MESSAGE PASSING INTERFACE MPI (PARTE A)

Adaptado de transparencias de Thomas Sterling y Chirag Dekate

Temas

- Introducción
- El estándar MPI
- El modelo MPI-1 y llamadas básicas
- MPI Communicators
- Comunicaciones punto a punto
- Deadlock
- Caso de estudio: Regla del Trapecio

Introducción

- Contexto: computadoras paralelas con memoria distribuida
- Procesos secuenciales que se comunican entre sí, cada uno con su propia memoria y sin acceso a la memoria de los demás.
  - Los procesos interactúan (intercambio de datos, sincronización) mediante paso de mensajes.
  - Inicialmente, cada fabricante tenía sus propias bibliotecas.
  - La primera estandarización fue PVM
    - Comienzo en 1989, primera edición pública en 1991
    - Trabaja bien en máquinas distribuidas
  - La siguiente fue MPI

El estándar MPI

- Entre 1992 y 1994, una comunidad formada por fabricantes y usuarios decidió crear una interface estándar para paso de mensajes en el contexto de computadoras paralelas de memoria distribuida (En ese momento, MPPs).

- El resultado fue MPI-1
  - Es una API
  - *Bindings* para FORTRAN77 y C.
  - Implementación de referencia (mpich).
  - Los fabricantes pueden definir su propia implementación.

El estándar MPI

- Desde entonces
  - MPI-1
  - MPI-2
    - Extendió MPI
    - Agregó nuevas funcionalidades
    - *Bindings* para FORTRAN90 y C++
- Referencia
  - http://www.mpi-forum.org/

MPI : Conceptos Básicos

- Todo programa MPI debe contener la directiva

```
#include "mpi.h"
```

- El archivo mpi.h contiene las definiciones y declaraciones necesarias para compilar un programa MPI.
- Se encuentra en el directorio “include” de la mayoría de las instalaciones MPI.

```
...
#include “mpi.h”
...
MPI_Init(&Argc,&Argv);
...
...
MPI_Finalize();
...
```

MPI: Inicialización

Función: MPI_init()

int MPI_Init(int *argc, char ***argv)

Descripción:

Inicializa el ambiente de ejecución MPI. MPI_init() debe ser invocada antes que cualquier otra función MPI, y debe ser llamada sólo una vez. ***argc*** es un puntero al número de argumentos y ***argv*** es un puntero al arreglo de argumentos. A la salida de la función, todos los procesos tendrán una copia de la lista de argumentos..

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Init.html

MPI: Finalización

Función: MPI_Finalize()

int MPI_Finalize()

Descripción:

Termina el ambiente de ejecución MPI. Todos los procesos MPI deben invocar a esta función antes de salir. No es necesario que sea la última secuencia ejecutable, y ni siquiera que se encuentre en la función **main**, pero debe ser llamada en algun punto después de la última llamada a cualquier otra función MPI.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Finalize.html

“Hola Mundo” en MPI

- Código fuente C para un Hola Mundo en MPI

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

int main( int argc, char *argv[])
{
    MPI_Init( &argc, &argv);
    printf("Hola, Mundo!\n");
    MPI_Finalize();
    return 0;
}
```

Incluir archivos de cabecerra

Inicializar el contexto MPI

Finalizar el contexto MPI

Compilar un programa MPI

- Utilizando las bibliotecas
  - El usuario sabe donde está el archivo cabecera y las bibliotecas, y se lo indica al compilador:

```
gcc -Iheaderdir -Llibdir mpicode.c –lmpich
```

- Usando un *wrapper*
  - Lo mismo, pero el usuario no necesita conocer los detalles:

```
mpicc -o ejecutable mpicode.c
```

Se puede utilizar cualquiera de los métodos, pero no ambos.

Por ejemplo, en una instalación de mpich-shmem en Linux puede ser:

```
gcc -I/usr/lib/mpich-shmem/include
-L/usr/lib/mpich-shmem/lib –o hello
hello.c -lmpich-shmem
```

O

```
mpicc -o hello hello.c
```

Ejecutar un programa MPI

- Algún número de procesos se ejecuta en alguna parte
  - El estándar no lo aclara
  - La implementación y la interface son variables
  - Por lo general, un comando de tipo *mpirun* lanza un número de copias de un ejecutable según algún mapeo.
  - Ejemplo:

    ‘`mpirun -np 2 ./a.out`’ ejecuta dos copias de `./a.out`
  - **La mayoría de los sistemas de supercómputo en producción envuelven el comando *mpirun* con scripts de más alto nivel que interactúan con un sistema de planificación, como PBS o Load Leveler para un manejo eficiente de los recursos.**
  - **Ejemplos :**

PBS File:

```
#!/bin/bash
#PBS -l walltime=120:00:00,nodes=8:ppn=4
cd /home/miguel/demo1/
pwd
date
mpirun -np 32 -machinefile $PBS_NODEFILE ./demo
date
```

LoadLeveler File:

```
#!/bin/bash
#@ job_type = parallel
#@ job_name = DEMO
#@ wall_clock_limit = 120:00:00
#@ node = 8
#@ total_tasks = 32
#@ initialdir = /home/miguel/datos
#@ executable = /usr/bin/demo
#@ arguments = -x
#@ queue
```

Ejecutando el programa

- Usando mpirun :

