

Introducción

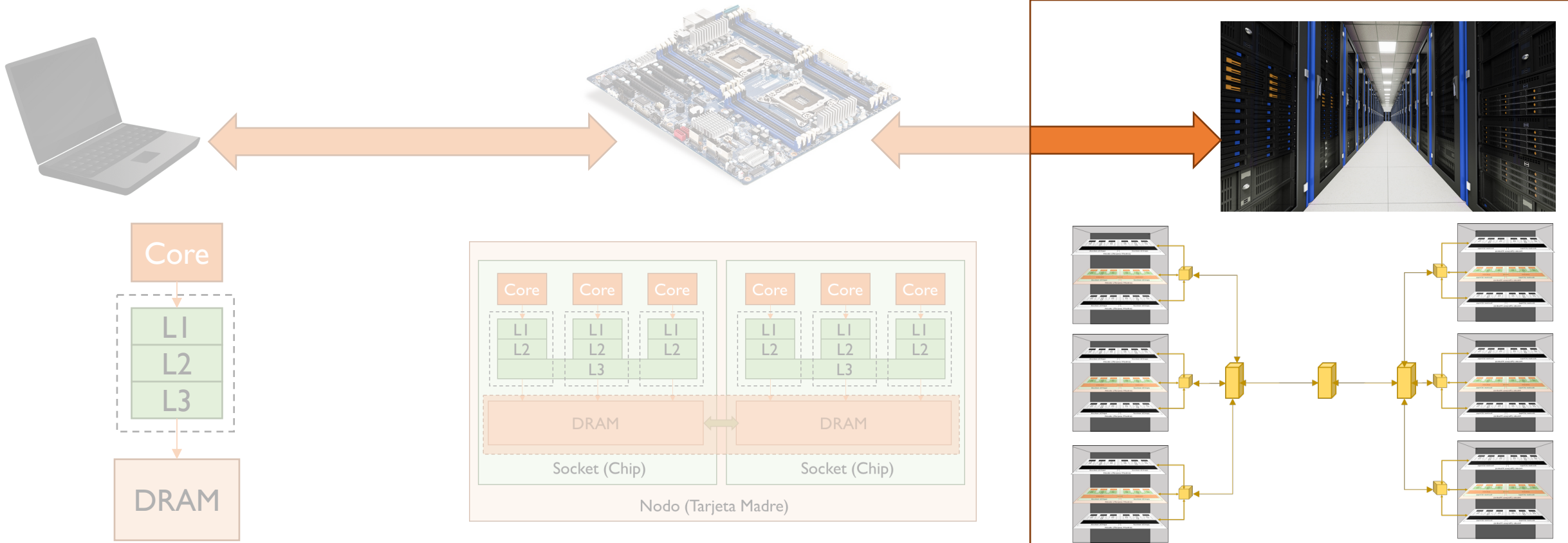
Memoria Distribuida

Programación en Paralelo

ICME Summer Workshop @ Santiago: Fundamentals of Data Science

Cindy Orozco

Hardware



0) Procesamiento Secuencial

1) Memoria Compartida

- ✓ Evitar conflictos memoria
- ✓ Distribuir Tareas

OpenMP

2) Memoria Distribuida

- ✓ Comunicación entre nodos
- ✓ Asignar Tareas

MPI

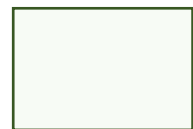
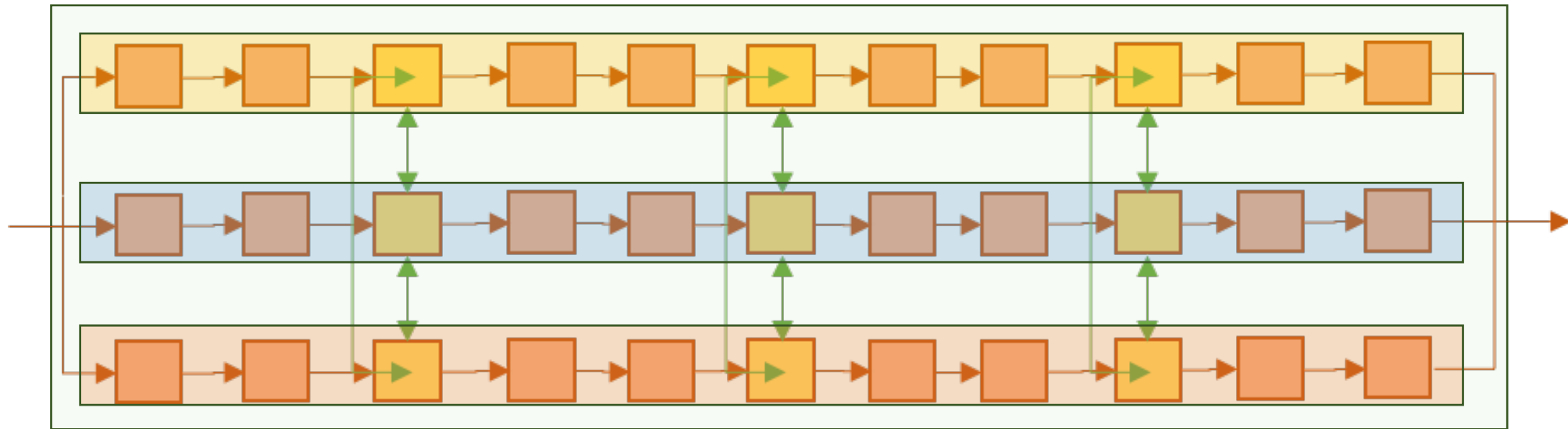
Planificador de Tareas

MPI : Interfaz de Paso de Mensajes

- ✓ **IEEE estándar:** No es un producto o una especificación de compiladores
- ✓ Diferentes implementaciones optimizadas para cada caso:
 - ✓ **Open MPI** <https://www.open-mpi.org/>
 - ✓ MPICH
 - ✓ IBM MPI
 - ✓ Intel MPI
 - ✓ MVAPICH
- ✓ Implementaciones de fabricantes aprovechan las especificaciones propias de hardware para mejorar performance
- ✓ Tiene más de 200 instrucciones, pero en un código estándar se usan por máximo 20

Granularidad Gruesa

La mayoría de cálculos pueden ser ejecutados en paralelo, con pequeños intercambios de comunicación.



