunicación standard, en el cual, MPI decide si los mensajes son grabados en el buffer. En caso sean grabados, se completa el envio, antes que se realize el recibo del mensaje.

Modos de comunicación:

Buffered, que puede iniciarse y completarse independientemente de que se confirme el recibo del mensaje. En caso no se reciba, el mensaje se graba en el buffer (error si no hay suficiente espacio)

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

buf: dirección de envio del buffer
count: número de elementos enviados
datatype: tipo del elemento enviado
dest: rank de destino (integer)
tag: tag del mensaje (integer)
comm: communicador

Modos de comunicación:

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

Utilizando

int **MPI_Buffer_attach**(void *buffer, int size);

se puede manipular el tamaño del buffer. Donde size representa la cantidad deseada de memoria

Modos de comunicación:

Sincronizado (synchronous), que puede iniciarse independentemente del recibo del mensaje, pero se completa solo si se confirma que el mensaje ha sido recibido, para que el buffer pueda volver a ser usado: los procesos estan sincronizados

Analogia con el teléfono:
sincronizado – se contesta personalmente
no-sincronizado – se activa la contestadora

Modos de comunicación:

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

buf: dirección de envio del buffer
count: número de elementos enviados
datatype: tipo del elemento enviado
dest: rank de destino (integer)
tag: tag del mensaje (integer)
comm: communicador

Modos de comunicación:

Listo (ready), se inicia solo si se ha recibido el mensaje (si no, arroja error). Por consiguiente, el mensaje se enviara tan pronto como sea posible

int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

buf: dirección de envio del buffer
count: número de elementos enviados
datatype: tipo del elemento enviado
dest: rank de destino (integer)
tag: tag del mensaje (integer)
comm: communicador

Solo hay una correspondiente operación **MPI_Recv**

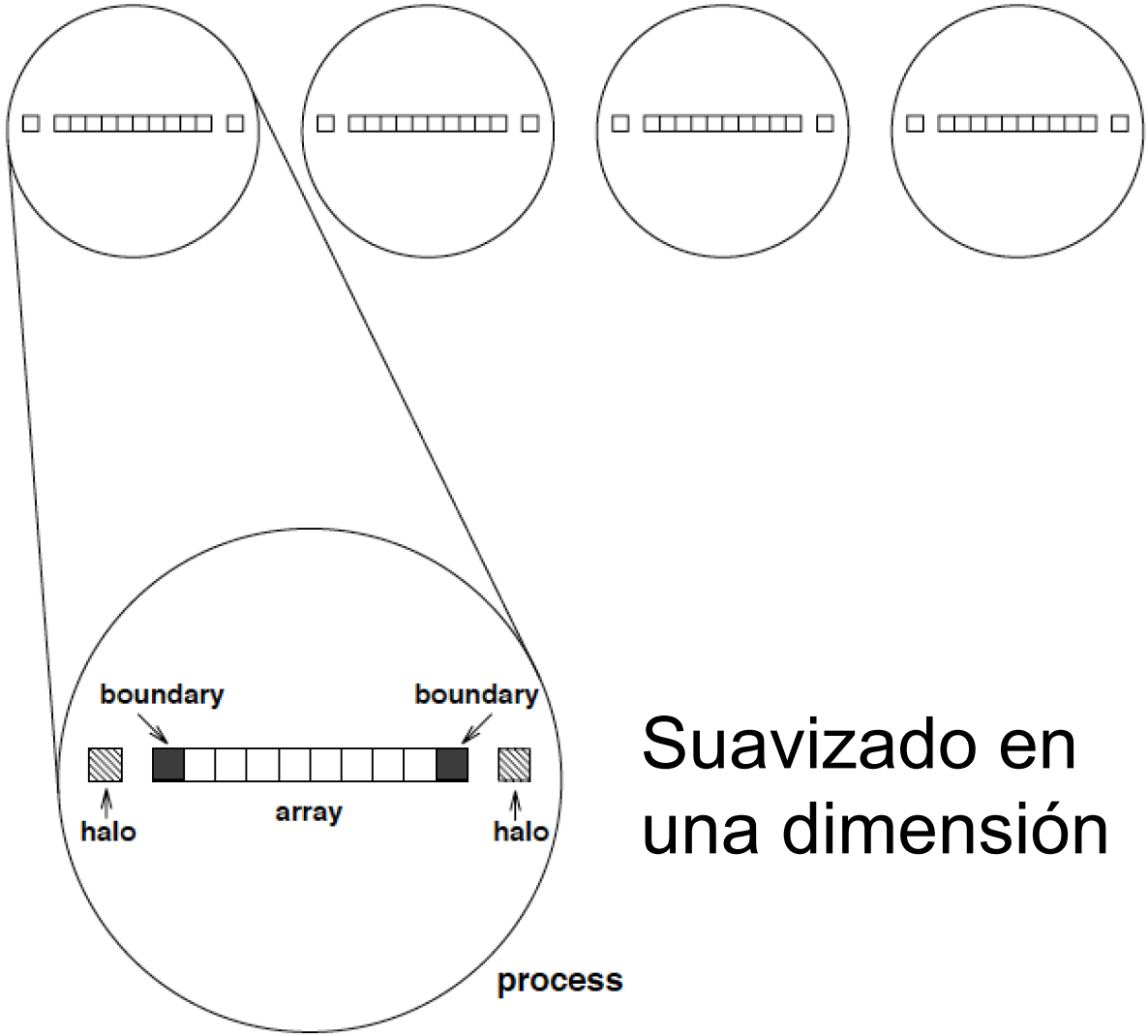
Modos de comunicación:

Sender mode	Notes	Synchronous?
Synchronous send	Message goes directly to receiver. Only completes when the receive has begun.	*synchronous*
Buffered send	Message is copied in to a "buffer" (provided by the application). Always completes (unless an error occurs), irrespective of receiver.	*asynchronous*
Standard send	Either synchronous or buffered (into a fixed size buffer provided by MPI system)	*both/hybrid*
Ready send	*Assumes* the matching receive has been called. Undefined behaviour (possibly an error) if the receiver is not ready.	*neither?*
Receive	Completes when a message has arrived	

Operation	MPI Call
Standard send	`MPI_Send`
Synchronous send	`MPI_Ssend`
Buffered send	`MPI_Bsend`
Ready send	`MPI_Rsend`
Receive	`MPI_Recv`

Comunicación no-bloqueada:

Operación de suavisado (**smoothing**) necesita promediar los vecinos inmediatos. Los extremos del sub-array necesitan comunicar sus valores a procesos vecinos



Suavizado en una dimensión

Comunicación no-bloqueada:

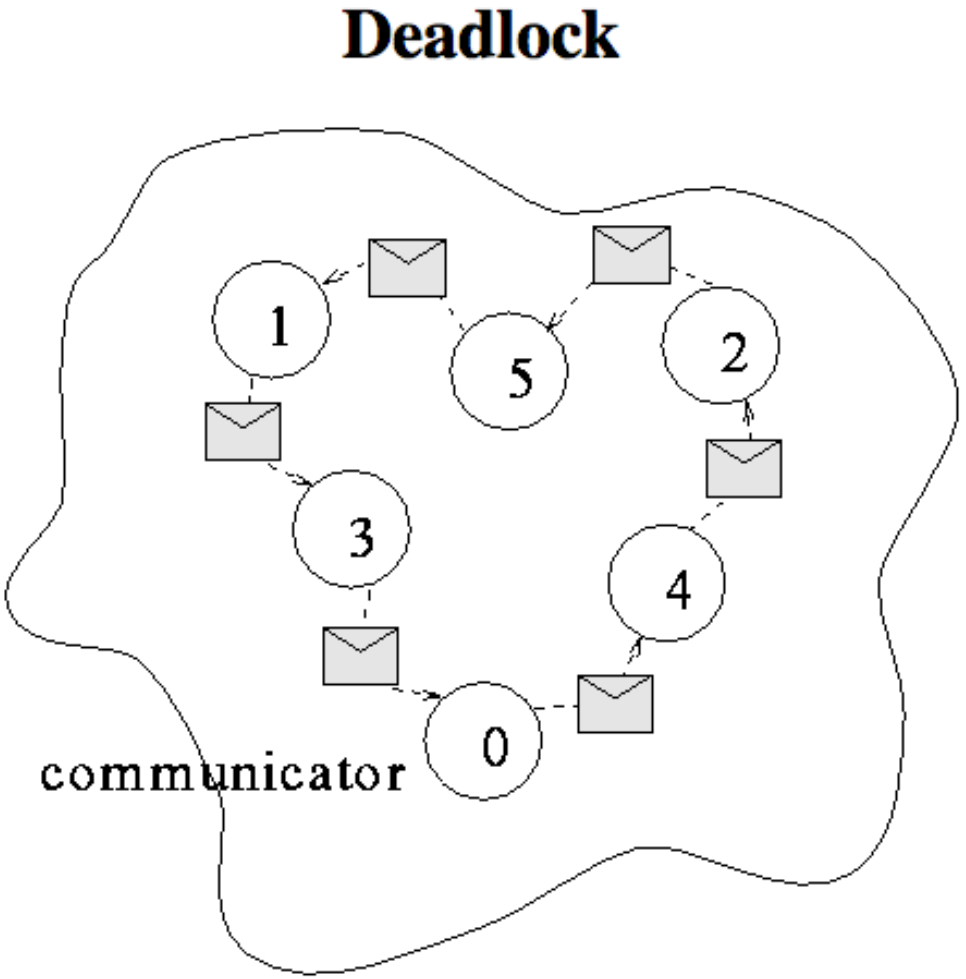
Hasta ahora, la comunicación no retorna ningún valor hasta que esta este terminada (cuando el buffer es libre de nuevo).
En el problema de suavizado sería:

```
for (iterations)
update all cells;
send boundary values to neighbours;
receive halo values from neighbours;
```

Comunicación bloqueada:

Causa retrasos de procesos a causa de otros que no culminan su trabajo.

Deadlock sucede cuando varios procesos esperan para poder continuar su trabajo



Comunicación no-bloqueada:

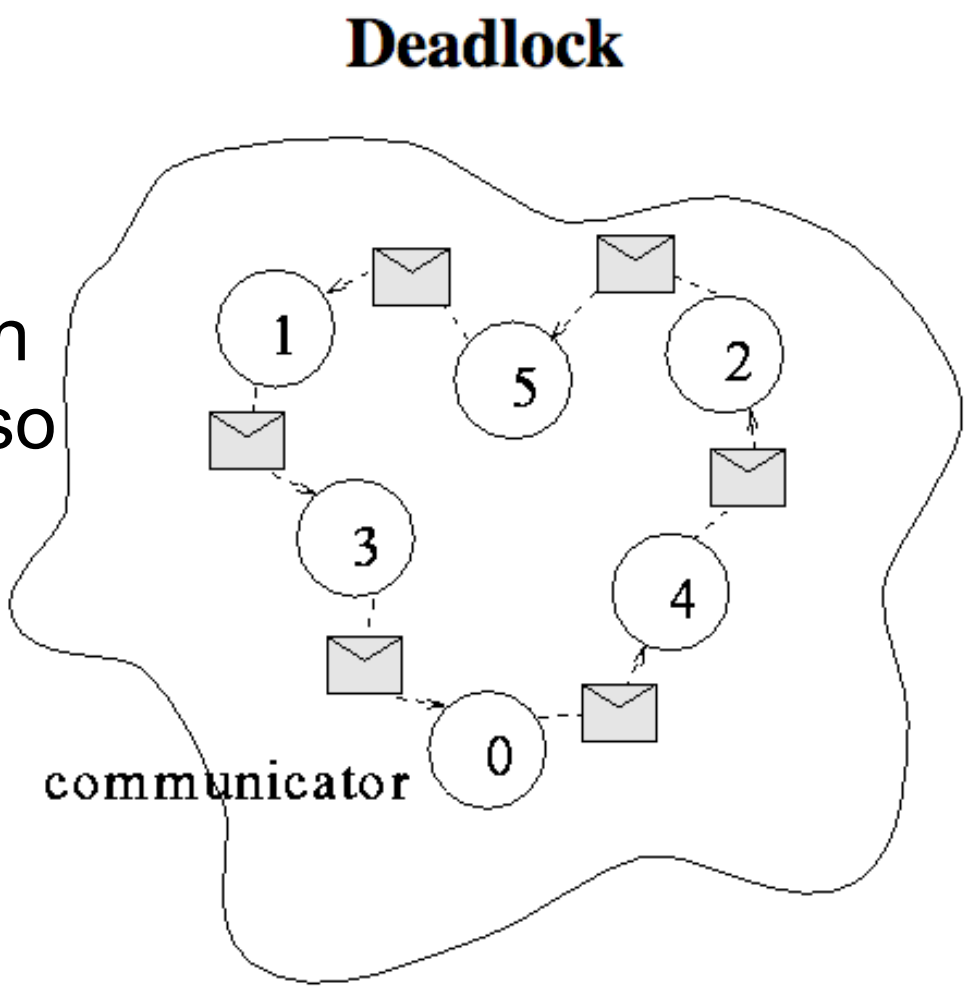
Se necesita separar el envío y recibo de información por los vecinos. Para ello, el Send y Recv retornan valores antes de completarse la comunicación, permitiendo hacerse cargo de otros envíos/recibos. Es decir, la comunicación tendrá ahora dos operaciones: de inicio y final

for(iterations)
update *boundary cells;*
initiate sending *of boundary values to neighbours;*
initiate receipt *of halo values from neighbours;*
update *non-boundary cells;*
wait for completion of sending *of boundary values;*
wait for completion of receipt *of halo values;*

Comunicación no-bloqueada

Ejecuta 3 fases:

- Inicializar la comunicación
- Seguir trabajando (incluso realizando otras comunicaciones)
- Esperar por la ejecucion de la comunicación no-bloqueada



Comunicación no-bloqueada:

Send no-bloqueado:

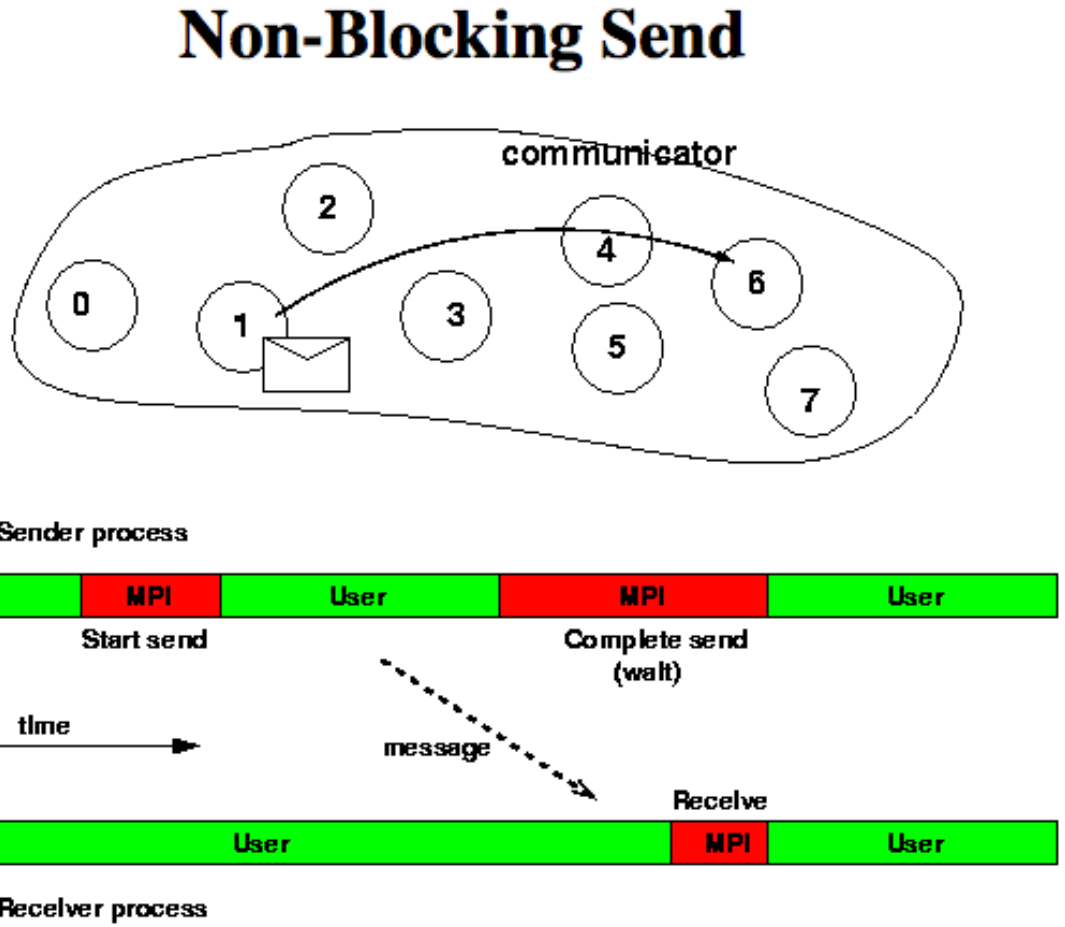
Analogia: pedir a la secretaria organize un meeting (se confirma luego y no se sabe si los participantes reciben la notificación)

Receive no-bloqueado:

Analogia: Esperar recibir regalos en tu cumpleaños, pero no poder pedirlos directamente ...