```
 mpirun -np 8  ./hello
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
```

- Usando PBS

hello.pbs :

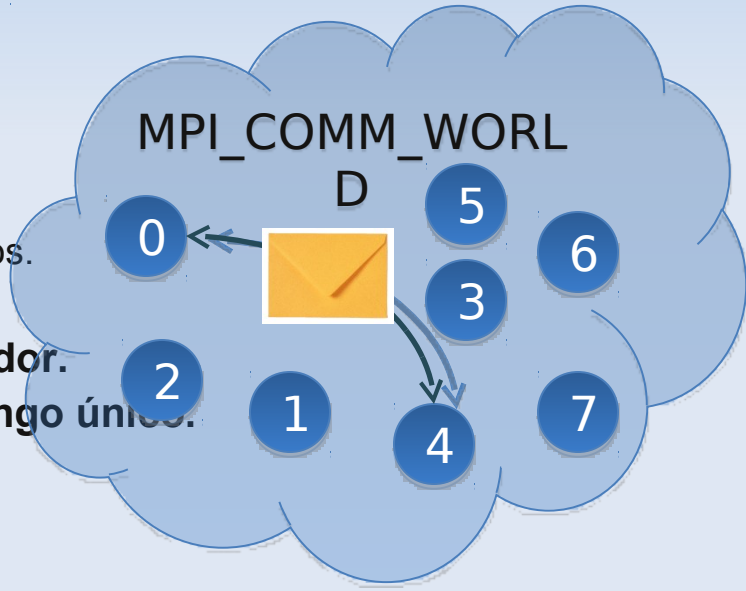
```
#!/bin/bash
#PBS -N hello
#PBS -l walltime=00:01:00,nodes=2:ppn=4
cd /home/miguel/mpi/demos/
pwd
date
mpirun -np 8 -machinefile $PBS_NODEFILE ./hello
date
```

more hello.o10030

```
/home/miguel/mpi/demos
mié mar 10 16:05:05 ART 2010
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
Hola, Mundo!
mié mar 10 16:05:08 ART 2010
```

Comunicadores MPI

- **Un Comunicador (Communicator) es un objeto interno**
- **Los programas MPI están compuestos por procesos que se comunican**
- **Cada proceso tiene su propio espacio de direcciones, con sus propios atributos (rank, size, argc, argv, etc.)**
- **MPI provee funciones que permiten interactuar.**
- **El comunicador por defecto es MPI_COMM_WORLD**
  - Todos los procesos pertenecen a él.
  - Tiene un tamaño (el número de procesos).
  - Todos los procesos tienen un rango dentro de él.
  - Puede considerarse una lista ordenada de todos los procesos.
- **Pueden existir comunicadores adicionales.**
- **Un proceso puede pertenecer a más de un comunicador.**
- **Dentro de un comunicador, cada proceso tiene un rango único.**



Tamaño de un comunicador

Función: MPI_Comm_size()

int MPI_Comm_size (MPI_Comm comm, int *size)

Descripción:

Determina el tamaño del grupo asociado a un comunicador (comm). En la lista de argumentos, *comm* se refiere al comunicador a ser interrogado, y el resultad se almacena en la variable *size*.

```
...
#include "mpi.h"
...
int size;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Comm_size.html

Rango de un proceso

Función: MPI_Comm_rank()

int MPI_Comm_rank (MPI_Comm comm, int *rank)

Descripción:

Devuelve el rango del proceso que lo invoca en el grupo asociado con el comunicador. El parámetro *comm* en la lista de argumentos es el comunicador a ser interrogado, y se almacena el resultado en *rank*.

```
...
#include "mpi.h"
...
int rank;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Comm_rank.html

Ejemplo : comunicadores

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

int main( int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("Hola, Mundo! desde %d de %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Determina el tamaño del comunicador MPI_COMM_WORLD

Determina el rango del proceso actual dentro de MPI_COMM_WORLD

```
...
Hola, Mundo! desde 1 de 8
Hola, Mundo! desde 0 de 8
Hola, Mundo! desde 5 de 8
...
```

Ejemplo : Size & Rank

- Compiling :

  mpicc -o hello2 hello2.c

- Result :

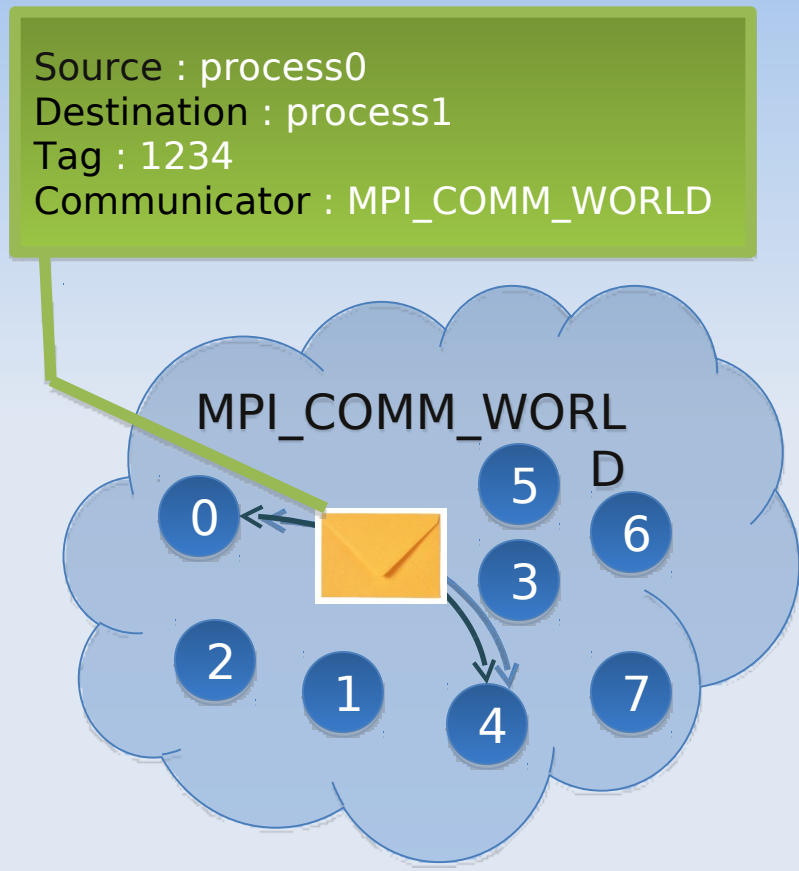
  Hola, Mundo! desde 4 de 8
  Hola, Mundo! desde 3 de 8
  Hola, Mundo! desde 1 de 8
  Hola, Mundo! desde 0 de 8
  Hola, Mundo! desde 5 de 8
  Hola, Mundo! desde 6 de 8
  Hola, Mundo! desde 7 de 8
  Hola, Mundo! desde 2 de 8

MPI : Primitivas de comunicación punto a punto

- La *comunicación punto a punto* de MPI es un mecanismo básico de comunicación en la cual un proceso envía datos y otro los recibe.
- Hay dos funciones principales que realizan el paso de mensajes en MPI:
  - MPI_Send – Envía un mensaje a un proceso determinado
  - MPI_Recv – Recibe un mensaje de un proceso
- Ambas llamadas intercambian los datos requeridos por los programas, más información adicional.
- El *sobre* (envoltura) del mensaje contiene la siguiente información:
  - El rango del receptor
  - El rango del emisor
  - Un *tag*
  - Un comunicador
- El argumento que contiene el origen sirve para distinguir entre los mensajes enviados por distintos procesos.
- Tag es de tipo *int* y permite distinguir entre los mensajes enviados por un mismo proceso.

El sobre (Envelope

- La comunicación entre procesos se realiza mediante mensajes.
- Cada mensaje consiste en un númer fijo de campos (Message Envelope) :
  - _ Envelope = **source, destination, tag, communicator**
  - _ Message = Envelope + Data
- Communicator es el espacio de nombres asociado con un grupo de procesos relacionados



MPI: (blocking) Send

Función: MPI_Send()