Región
Paralela



Rank 0



Rank 1



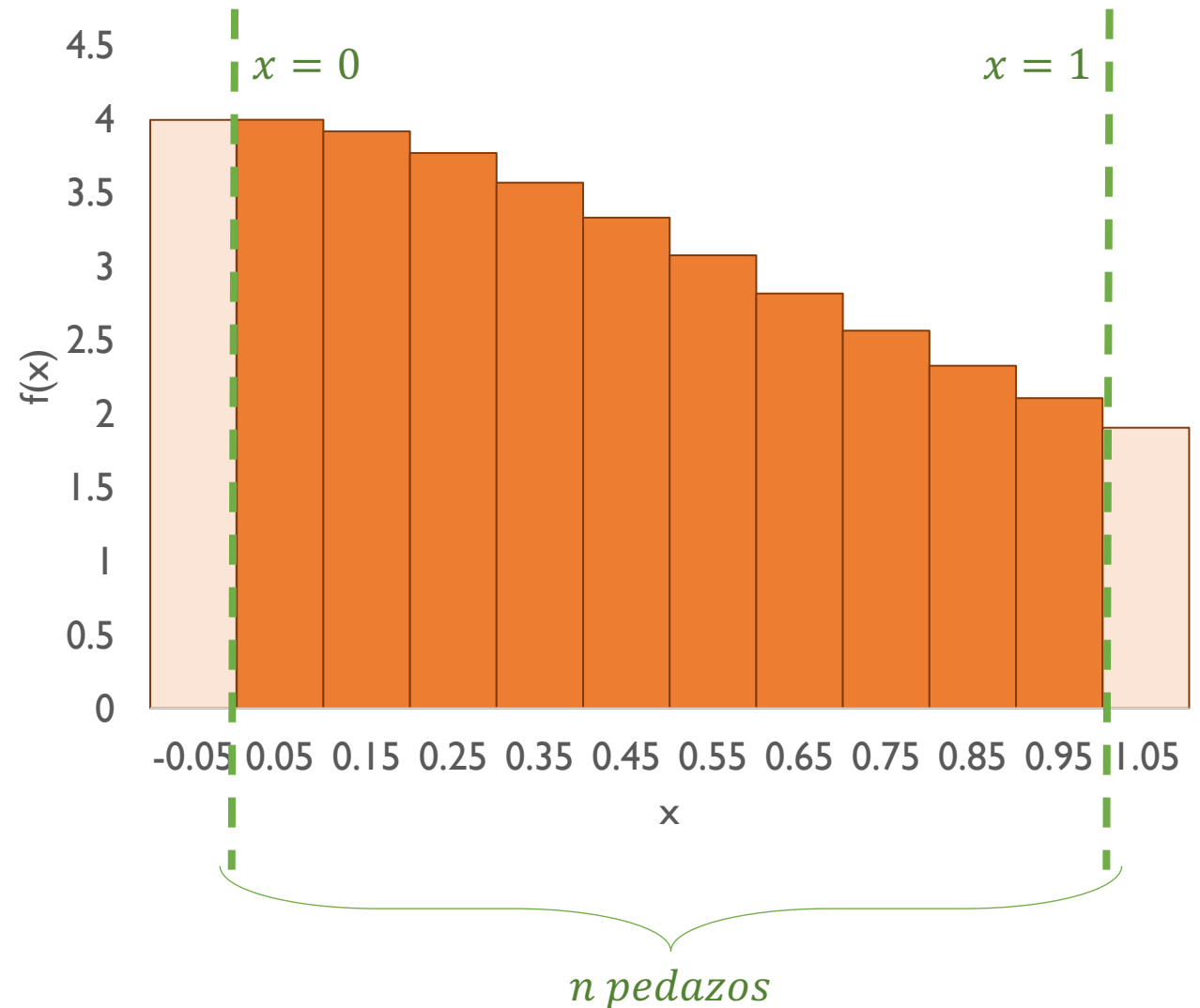
Rank 2

Problema: Calcular π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \sum_{i=0}^{n-1} f(x_i) * \frac{1}{n}$$

donde

$$x_i = \frac{i + 0.5}{n}$$



Problema: Calcular π / Código Serial


```
#include <stdio.h>
int main(){
    double sum = 0;
    int n = 10;
    double step = 1. / ((double) n);
    double x;
    for (int i = 0; i < n ; ++i){
        x = (i + 0.5) * step;
        sum += 4. / (1. + x * x);
    }
    sum = sum * step;
    printf("El valor de pi es %f",sum);
}
```


$$\sum_{i=0}^{n-1} f\left(\frac{i + 0.5}{n}\right) * \frac{1}{n}$$

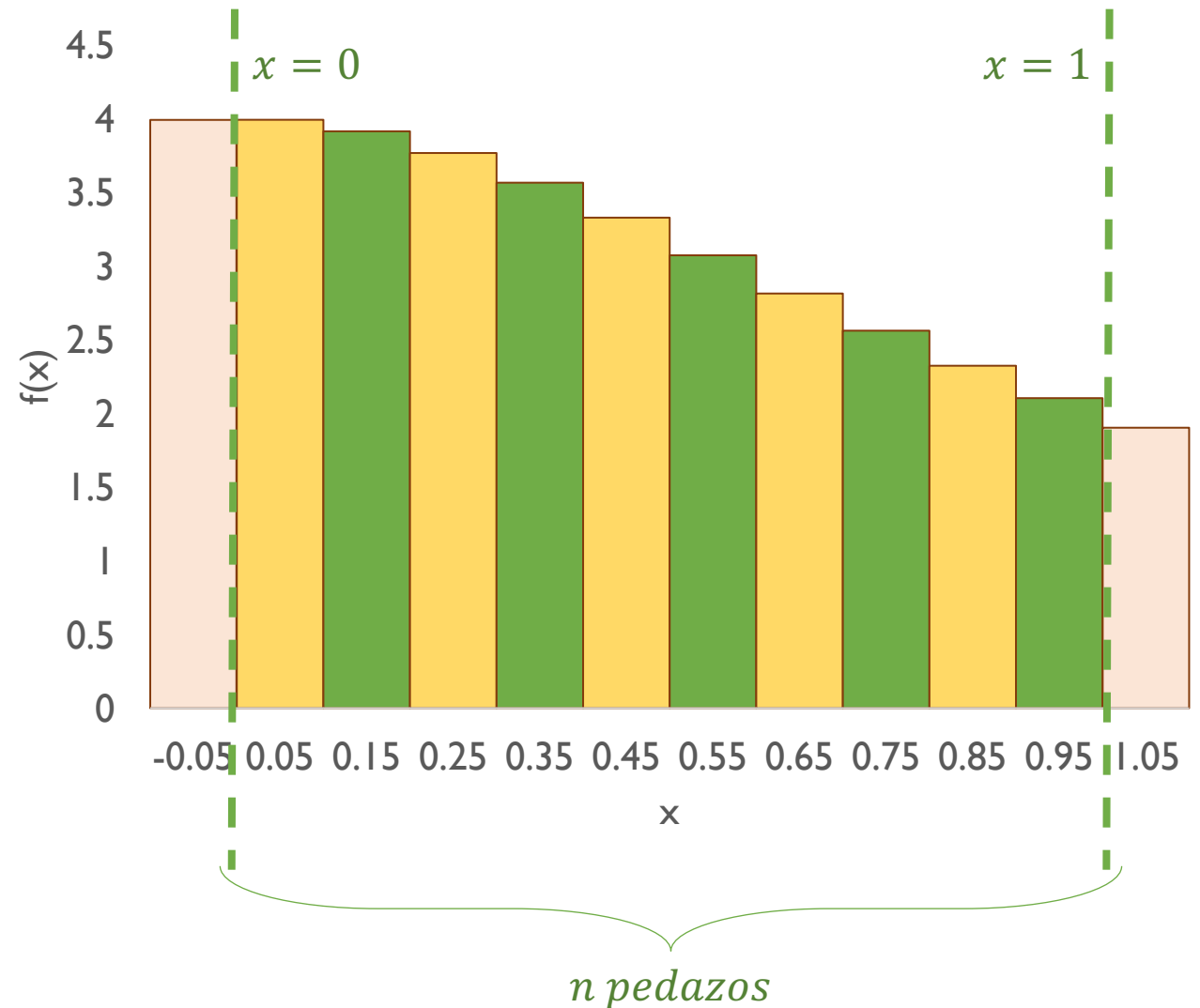
Cada iteración es independiente
→ Paralelizar