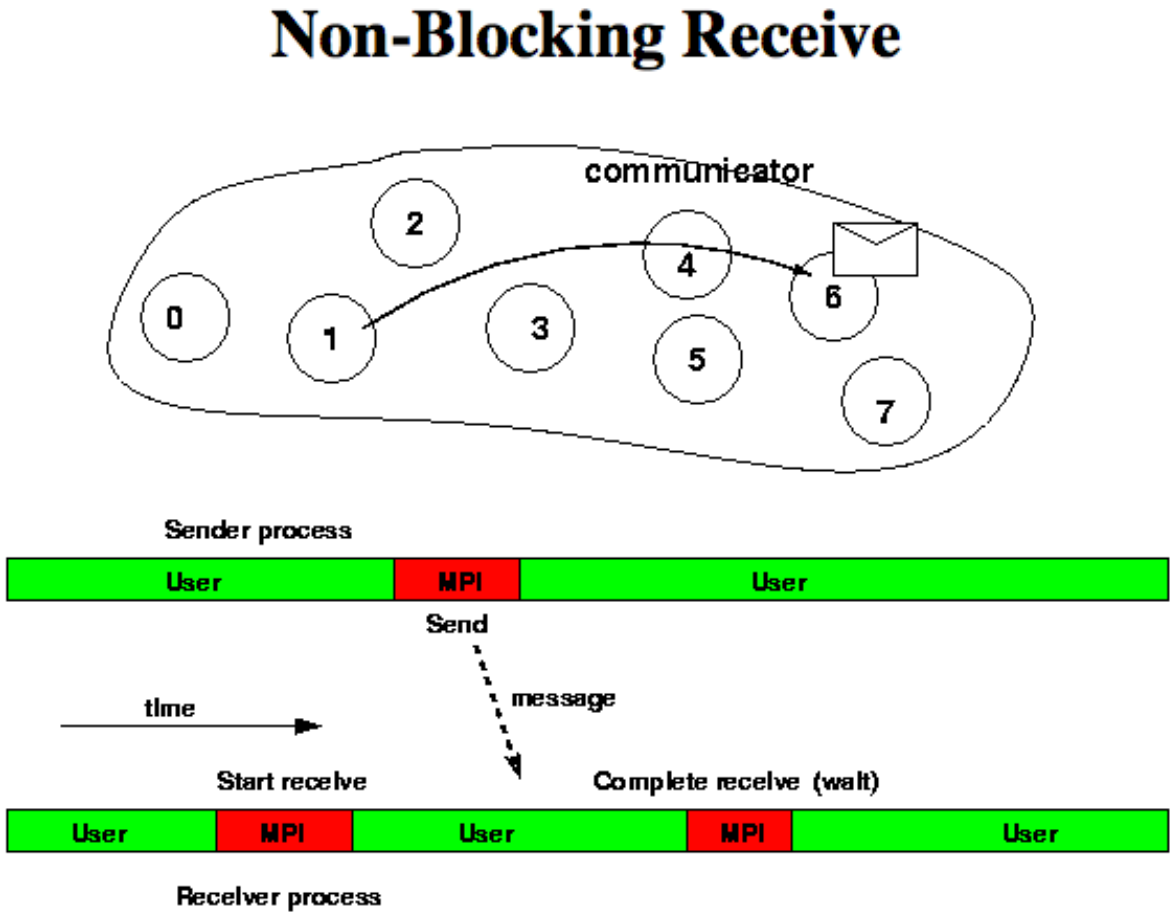
El envío se inicia con **MPI_Isend**, y continúa con otras comunicaciones

int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)



El recibo se inicia con **MPI_Irecv**, y continúa con otras comunicaciones

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)



Resumen Comunicación no-bloqueada:

Las operaciones **bloqueadas** solo retornan del procedimiento MPI cuando la operación (envio o recibo) se ha completado.
En caso de operaciones **no bloqueadas**, se retorna immediatamente antes de ser completado, i.e. El programa ejecuta la siguiente orden, mientras MPI hace la comunicación, y MPI confirma el fin de la operación mas tarde.

La comunicación no-bloqueada cuenta con **operaciones de espera** (matching wait), para que la memoria no se libere hasta que **wait** ha sido llamada

Conceptos para programar correcta y eficientemente en paralelo:

Sincronizado vs. no-sincronizado:

Describe el tiempo relativo a mensajes enviados o recibidos.

Bloqueado vs. No-bloqueado:

Describe fin de la operación en el que envia o recibe, independientemente

Test de término de comunicación

Es necesario saber si la comunicación terminó antes de utilizar el resultado del proceso o de re-utilizar el buffer. Hay dos tipos de control:

1. Wait type, que bloquean la comunicación hasta que culmine. Es equivalente a comunicación bloqueada

2. **Test type**, que retornan TRUE o FALSE dependiendo si la comunicación ha culminado o no, pero no bloquean.