```
int MPI_Send(
        void            *message,
        int             count,
        MPI_Datatype    datatype,
        int             dest,
        int             tag,
        MPI_Comm        comm )
```

Descripción:

El contenido del mensaje se almacena en un bloque de memoria referenciado por el primer parámetro, *message*. Los siguientes dos parámetros, *count* y *datatype*, permiten que el sistema determine cuánto espacio se requiere para el mensaje: el mensaje contiene una secuencia de *count* valores, cada uno de los cuales es del tipo *MPI datatype.* Para ser recibido, el receptor debe haber reservado el espacio suficiente. De no ser así, se producirá un error de *overflow*. El parámetro *dest* corresponde al rango del destinatario.

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Send.html

MPI : Tipos de datos

Tipo de MPI	Tipo de 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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Es posible definir tipos derivados (derived datatypes), tales como arreglos o estructuras

MPI: (blocking) Receive

Función: MPI_Recv()

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

Descripción:

El contenido del mensaje se almacena en un bloque de memoria referenciado por el primer parámetro, *message*. Los siguientes dos parámetros, *count* y *datatype*, permiten que el sistema determine cuánto espacio se requiere para el mensaje: el mensaje contiene una secuencia de *count* valores, cada uno de los cuales es del tipo *MPI datatype*. Para ser recibido, el receptor debe haber reservado el espacio suficiente. De no ser así, se producirá un error de *overflow*. El parámetro *source* corresponde al rango del proceso emisor. El parámetro MPI_Status devuelve información acerca de los datos que efectivamente se recibieron.

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Recv.html

MPI_Status object

Objeto: MPI_Status

Uso :

```
MPI_Status    status;
```

Descripción:

El objeto MPI_Status se usa en las funciones de recepción para devolver datos acerca del mensaje. Específicamente, el objeto contiene la identificación del proceso emisor (MPI_SOURCE), el tag (MPI_TAG), y el estado de errror (MPI_ERROR) .