Problema: Calcular π

Si tenemos $p = 2$ procesadores

 Procesador 0 = $\sum_{i=0,2,\dots}^{n-1} f(x_i) * \frac{1}{n}$

 Procesador 1 = $\sum_{i=1,3,\dots}^{n-1} f(x_i) * \frac{1}{n}$



Problema: Calcular π /Solución con OpenMP sin atajos

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

```
#include <omp.h>
```

```
#define NUM_THREADS 2
```

```
int main(){
```

```
    double sum[NUM_THREADS] = {0};
```

```
    int n = 10;
```

```
    double step = 1. / ((double) n);
```

```
    omp_set_num_threads(NUM_THREADS);
```

```
    #pragma omp parallel
```

```
{
```

```
    double x ,mysum;
```

```
    int tid = omp_get_thread_num();
```

```
    int nthreads = omp_get_num_threads();
```

```
    for (int i = tid; i < n; i+= nthreads {
```

```
        x = (i + 0.5) * step;
```

```
        mysum += 4. / (1. + x * x);
```

```
    }
```

```
    sum[tid] = mysum;
```

```
}
```

```
for (int j = 1; j < NUM_THREADS; ++j){
```

```
    sum[0] += sum[j];
```

```
sum[0] = sum[0] * step;
```

```
printf("El valor de pi es %f", sum[0] );
```

```
}
```

→ Librería OpenMP

→ Fijar número de hilos

→ Iniciar región paralela

→ Cada hilo identifica ¿quién es?
y ¿Cuántos hay en total?

→ Subconjunto de iteraciones

→ Agrupar resultado de los
threads

Problema: Calcular π /Solución con MPI

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

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

```
int main(){
```

```
    MPI_Init(&argc, &argv)
```

```
    int n = 10;
```

```
    double step = 1. / ((double) n);
```

```
    double sum = 0;
```

```
    double x;
```

```
    int rank, size;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    for (int i = rank; i < n; i += size){
```

```
        x = (i + 0.5) * step;
```

```
        sum += 4. / (1. + x * x);
```

```
    }
```

```
    if (rank == 0){
```

```
        total = sum;
```

```
        for (int j = 1; j < size; ++j){
```

```
            MPI_Recv(&sum, 1, MPI_DOUBLE, j, tag, MPI_COMM_WORLD);
```

```
            total += sum;
```

```
        total = total * step;
```

```
        printf("El valor de pi es %f", total);
```

```
    }
```

```
    else{
```

```
        MPI_Send(&sum, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
```

```
    MPI_Finalize();
```

```
}
```

→ Librería MPI

→ Iniciar región paralela

→ Cada rank identifica ¿quién es?
y ¿Cuántos hay en total?

→ Subconjunto de iteraciones

→ Agrupar resultado de los ranks

→ Terminar región paralela

Características de MPI

- Tenemos múltiples **ranks**, cada uno con su propia memoria
- Compartir memoria se realiza explícitamente

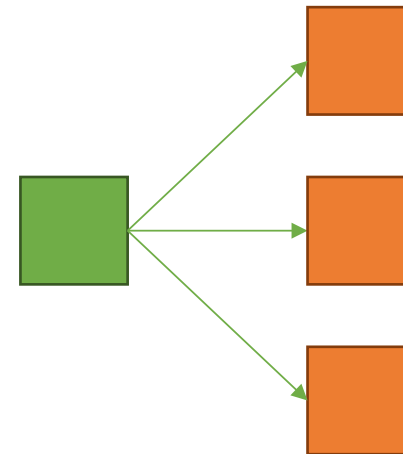
Inicialización

*¿Quién soy?
¿Cuántos hay
en total?*

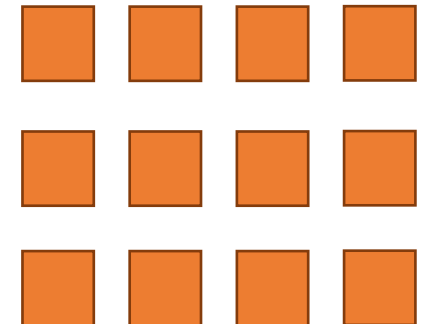
Comunicación I a I



Comunicación colectiva



Herramientas adicionales



Inicialización

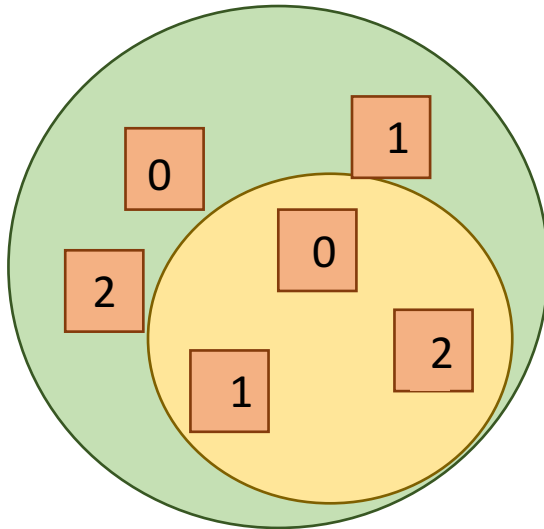
- Iniciar y terminar el código en paralelo

```
MPI_Init(&argc, &argv)
```

```
MPI_Finalize()
```

Procesa los parámetros => secuencial

- Manejar el comunicador principal MPI_COMM_WORLD



Comunicador: Conjunto de nodos

Cada nodo dentro del comunicador recibe un identificador **rank**

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

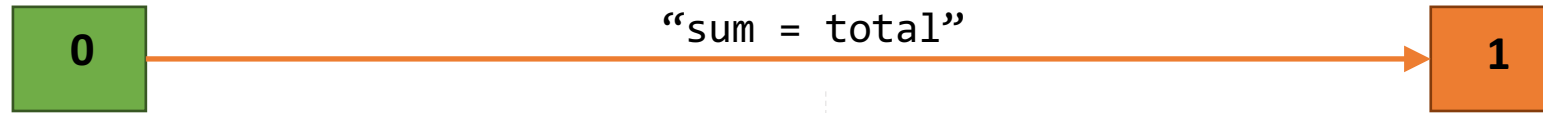
El número total de nodos en el comunicador es el tamaño **size**

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Toda comunicación requiere de un comunicador y de un rank

Podemos crear nuestros propios comunicadores, con sus propios rank

Comunicación 1 a 1



```
double total= 10;
```

```
MPI_Send(&total,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
```



```
double sum;
```

```
MPI_Recv(&sum,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,status);
```



¿Cómo se realiza la comunicación?

