Procesamiento paralelo

Universidad Tecnológica Nacional - FRBA

Autor: Gustavo Nudelman

Clasificación

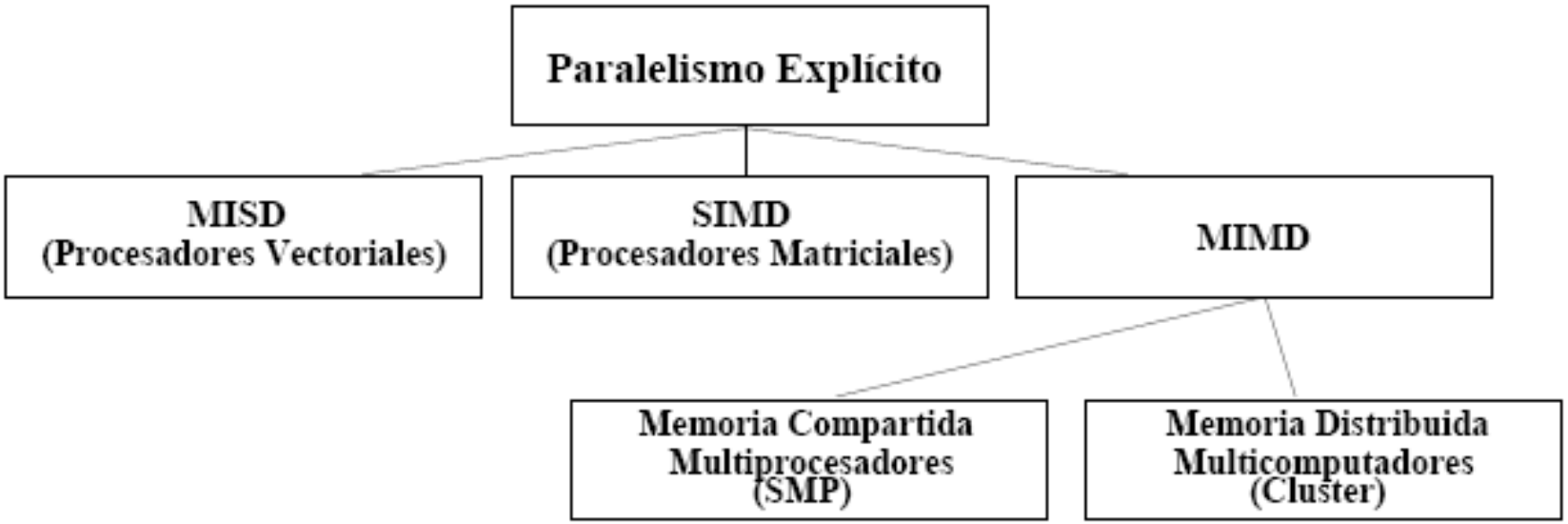
Definimos como procesamiento paralelo al concepto de agilizar la ejecución de un programa mediante su división en fragmentos que pueden ser ejecutados simultáneamente

- Paralelismo implícito – bajo nivel
  - Mejora de la concurrencia de la CPU
  - oculta a la arquitectura computacional
  - Aprovechamiento de los recursos de paralelismo de la CPU
  - Segmentación o *pipeline*
  - Ejecución fuera de orden
  - Procesadores auxiliares (Video – FPU)
  - SIMD

TD3

- Paralelismo explicito – Alto nivel
  - Varios procesadores con memoria dedicada
  - Varios procesadores con memoria compartida
  - ofrecen una infraestructura explícita para el desarrollo de software

Paralelismo explícito

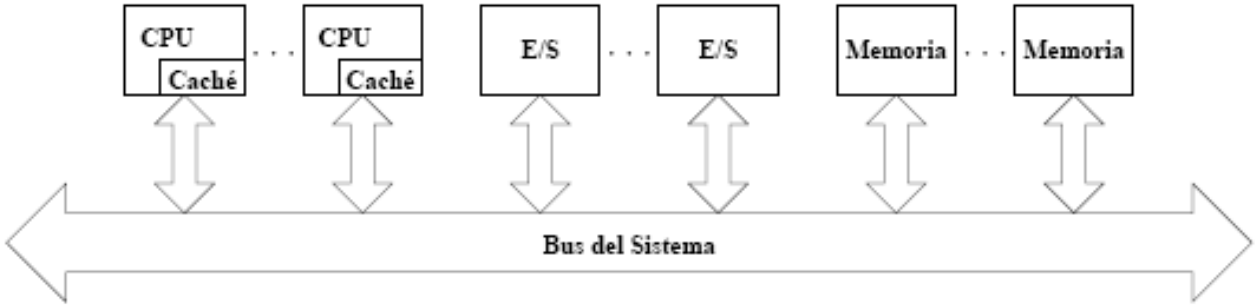


Memoria compartida - Multiprocesadores

La organización de un sistema multiprocesador puede clasificarse como sigue:

- Bus de Tiempo Compartido o Común
- Memoria multipuerto
- Unidad de control central