```
#include "mpi.h"
...
MPI_Status status; /* return status for    */
...
MPI_Init(&argc, &argv);
...
if (my_rank != 0) {
...
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  else { /* my rank == 0 */
    for (source = 1; source < p; source++ ) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &status);
...
  MPI_Finalize();
...
```

MPI : Ejemplo send/recv

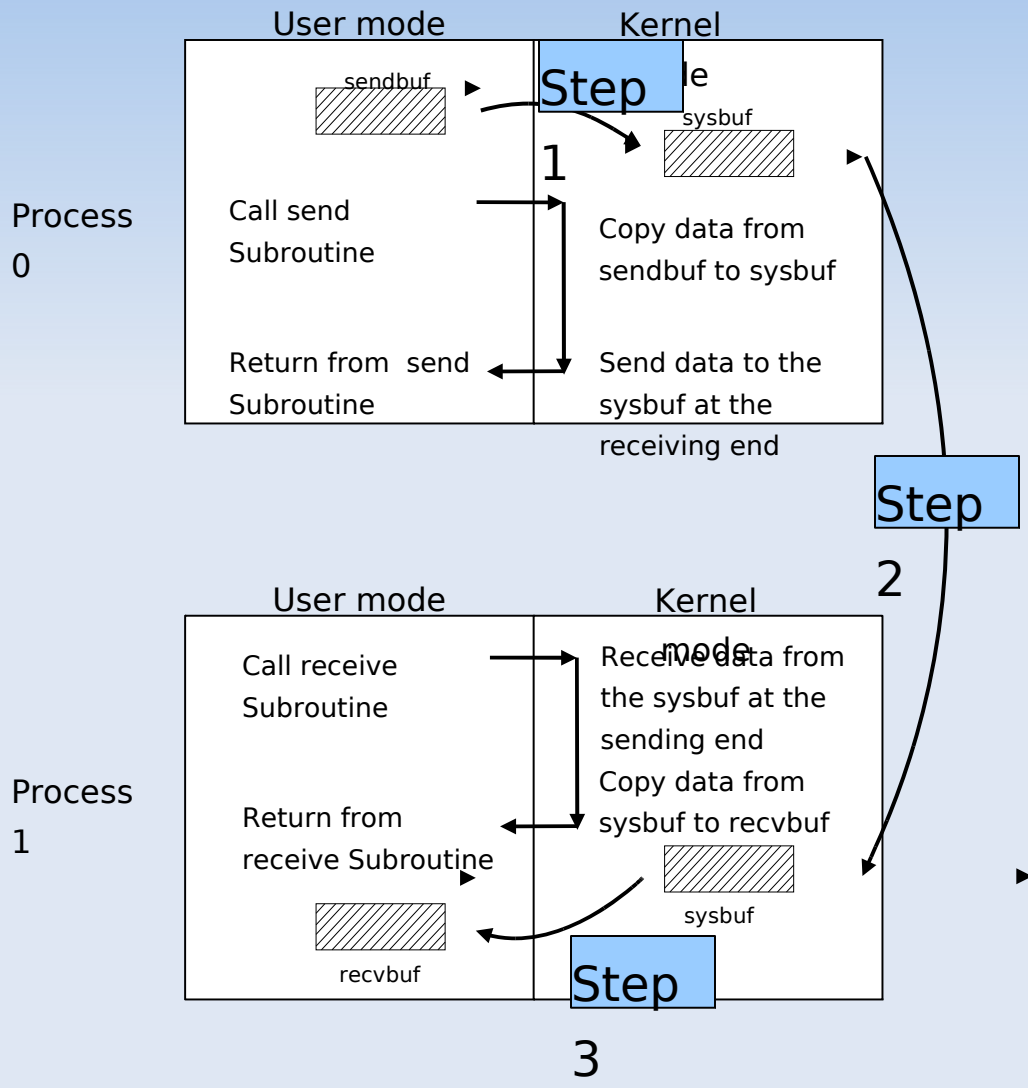
```
/* hello world, MPI style */

#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
  int my_rank;       /* rank of process     */
  int p;             /* number of processes */
  int source;        /* rank of sender      */
  int dest;          /* rank of receiver    */

  int tag=0;         /* tag for messages    */
  char message[100]; /* storage for message */
  MPI_Status status; /* return status for   */
                     /* receive             */

  /* Start up MPI */
  MPI_Init(&argc, &argv);

  /* Find out process rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  /* Find out number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &p);

 if (my_rank != 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!",
my_rank);
    dest = 0;
    /* Use strlen+1 so that \0 gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
dest, tag, MPI_COMM_WORLD);
  }
  else { /* my rank == 0 */
    for (source = 1; source < p; source++ ) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &status);
      printf("%s\n", message);
    }
    printf("Greetings from process %d!\n", my_rank);
  }

  /* Shut down MPI */
  MPI_Finalize();