¿Cómo se sincronizan los ranks?

¿Cómo se utilizan buffers?

Comunicación 1 a 1: Blocking



Por defecto, la comunicación es **blocking**: Sólo se termina hasta que el espacio de memoria se puede reutilizar

Recibir Termina cuando toda la información esperada está en
Standard(MPI_Recv)

Espacio de memoria

Enviar Termina cuando toda la información se copia fuera del

Espacio de memoria

Comunicación I a I: Blocking



Enviar Termina cuando toda la información se copia fuera del Espacio de memoria

Buffered(MPI_Bsend)

Buffer

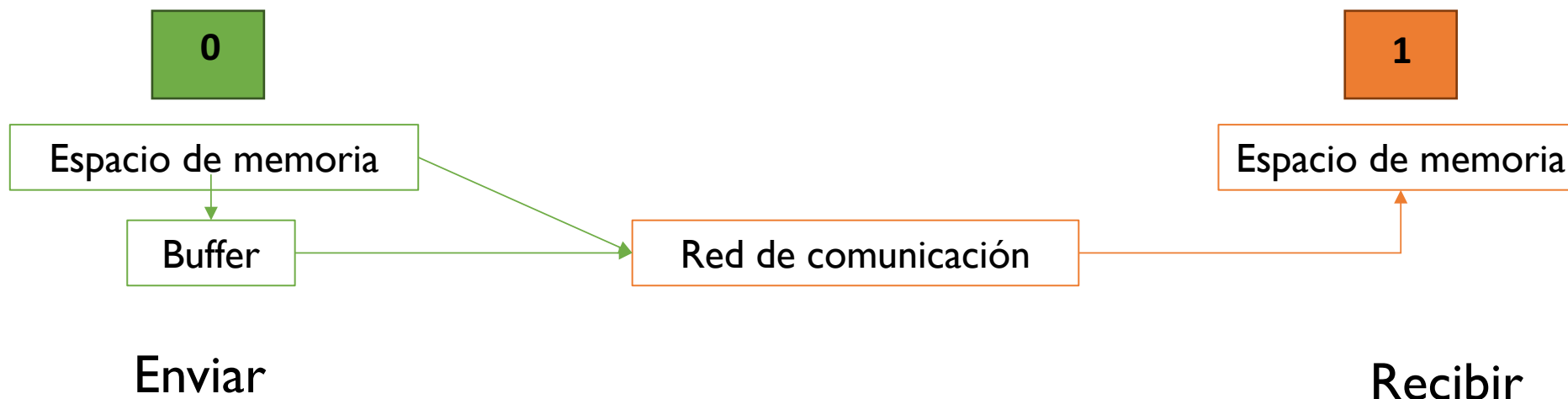
Sincrónico (MPI_Ssend)

Espacio de memoria

Espera hasta que rank 1 recibe

Estándar (MPI_Send) Decide buffer o directo dependiendo del tamaño del mensaje

Comunicación I a I: Blocking



¿Cuál Usar?

MPI_Send

MPI_Bsend
(buffered)

MPI_Ssend
(Sincrónico)

Requiere más
memoria disponible

Se espera más tiempo
hasta confirmación

MPI_Recv

Comunicación I a I: Blocking

0

```
double total= 10;  
double sum;  
  
MPI_Send(&total,1,MPI_DOUBLE,1,...);  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);
```

1

```
double total = 11;  
double sum;  
  
MPI_Send(&total,1,MPI_DOUBLE,0,...);  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);
```

Si no se utiliza buffer, ranks no podrán terminar de enviar (Send) porque no existe un Recv donde puedan copiar la memoria

Deadlock o Bloqueo Mutuo

Comunicación 1 a 1: Blocking

0

```
double total= 10;  
double sum;  
  
MPI_Send(&total,1,MPI_DOUBLE,1,...);  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);
```

1

```
double total = 11;  
double sum;  
  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);  
MPI_Send(&total,1,MPI_DOUBLE,0,...);
```

Si no se utiliza buffer, ranks no podrán terminar de enviar (Send) porque no existe un Recv donde puedan copiar la memoria

Deadlock o Bloqueo Mutuo

El orden de las llamadas es importante

Comunicación 1 a 1: Blocking

0

```
double total= 10;  
MPI_Send(&total,1,MPI_DOUBLE,1,...);
```

Código que no depende de total

Código con total

1

```
double sum;  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);
```

Código que no depende de sum

Código con sum

Tiempo de ejecución

T blocking

Rank 0

Comunicación

Código sin total

Código con total

Rank 1

Comunicación

Código sin sum

Código con sum

Comunicación I a I: Blocking

0

```
double total= 10;  
MPI_Send(&total,1,MPI_DOUBLE,1,...);
```

Código que no depende de total

Código con total

1

```
double sum;  
MPI_Recv(&sum,1,MPI_DOUBLE,0,...);
```

Código que no depende de sum

Código con sum

Tiempo de ejecución

Rank 0

Código sin total

Código con total

Comunicación

Rank 1

Código sin sum

Código con sum

Comunicación

T No
blocking

T blocking

*Para tener un mejor
rendimiento, se deben
esconder las
comunicaciones*

Comunicación I a I: Non Blocking

Se separan las instrucciones:

- ✓ Que inician la comunicación
Retornan inmediatamente

`MPI_Irecv`

`MPI_Isend`

`MPI_Ibsend`
(buffered)

`MPI_issend`
(Sincrónico)