Obtener información y controlar el proceso de comunicación a traves de:

```
int MPI_Wait(MPI_Request *req, MPI_Status *status);
```

Se declara luego de **ISend** y **Irecv**, los cuales detienen su operacion hasta que se completa la operación referenciada por Request ***req**. (manejo de la operacion - handle)
El estado de la operacion esta referenciado por ***status**

Obtener información y controlar el proceso de comunicación a traves de:

```
int MPI_Test(MPI_Request *req, int *flag,
MPI_Status *status);
```

Solo observa el fin de determinada operación.
Flag es un puntero a entero que contendra el resutado del test (true o false)

Comunicación múltiple

MPI prueba el término de comunicación múltiple controlando todas (***all***) las comunicaciones, cualquiera (***any***) de ellas, o algunas (***some***) de ellas

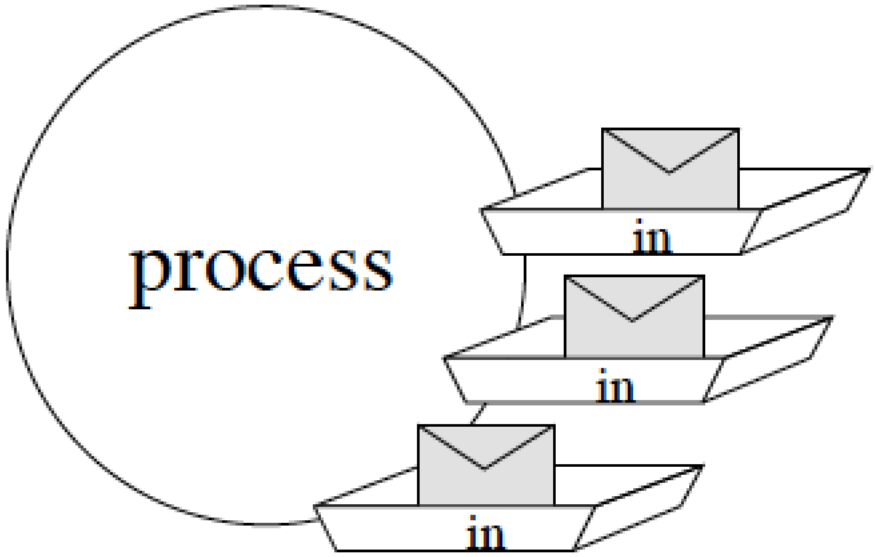
Test for completion	WAIT type (blocking)	TEST type (query only)
At least one, return exactly one	MPI_WAITANY	MPI_TESTANY
Every one	MPI_WAITALL	MPI_TESTALL
At least one, return all which completed	MPI_WAITSOME	MPI_TESTSOME

Término de todas las comunicaciones

int **MPI_Waitall**(int count, MPI_Request *req, MPI_Status *status);

Espera el final de todas las operaciones referenciadas por ***req**

count es el tamaño de la lista de req (y status), normalmente el numero de procesos



MPI_Testall (int count, MPI_Request *req, flag, MPI_Status *status)

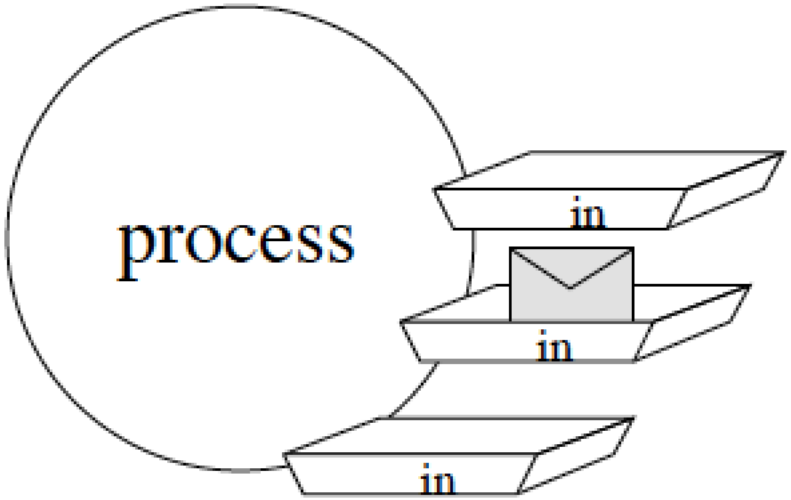
Si todos terminan, flag is TRUE

Término de cualquiera de las comunicaciones

int **MPI_Waitany**(int count, MPI_Request *req, index, MPI_Status *status);

Espera el final de alguna de las operaciones referenciadas por ***req**

count es el tamaño de la lista de req (y status), index da la posición del proceso terminado.



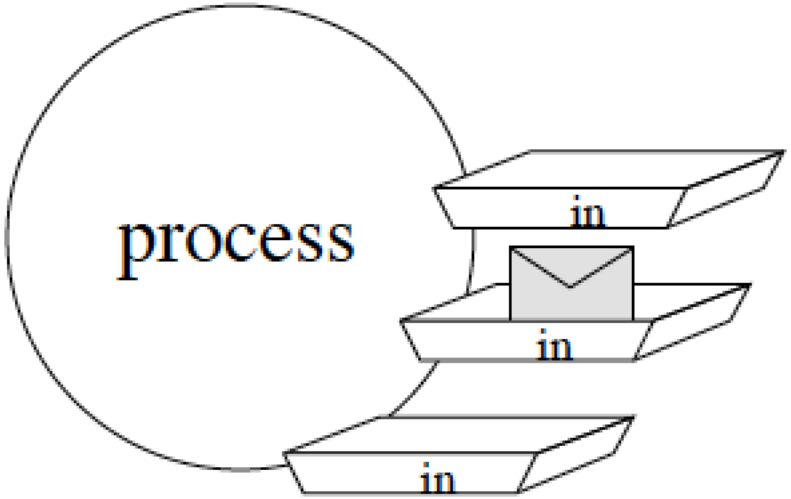
MPI_Testany (int count, MPI_Request *req, index, flag, MPI_Status *status)

flag (TRUE/FALSE) contiene el resultado del test

Término de alguna de las comunicaciones

int **MPI_Waitsome**(int count, MPI_Request *req,int outcount, *index, MPI_Status *status);

Similar a MPI_Waitany, MPI_Testany, pero retorna el status de todas las comunicaciones terminadas



MPI_Testsome (int count, MPI_Request *req, int outcount, *index, MPI_Status *status)