} /* end main */
```

Src : Prof. Amy Apon

Comunicación punto a punto

- Permite interactuar a dos procesos
- Es el tipo de comunicación mas flexible en MPI
- Dos variedades
  - Bloqueante y no bloqueante
- Dos funciones básicas
  - Send y receive
- Con estas dos funciones, más las cuatro ya vistas, se puede hacer todo
  - Hay funciones que nos brindan mejores formas de hacer las cosas

Comunicación punto a punto : (buffered)

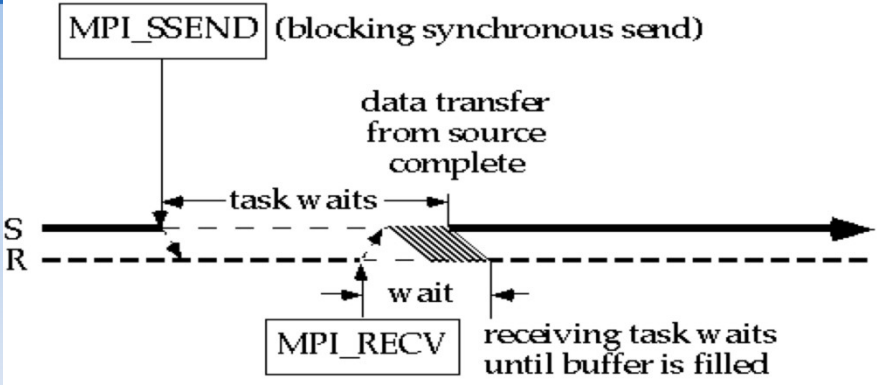


1. Los datos enviados por el usuario se copian del espacio de usuario al buffer del sistema
2. El dato se envía desde el buffer del sistema, a través de la red, hasta el buffer del sistema del receptor
3. El proceso receptor copia los datos del buffer del sistema a la memoria en espacio de usuario.

Modos de comunicación

- MPI ofrece varios modos de comunicación, que afectan el manejo de datos y el rendimiento:
  - Buffered
  - Ready
  - Standard
  - Synchronous
- Cada modo de comunicación tiene primitivas bloqueantes y no bloqueantes
  - En la comunicación punto a punto bloqueante, la llamada a send bloquea hasta que el bloque enviado pueda ser utilizado. De manera similar, la función receive bloquea hasta que el buffer ha obtenido el contenido del mensaje en forma exitosa.
  - En la comunicación punto a punto no bloqueante las llamadas send y receive permiten la superposición de comunicación y computación. La comunicación suele hacerse en dos fases: envío y testeo de completamiento.
- **Sobrecarga de sincronización**: tiempo gastado esperando la ocurrencia de un evento en otra tarea.
- **Sobrecarga del sistema**: Tiempo gastado copiando el mensaje desde el buffer del emisor a la red y desde la red al buffer del receptor.

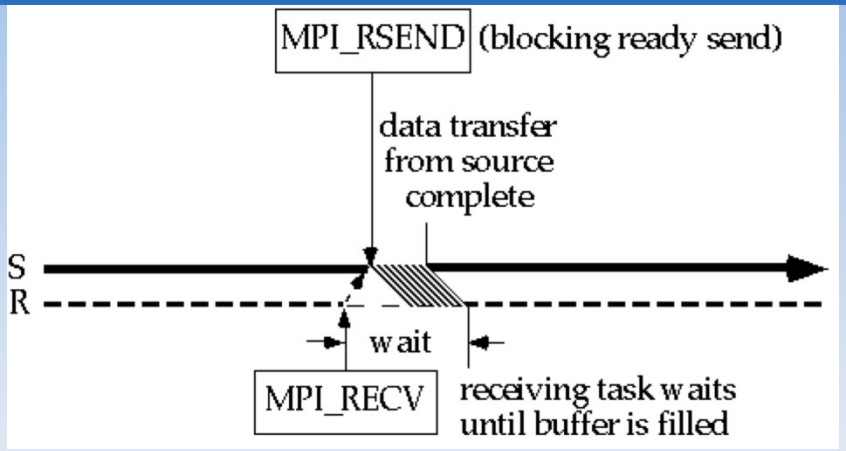
Comunicación punto a punto
Send Bloqueante Síncrono



- El modo de comunicación se selecciona al invocar la rutina **send**.
- Cuando se ejecuta un **send bloqueante síncrono (MPI_Ssend())**, la tarea emisora envía un mensaje "*ready to send*" a la tarea receptora.
- Cuando el receptor ejecuta ***MPI_Recv()***, se envía un mensaje "*ready to receive*", seguido por la transferencia de los datos.
- El emisor debe esperar a que se ejecute el receive y que llegue el handshake antes de que se pueda transferir el mensaje (**Sobrecarga de Sincronización**)
- El receptor debe esperar hasta que se complete el handshake. (**Sobrecarga de Sincronización**)
- También existe la sobrecarga de copia entre los buffers y la red.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

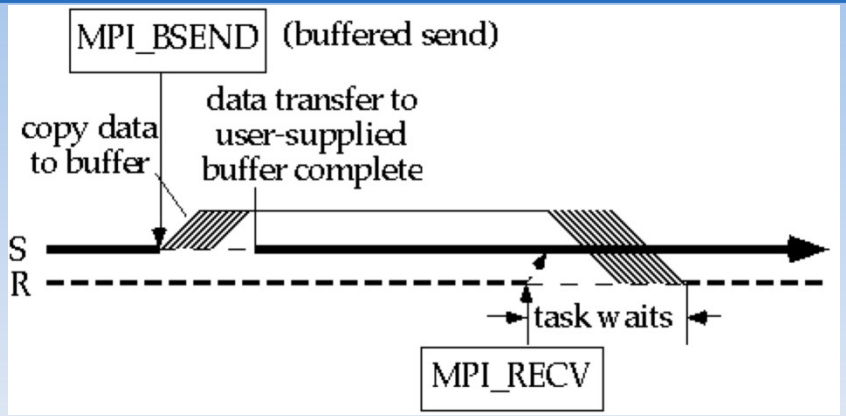
Comunicación punto a punto
Blocking Ready Send



- El modo **ready** (**MPI_Rsend**) envía el mensaje por la red una vez que se ha recibido el mensaje "ready to receive".
- Si el mensaje "ready to receive" no ha llegado, el send producirá un error.
- Este mode minimiza la sobrecarga de sistema y de sincronización en el emisor.
- El receptor puede tener una sobrecarga de sincronización importante, dependiendo de cuanto antes se realizó la llamada.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

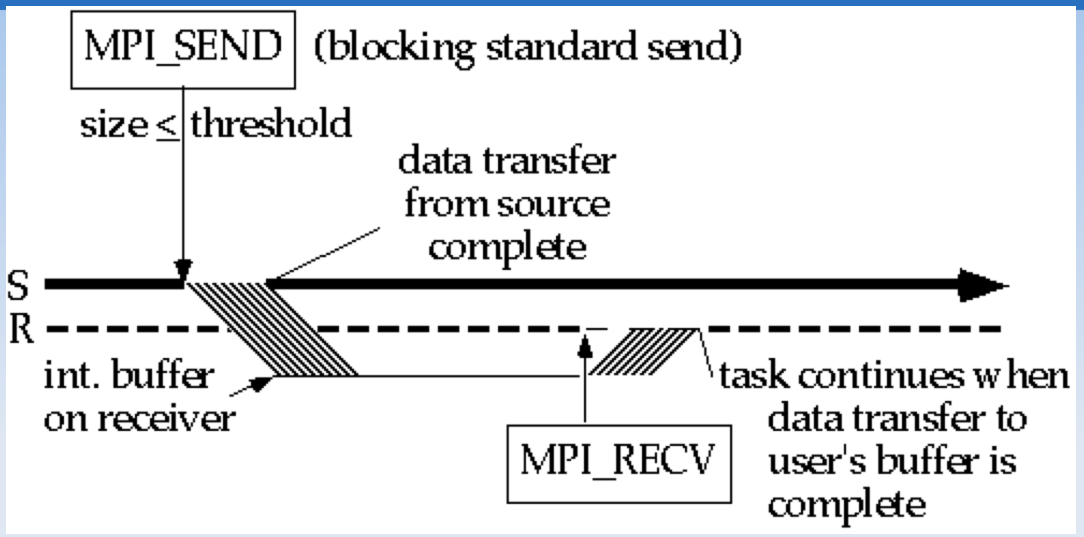
Comunicación Punto a Punto
Blocking Buffered Send



- La llamada MPI_Bsend() copia los datos desde el buffer del mensaje a un buffer provisto por el usuario y retorna.