- ✓ Que verifican que el espacio de memoria se puede utilizar

`MPI_Test(request, flag, ...)`

Retorna V/F si la comunicación ha sido terminada

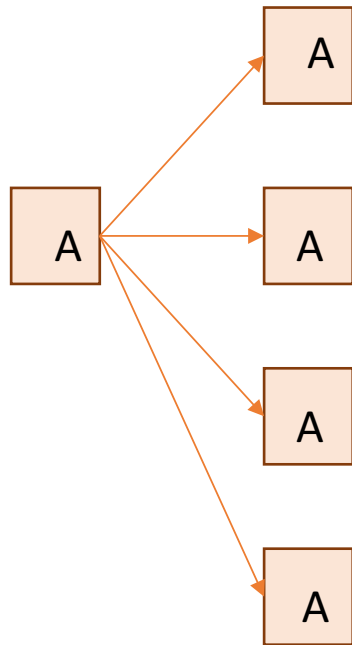
`MPI_Wait(request, ...)`

Espera hasta que la comunicación ha sido terminada

Comunicación colectiva

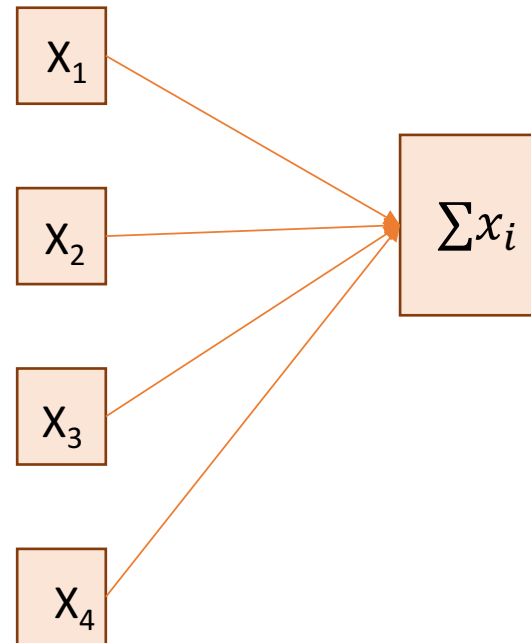
MPI_Bcast

Broadcast



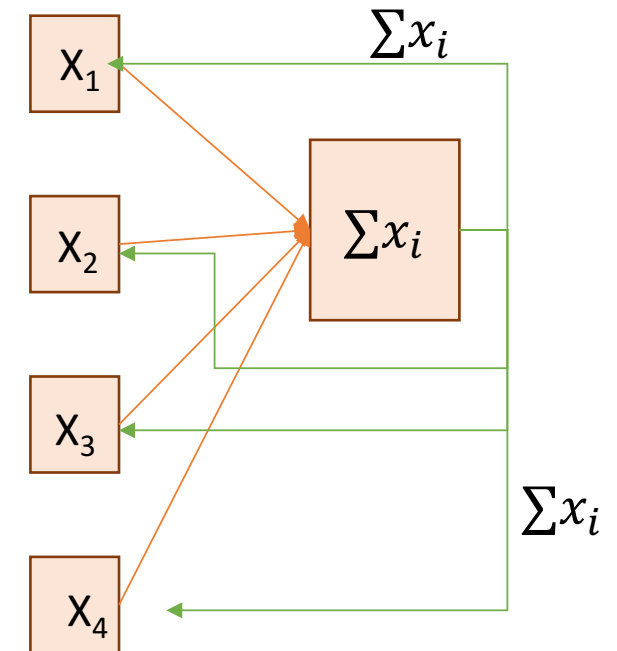
MPI_Reduce

Reducción



MPI_Allreduce

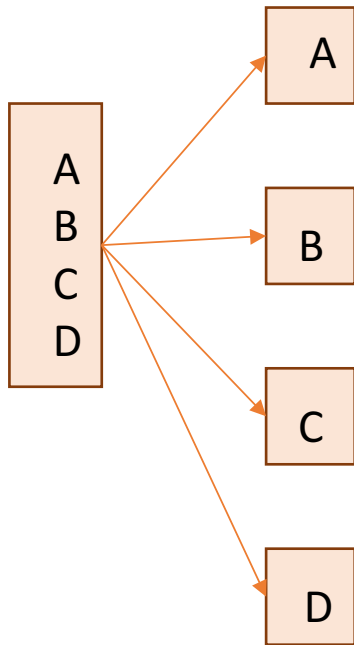
Reducción a todos



Comunicación colectiva

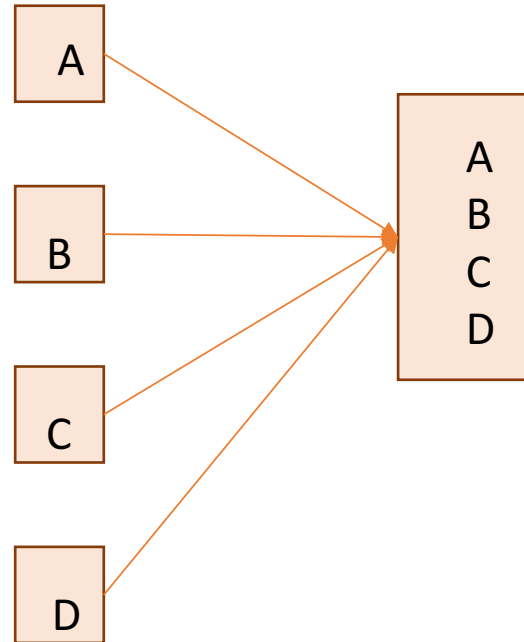
MPI_Scatter

Dispersar



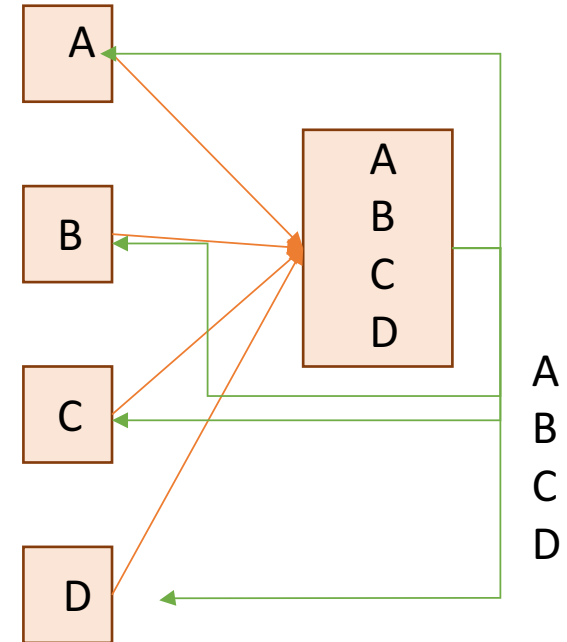
MPI_Gather

Reunir



MPI_Allgather

Reunir en todos



Problema: Calcular π /Solución con MPI + reducción

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

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

→ Librería MPI

```
int main(int argc, char **argv){
```

```
    MPI_Init(&argc, &argv);
```

→ Iniciar región paralela

```
    int n = 10;
```

```
    double step = 1. / ((double) n);
```

```
    double sum = 0;
```

```
    double x;
```

```
    int rank, size;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Cada rank identifica ¿quién es?
y ¿Cuántos hay en total?