Ejemplo: proceso sincronizado y no-bloqueado

```
MPI_Request request;

MPI_Isend(buf, count,
datatype, dest, tag,
          comm, &request);

MPI_Wait(&request, &status);
```

```
MPI_Request request;

MPI_Irecv(buf, count,
datatype, src, tag,
          comm, &request);

MPI_Wait(&request, &status);
```

Bloqueado y no-bloqueado:

- Send y Recv pueden ser bloqueados o no-bloqueados
- Un Send bloqueado puede ser usado con un Recv no-bloqueado, y viceversa
- Send no-bloqueados pueden usar cualquier modo (sincronizado, buffered, standard, o ready)

Communication Modes

Non-Blocking Operation	MPI Call
Standard send	`MPI_Isend`
Synchronous send	`MPI_Issend`
Buffered Send	`MPI_Ibsend`
Ready send	`MPI_Irsend`
Receive	`MPI_Irecv`

Ejercicio 8:

Inicializar array, donde n= (1<<18)

```
if(world_rank==0){
        int c;
        srand(time(NULL));
        for(c = 0; c < n; c++) {
          original_array[c] = rand() % 100;
          }
        }
```

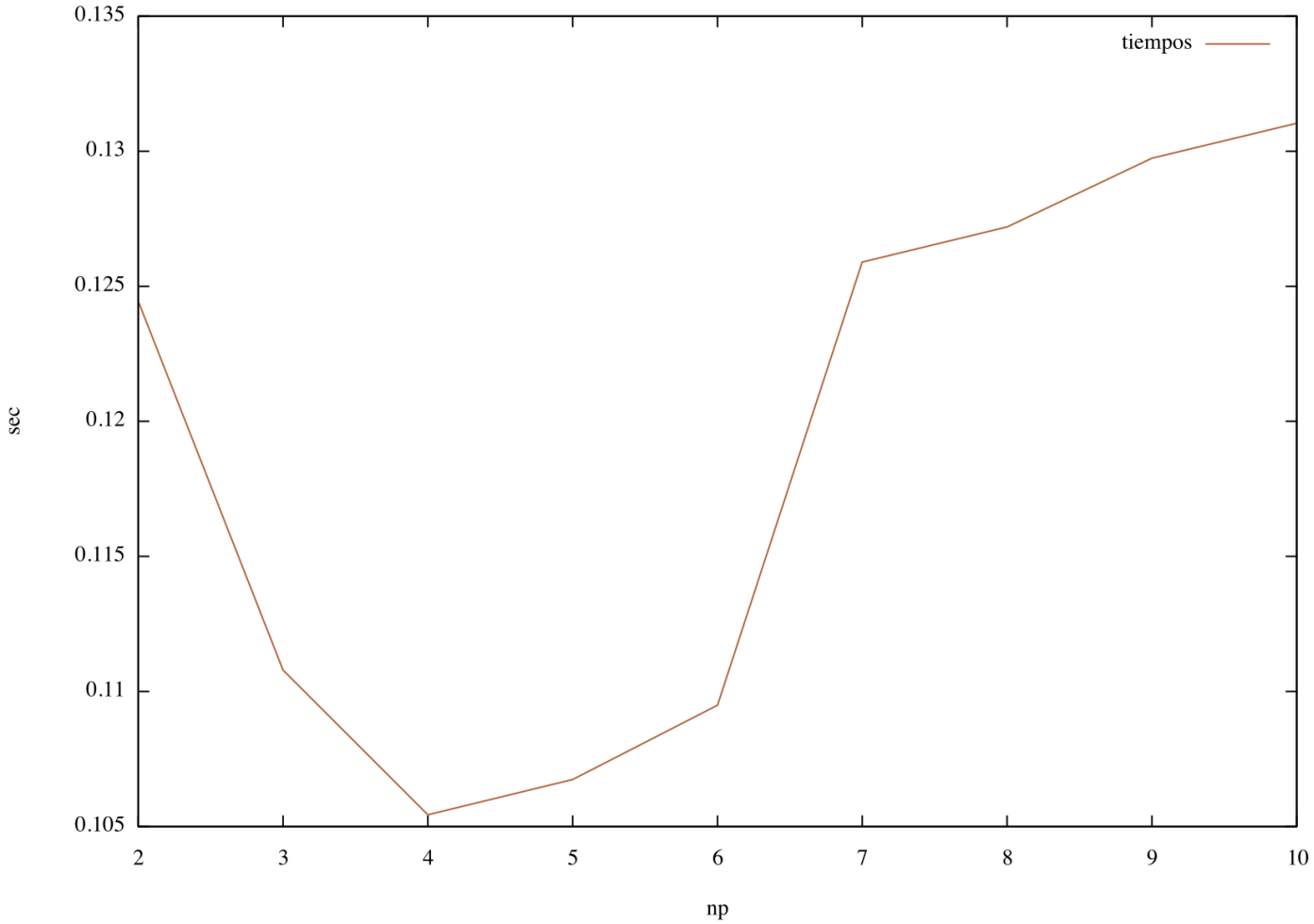
Ejercicio 8:

Repartir array[n] en sub_array[n/np], reunirlas y ordenar array[ ] de nuevo

```
int size = n/world_size;
/********** enviar cada sub-array a cada proceso **********/
int sub_array[size];
MPI_Scatter(original_array, size, MPI_INT, sub_array, size, MPI_INT, 0, MPI_COMM_WORLD);
mergeSort(sub_array, 0, size - 1);
/********** reunir las sub-arrays en una **********/
int sorted[n];
MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0, MPI_COMM_WORLD);

/********** ordenamiento final **********/
if(world_rank == 0) {
        mergeSort(sorted, 0, (n - 1));
}
MPI_Barrier(MPI_COMM_WORLD);

t2=MPI_Wtime();

if(world_rank == 0) {
        printf("Tiempo: %f \n",t2-t1);
}
```