- El buffer del mensaje puede ser reutilizado por el proceso emisor sin afectar los datos enviados.
- Cuando llega la notificación “ready to receive” se envían los datos desde el buffer provisto por el usuario hacia el receptor.
- Las copias replicadas agregan sobrecarga del sistema.
- La sobrecarga de sincronización en el emisor desaparece, debido a que el proceso emisor puede continuar de inmediato.
- Puede existir sobrecarga de sincronización en el receptor, si el *receive* se ejecuta antes del *send*.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

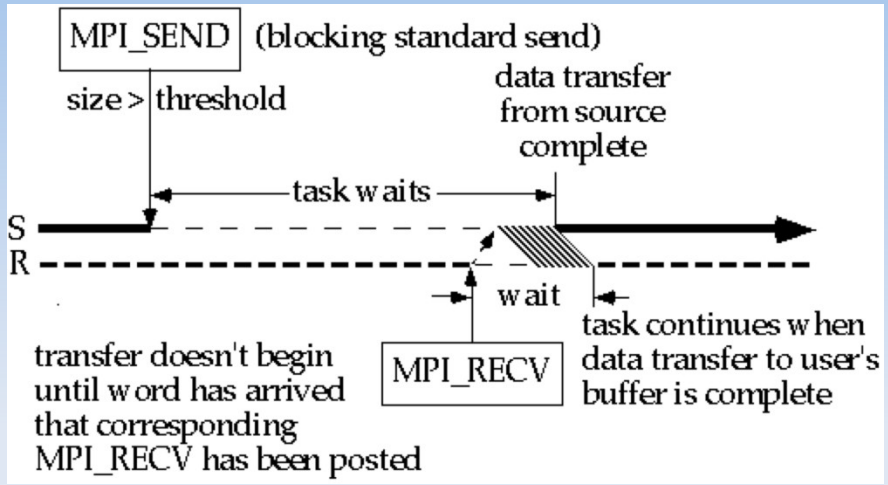
Comunicación Punto a Punto
Blocking Standard Send



- La operación MPI_Send() depende de la implementación.
- Cuando el tamaño de los datos es inferior a cierto límite (dependiente de la implementación):
  - MPI_Send() copia el mensaje a través de la red en el buffer de sistema del nodo receptor, tras lo cual el proceso emisor continúa con la computación.
  - Cuando se ejecuta MPI_Recv() el mensaje se copia desde el buffer de sistema al de la tarea receptora.
  - La disminución de la sobrecarga de sincronización es usualmente al costo del incremento de la sobrecarga de sistema ocasionada por la copia adicional.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

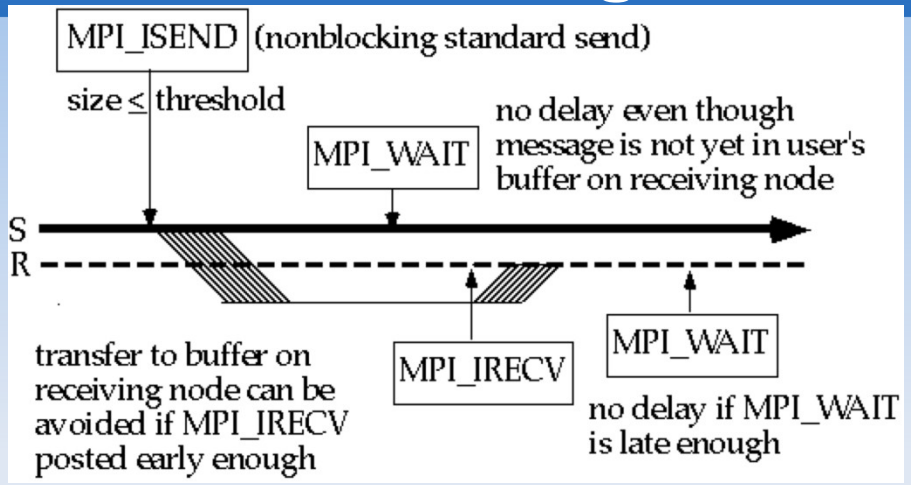
Comunicación Punto a Punto
Buffered Standard Send



- Cuando el tamaño del mensaje es mayor que un determinado límite:
  - El comportamiento es el mismo que en el modo síncrono.
  - Los mensajes pequeños se benefician de una menor sobrecarga de sincronización.
  - Los mensajes largos implican mayores costos de copia al buffer, y mayor sobrecarga de sistema.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

Comunicación Punto a Punto
Non-blocking Calls



- MPI_Isend()) realiza un send estándar no bloqueante cuando los contenidos del buffer del mensaje están listos para ser transmitidos.
- El control retorna inmediatamente sin esperar que se complete la copia al buffer remoto. MPI_Wait se llama antes de que la tarea emisora necesite sobreescribir el buffer de mensaje.
- El programador es responsable de verificar el estado del mensaje para saber cuando se han copiado los datos desde el buffer de envío.
- MPI_Irecv() emite un receive no bloqueante tan pronto el buffer de mensaje está listo para almacenar el mensaje. El receive no bloqueante retorna sin esperar a que el mensaje llegue. La tarea receptora llama a MPI_Wait cuando necesita usar los datos del mensaje entrante.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

Comunicación Punto a Punto
Non-blocking Calls

- Cuando el buffer de sistema está lleno, un send bloqueante tiene que esperar hasta que la tarea receptora extraiga datos del buffer. El uso de llamadas no bloqueantes permiten que se realice computación durante ese intervalo.
- **Las llamadas no bloqueantes permiten evitar situaciones de deadlock.**

Deadlock

- Situación en la que existen dependencias cíclicas entre procesos
  - Un proceso espera un mensaje de otro, pero el segundo está esperando un mensaje del primero, por lo que nada ocurre hasta que el tiempo asignado se cumple y se mata a la tarea. MPI no tiene *timeouts*.

Ejemplo de Deadlock