```
    for (int i = rank; i < n; i += size){
```

```
        x = (i + 0.5) * step;
```

```
        sum += 4. / (1. + x * x);
```

→ Subconjunto de iteraciones

```
    }
```

```
    double total;
```

```
    MPI_Reduce(&sum,&total,1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)
```

→ Agrupar resultado de los ranks

```
    if (rank == 0){
```

```
        total = total * step;
```

```
        printf("El valor de pi es %f",total);
```

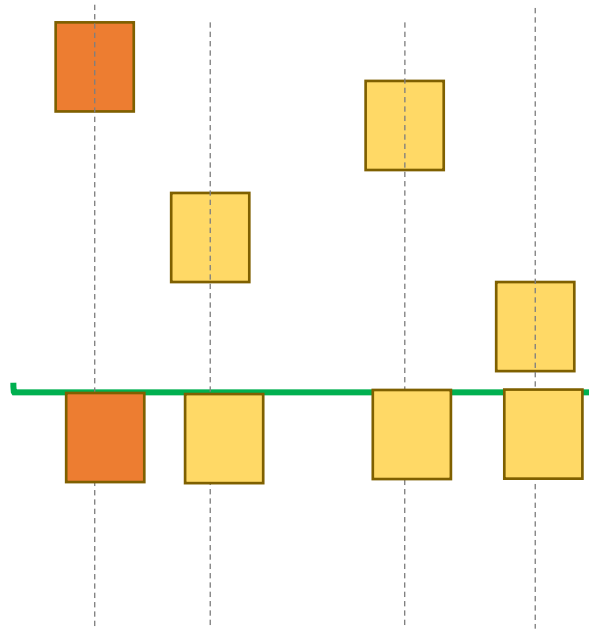
```
    }
```

```
    MPI_Finalize();
```

→ Terminar región paralela

Herramientas adicionales: MPI_Barrier()

- Pone una barrera a todos los ranks para asegurar que todos alcancen el mismo punto en el código antes de continuar



Herramientas adicionales: MPI_Wtime()

- Medir el tiempo real de ejecución (Wall time), que es diferente al tiempo en CPU que solo mide cuando el procesador estaba trabajando activamente en una tarea

```
double start, finish, time;  
MPI_Barrier(MPI_COMM_WORLD);  
start = MPI_Wtime();  
  
...  
MPI_Barrier(MPI_COMM_WORLD);  
finish = MPI_Wtime();  
time = finish - start;
```

Herramientas Adicionales: Manejo de datos

- Al comunicar debemos especificar tipo y cantidad de información

MPI datatype

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_LONG_LONG_INT
MPI_LONG_LONG (as a synonym)
MPI_SIGNED_CHAR
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_LONG_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_WCHAR
MPI_BYTE
MPI_PACKED

C datatype

signed char
signed short int
signed int
signed long int
signed long long int
signed long long int
signed char
unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int
float
double
long double
wchar_t

Herramientas Adicionales: Manejo de datos

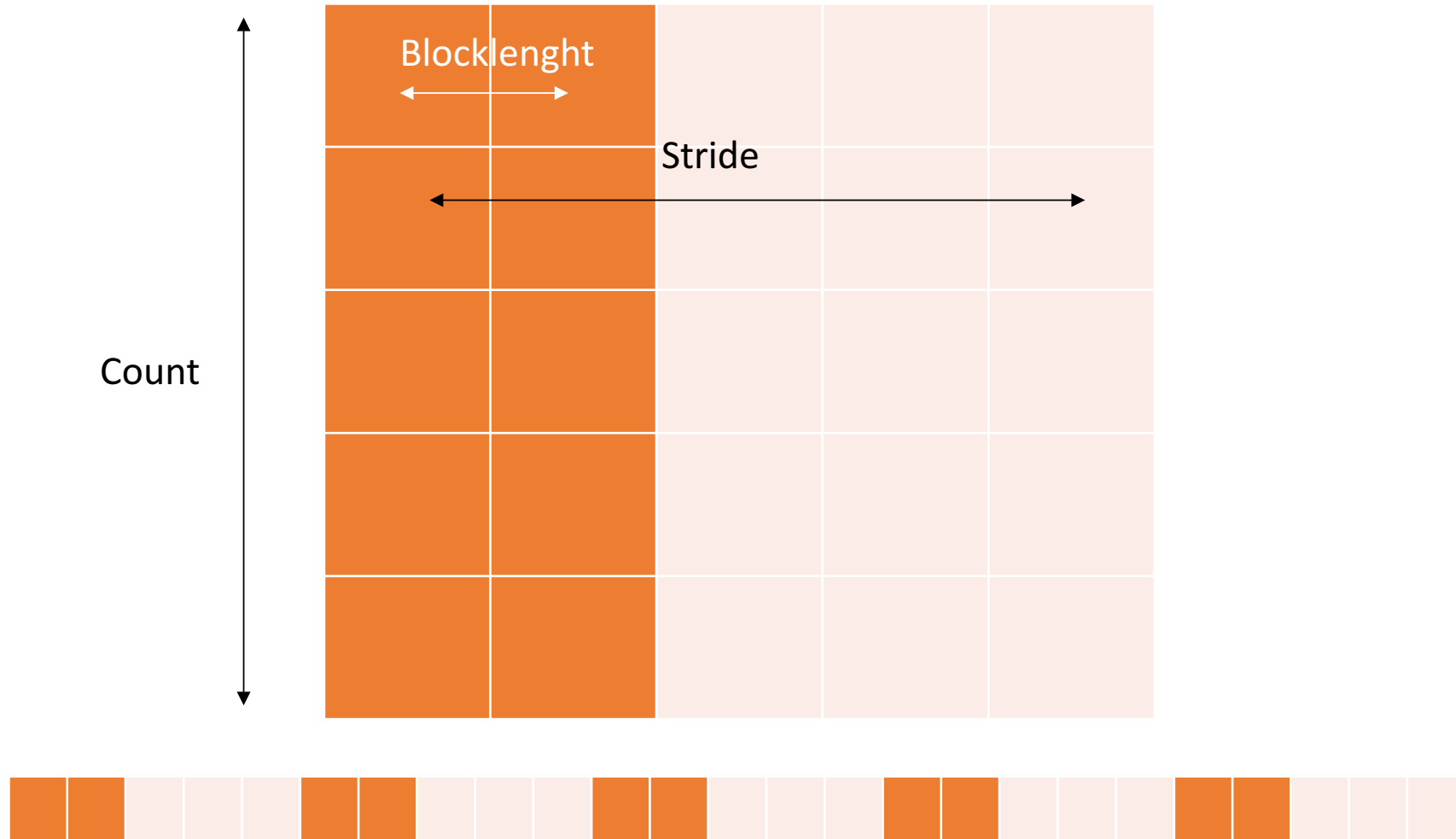
- ✓ ¿Qué hacer cuando los datos no están consecutivos en memoria?
- ✓ ¿Qué hacer cuando queremos enviar datos de múltiples tipos?

Opción 1: Establecer múltiples comunicaciones, una por cada tipo de dato. Sin embargo establecer cada comunicación es costoso (latencia).

Opción 2: Crear nuestro propio tipo de dato para enviar.

Herramientas Adicionales: Manejo de datos

`MPI_Type_vector(count, blocklength, stride, oldtype, *newtype)`



Herramientas Adicionales: Manejo de datos