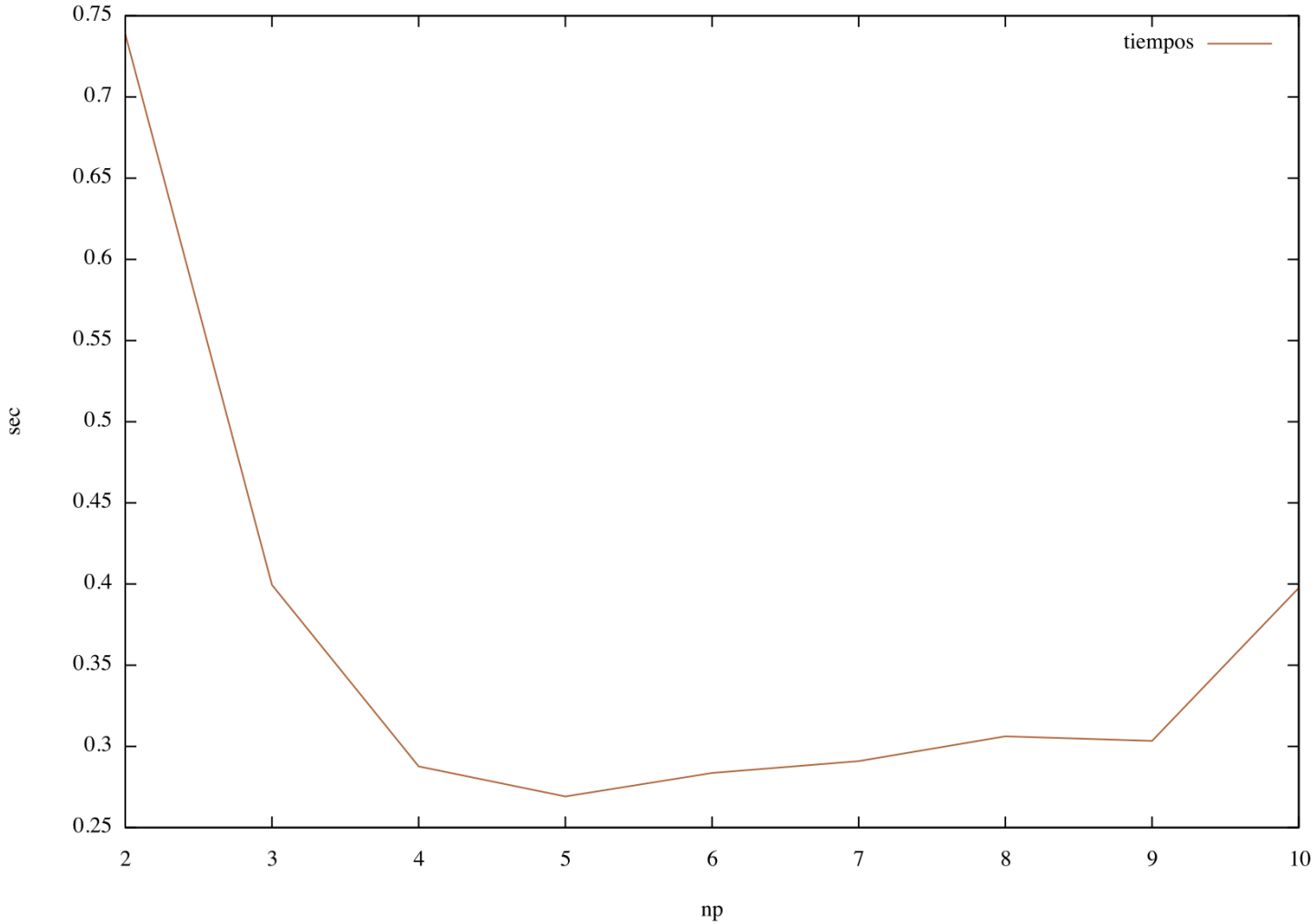
Ejercicio 8:

Medir tiempos para np=2,...,8



Ejercicio 9:

Medir tiempos para np=2,...,8



Ejercicio 10:

Código en serie con std rand()

```
static long num_trials = 10000000;

int main ()
{
   long i;  long Ncirc = 0;
   double pi, x, y, test;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   time_t t;

   srand(time(&t));

   for(i=0;i<num_trials; i++)
   {
      x  = (double)rand() / RAND_MAX;
      y  = (double)rand() / RAND_MAX;
      test = x*x + y*y;
      if (test <= r*r) Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %ld trials, pi is %lf %d \n",num_trials, pi, RAND_MAX);
    return 0;
}
```

Ejercicio 10:

Código en paralelo

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

srandom (taskid);

avepi = 0;
for (i = 0; i < ROUNDS; i++) {
   homepi = dboard(DARTS);

   rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM,
                   MASTER, MPI_COMM_WORLD);

   if (taskid == MASTER) {
      pi = pisum/numtasks;
      avepi = ((avepi * i) + pi)/(i + 1);
      printf("   After %8d throws, average value of pi = %10.8f\n",
              (DARTS * (i + 1)),avepi);
   }
}
if (taskid == MASTER)
   printf ("\nReal value of PI: 3.1415926535897 \n");