```
If (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}else {
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
}
```

- Si el tamaño de los mensajes es suficientemente pequeño, puede funcionar debido a los buffers de sistema.
- Si los mensajes son muy grandes, o no se usan buffers de sistema, ambos procesos se colgarán.

Deadlock: Soluciones

```
If (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}else {
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
}
```

o

```
If (rank == 0) {
  err = MPI_Isend(sendbuf, count, datatype, 1, tag, comm, &req);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm);
  err = MPI_Wait(req, &status);
}else {
  err = MPI_Isend(sendbuf, count, datatype, 0, tag, comm, &req);
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm);
  err = MPI_Wait(req, &status);
}
```

Integración numérica usando la regla del trapecio

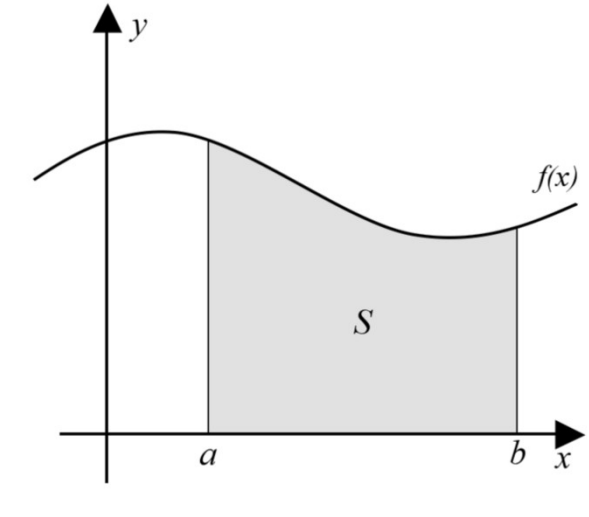
- Resumen: 6 funciones principales:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- Con esas 6 funciones podemos construir distintos tipos de aplicaciones paralelas.
- Veremos como utilizar esas 6 funciones para implementar una versión paralela de la regla del trapecio.

Aproximando Integrales: Integral Definida

- Problema : encontrar el valor aproximado de una integral definida:

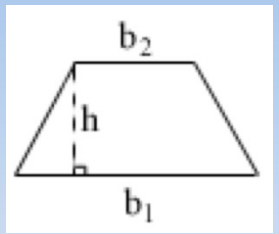
$$\int_a^b f(x)\,dx.$$

- La integral definida entre a y b de una función no negativa f(x) puede pensarse como el area S limitada por el eje X, las líneas verticales x=a y x=b y el gráfico de f(x)

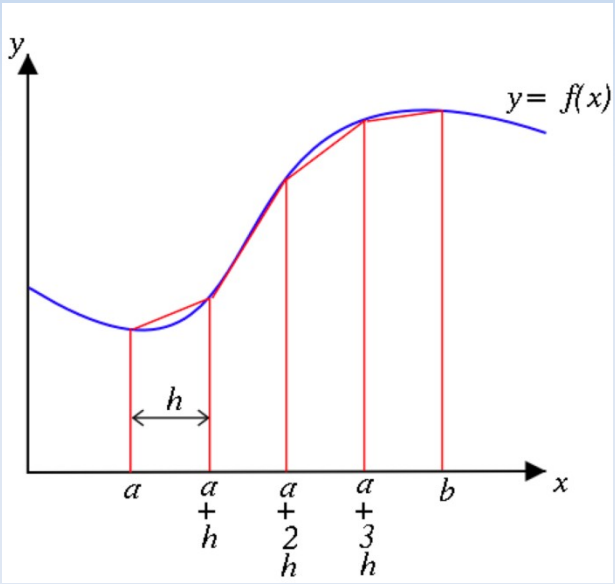


Aproximando Integrales : La Regla del Trapecio

- Se puede aproximar el área bajo la curva dividiendo dicha región en formas geométricas y sumando sus superficies.
- En la Regla del Trapecio la región entre *a* y *b* se divide en *n* trapecios de altura *h = (b-a)/n*
- El área del trapecio puede calcularse como
- En el caso de nuestra función, el área para el primer bloque puede representarse como



$$\frac{h(b_1+b_2)}{2}$$



- El área bajo la curva limitada por a y b puede aproximarse como:

$$\left[\frac{h(f(a)+f(a+h)}{2}\right]+\left[\frac{h(f(a+h)+f(a+2h)}{2}\right]+\left[\frac{h(f(a+2h)+f(a+3h)}{2}\right]+\left[\frac{h(f(a+3h)+f(b)}{2}\right]$$

Aproximando Integrales : La Regla del Trapecio

- Podemos generalizar este concepto de aproximación de integrales como una suma de áreas trapezoidales.

$$\frac{١}{٢}h[f(x_٠)+f(x_١)]+\frac{١}{٢}h[f(x_١)+f(x_٢)]+\ldots+\frac{١}{٢}h[f(x_{n-١})+f(x_n)]$$

$$=\frac{h}{٢}[f(x_٠)+f(x_١)+f(x_١)+f(x_٢)+\ldots+f(x_{n-١})+f(x_n)]$$

$$=\frac{h}{٢}[f(x_٠)+٢f(x_١)+٢f(x_٢)+\ldots+٢f(x_{n-١})+f(x_n)]$$

$$=h\left[\frac{f(x_٠)}{٢}+f(x_١)+f(x_٢)+\ldots+f(x_{n-١})+\frac{f(x_n)}{٢}\right]$$

€

Aproximando Integrales : La Regla del Trapecio

- Podemos generalizar este concepto de aproximación de integrales como una suma de áreas trapezoidales.

$$\frac{1}{2}h[f(x_0)+f(x_1)]+\frac{1}{2}h[f(x_1)+f(x_2)]+\ldots+\frac{1}{2}h[f(x_{n-1})+f(x_n)]$$

$$=\frac{h}{2}[f(x_0)+f(x_1)+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+f(x_n)]$$

$$=\frac{h}{2}[f(x_0)+2f(x_1)+2f(x_2)+\ldots+2f(x_{n-1})+f(x_n)]$$

$$=h\left[\frac{f(x_0)}{2}+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+\frac{f(x_n)}{2}\right]$$

€

Regla del Trapecio: Programa secuencial en C