```
MPI_Datatype mi_vector;
```

```
int bloque = 2;
```

```
int cols = 5;
```

```
int rows = 5;
```

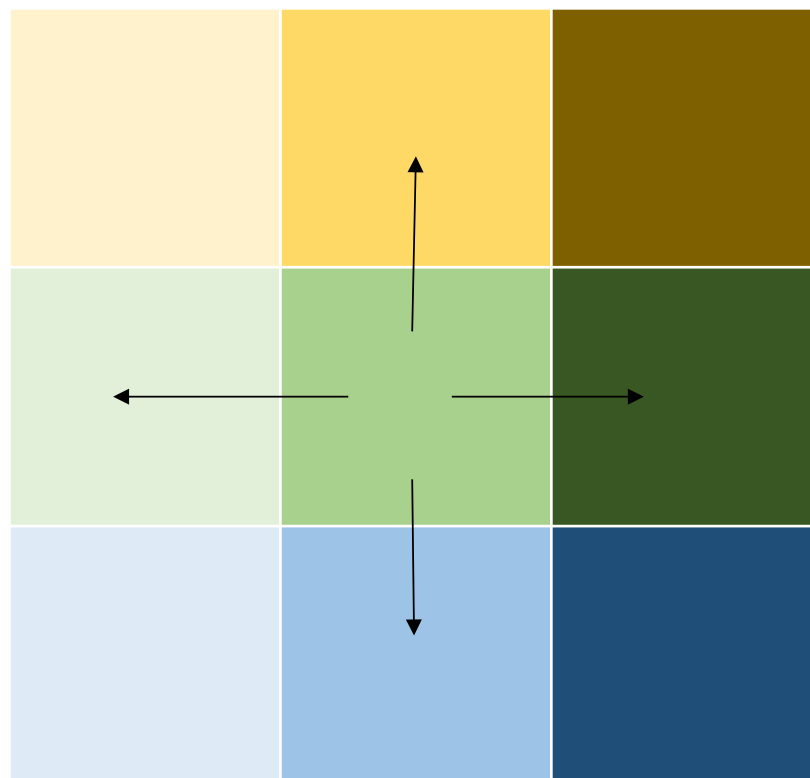
```
MPI_Type_vector(rows,bloque,cols,MPI_DOUBLE,&mi_vector);
```

```
MPI_Type_commit(&mi_vector); //Para empezarlo a usar
```

```
MPI_Type_free (&mi_vector); //Para liberarlo
```

Herramientas Adicionales: Topologías

- La forma de identificar cada rank es con un número de $0, \dots, \text{nodos} - 1$
- Las topologías son una identificación adicional de acuerdo a cierta geometría



¿Cómo
identificar a los
vecinos?
Creamos un
nuevo
comunicador

Herramientas Adicionales: Topología Cartesiana

```
MPI_Cart_create(old_comm, ndims, dims, periods, reorder, &new_comm)
```

old_comm: Comunicador anterior: MPI_COMM_WORLD

ndims: # dimensiones: 1,2,3,...

dims: array con el tamaño de cada dimensión

periods: 0/1 array para definir si la dimensión es periódica o no

reorder: si se reorganizan los ranks

new_comm: nombre de la nueva topología

Resumen de instrucciones

- Inicialización
 - `MPI_Init, MPI_Finalize`
- Comunicación Blocking
 - Se espera hasta la memoria se encuentra disponible*
 - `MPI_Recv`
 - `MPI_Send/MPI_Bsend,/MPI_Ssend`
 - `MPI_Irecv`
- Comunicación Non Blocking
 - Se espera separa el inicio y la verificación de memoria*
 - `MPI_Isend/MPI_Ibsend,/MPI_Issend`
- Comunicación colectiva
 - `MPI_Bcast/MPI_Reduce,/MPI_Allreduce`
 - `MPI_Scatter/MPI_Gather/MPI_Allgather`
- Herramientas adicionales
 - `MPI_Barrier`
 - `MPI_Wtime`
 - `MPI_Type_vector`
 - `MPI_Cart_create`

¿Cómo se usa MPI?

- Se debe incluir la librería

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

- Se debe compilar dependiendo de la distribución de MPI

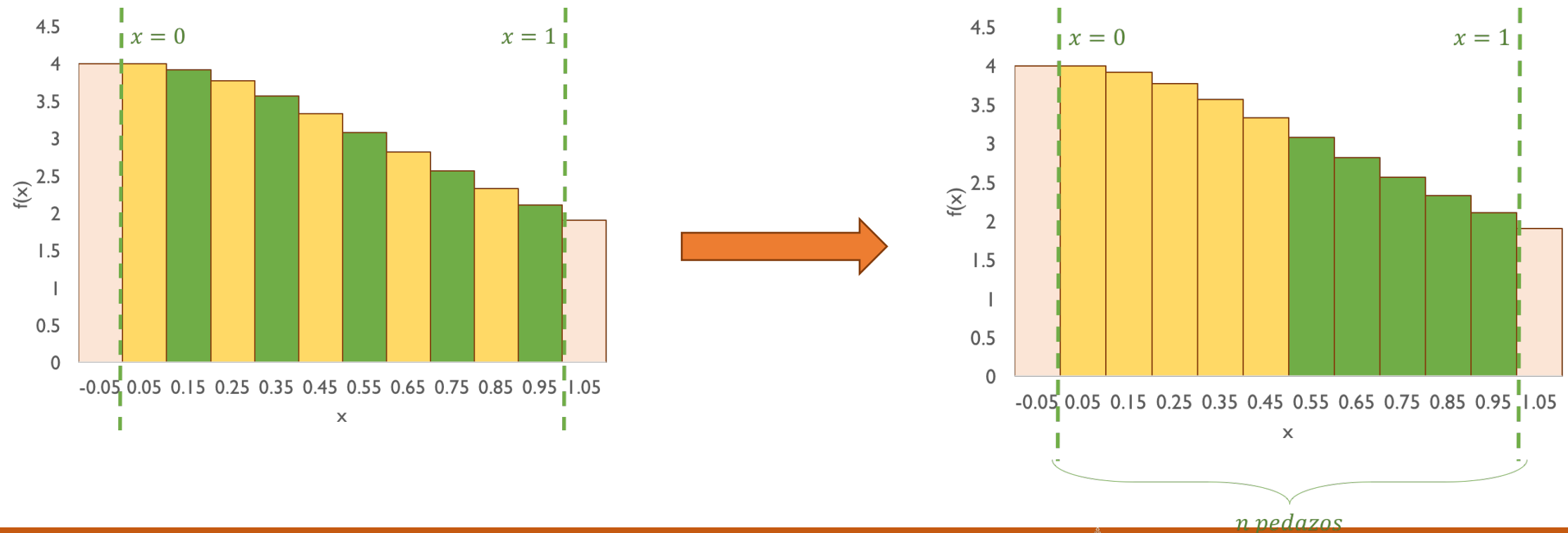
```
mpicc hello_world_mpi.c -o hello
```

- Se corre el programa especificando el numero de ranks requeridos