MPI_Finalize();
```

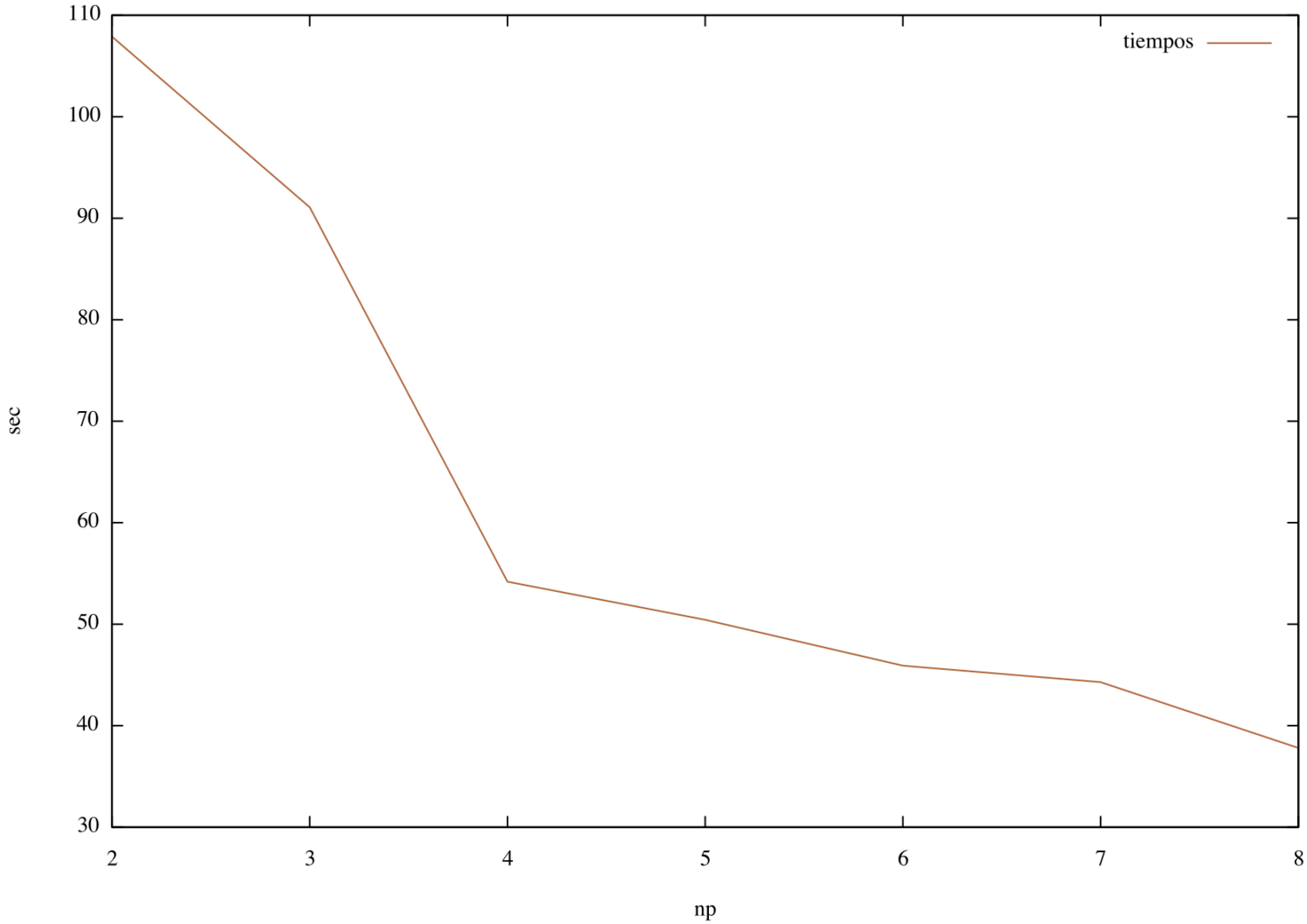
Ejercicio 10:

Código en paralelo: funcion dboard()

```
double dboard(int darts)
{
#define sqr(x)  ((x)*(x))
long random(void);
double x_coord, y_coord, pi, r;
int score, n;
unsigned int cconst;  /* must be 4-bytes in size
   /* 2 bit shifted to MAX_RAND later used to s
   cconst = 2 << (31 - 1);
   score = 0;

   /* "throw darts at board" */
   for (n = 1; n <= darts; n++)  {
      r = (double)random()/cconst;
      x_coord = (2.0 * r) - 1.0;
      r = (double)random()/cconst;
      y_coord = (2.0 * r) - 1.0;

      if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
           score++;
      }

pi = 4.0 * (double)score/(double)darts;
return(pi);
}
```

Ejercicio 10:

Medir tiempos para np=2,...,8



Ejercicio 1: ALGORITMO DE PING-PONG

Programar el algoritmo de Ping-Pong, por el que utilizando MPI_Send y MPI_Recv, se manda un mensaje de ida y venida entre dos procesos
Utilizar un vector de N elementos como mensaje, y un numero de iteraciones T
Variar N y T para medir un tiempo significativo de ejecución, utilizando MPI_Time()

Utilizar MPI_Ssend() en el mismo problema, y variar el tamaño del mensaje (N) para estimar el tamaño del buffer

Reemplazar MPI_Send() por MPI_Bsend() e indicar en que casos el programa no ejecuta.
Utilizar MPI_Buffer_Attach(buffer,size) para modificar el tamaño del buffer

Comparar los tiempos de ejecución utilizando MPI_Send(), MPI_Bsend() y MPISsend()

Ejercicio 2:
Programe el algoritmo del anillo. Similar al problema 3 del examen parcial, con la diferencia que el ultimo proceso, debe comunicarse con el proceso 0 antes de terminar el algoritmo.
Cada proceso debe acumular la suma de usuarios (1000+100*rank)

Utilizar comunicacion no-bloqueada (Isend, Irecv) e imprimir los resultados acumulados de cada proceso con np=5

Ejercicio 3:
Programar el cálculo del promedio de un array de 100 floats, aleatorios, y en el rango de [0,10], utilizando comunicacion no-bloqueada (MPI_Isend, MPI_Irecv)
Incrementar el tamaño del array (N=1000000) para medir tiempos significativos de proceso y comparar con los tiempos utilizando comunicacion bloqueada (MPI_Send, MPI_Recv)