```
/* serial.c -- serial trapezoidal rule
 *
 * Calculate definite integral using trapezoidal
rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *    using n trapezoids.
 *
 * See Chapter 4, pp. 53 & ff. in PPMPI.
 */

#include <stdio.h>

main() {
   float  integral;   /* Store result in integral   */
   float  a, b;       /* Left and right endpoints   */
   int    n;          /* Number of trapezoids       */
   float  h;          /* Trapezoid base width       */
   float  x;
   int    i;
   float f(float x);  /* Function we're integrating
*/

   printf("Enter a, b, and n\n");
   scanf("%f %f %d", &a, &b, &n);

   h = (b-a)/n;
   integral = (f(a) + f(b))/2.0;
   x = a;
   for (i = 1; i <= n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;

   printf("With n = %d trapezoids, our
estimate\n",
      n);
   printf("of the integral from %f to %f = %f\n",
      a, b, integral);
} /* main */


float f(float x) {
   float return_val;
   /* Calculate f(x).  Store calculation in
return_val. */

   return_val = x*x;
   return return_val;
} /* f */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

Resultados del programa secuencial

a	b	n	f(x)
2	25	1	7233.500000
2	25	2	5712.625000
2	25	10	5225.945312
2	25	30	5207.916992
2	25	40	5206.934082
2	25	50	5206.475098
2	25	1000	5205.664551

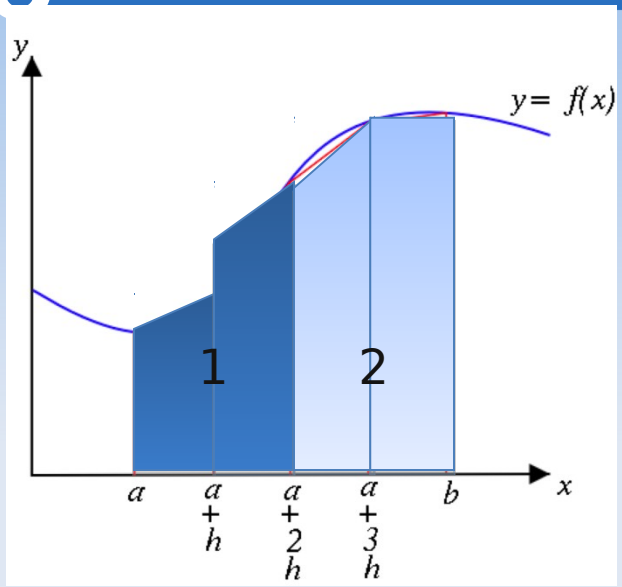
Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

Paralelización de la Regla del Trapecio

- Una posible forma:
  - Distribuimos la carga de trabajo en segmentos. A cada proceso le corresponde un subintervalo de ***[a,b].***
  - Calculamos ***f*** para cada subintervalo
  - Sumamos los ***f*** calculados en todos los subintervalos para producir la solución del problema completo.
- Temas a considerar
  - El número de trapecios (*n*) puede dividirse equitativamente entre los (*p*) procesos (Balance de carga).
  - El primer proceso calcula el área de los primeros

    n/p trapecios, el segundo de los siguientes *n/p* trapecios y así sucesivamente.
- Cada proceso necesita:
  - Conocer su rango
  - Ser capaz de calcular el subintervalo en función de su rango.

  Premisa : El proceso 0 hace la suma



Paralelización de la Regla del Trapecio

- Algoritmo
  Premisa: El número de trapecios *n* es divisible por el número de procesos *p*.
  - Calcular:

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

  - Cada proceso calcula el intervalo a integrar
    - Número local de trapecios (local_n) = ***n/p***
    - Punto de inicio local (local_a) = a+(process_rank *local_n* h)
    - Punto de finalización local (local_b) = (local_a + local_n * h)
  - Cada proceso calcula su propia integral para los intervalos locales
    - Calcular el área de cada uno de los local_n trapecios
    - Sumar el área de los local_n trapecios
  - If PROCESS_RANK == 0
    - Recibir los mensajes que contienen los resultados de cada subintervalo
    - Sumar las áreas.
  - If PROCESS_RANK > 0
    - Enviar el área del subinterval a PROCESS_RANK(0)

SPMD clásico: todos los procesos ejecutan el mismo programa sobre distintos datos.