```
mpirun -n 2 hello
```

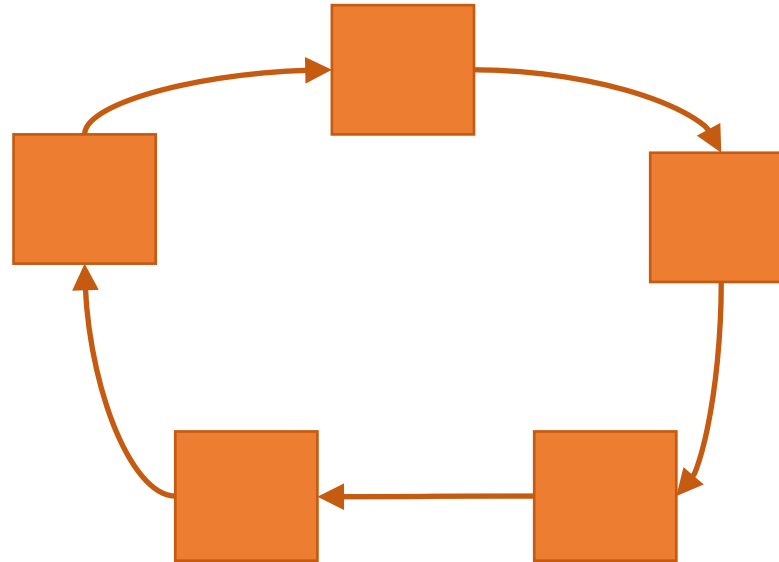
Prueba las diferentes soluciones de Pi

- ¿Cómo afecta la cantidad de ranks asignados?
- ¿Cómo la cambiarías para cambiar la forma en que se recorren los datos?

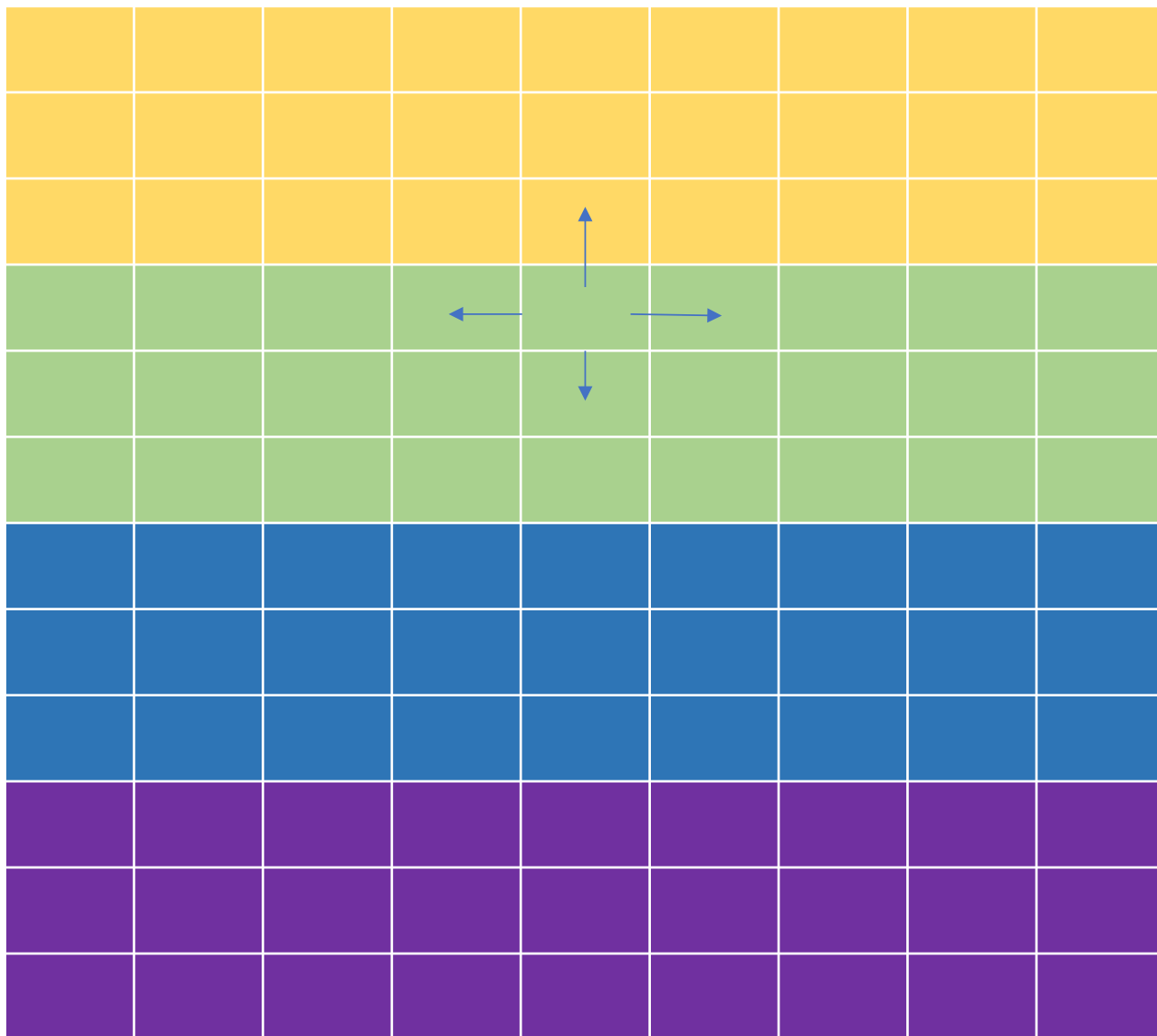


Otros ejercicios

- ✓ ¿Cómo establecería la comunicación en un anillo? Implemente una función usando MPI en donde para cualquier numero de procesadores, cada uno envía a su vecino un número entero.



Otros ejercicios



- ✓ Tenemos una función en 2D y queremos aproximar el Laplaciano en cada punto

$$\Delta[i, j] \approx \frac{x[i-1, j] + x[i+1, j] + x[i, j+1] + x[i, j-1] - 4 * x[i, j]}{n^2}$$

- ✓ Cada banda de color pertenece a un nodo diferente, entonces ¿cómo sería la comunicación?
- ✓ Realice una implementación donde $x[i, j] = i * i + j * j$