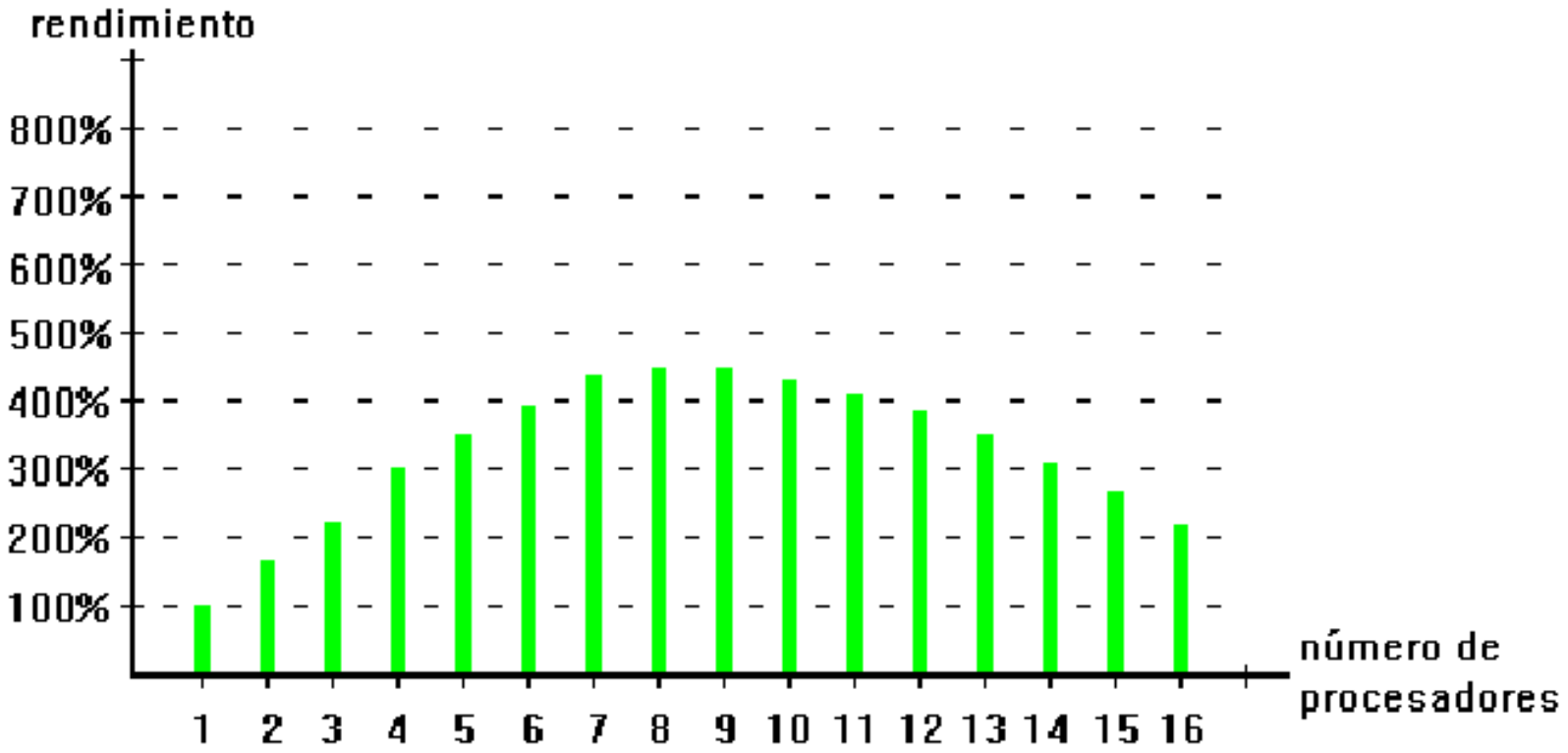
Bus de tiempo compartido



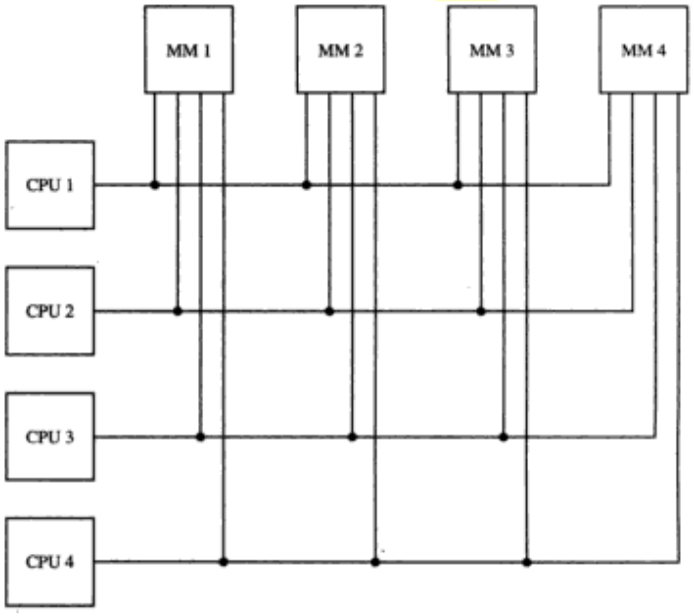
- Los procesadores compiten por el control del Bus
- Es el utilizado por los sistemas SMP de Intel. (Ver norma MPS de intel para sistemas operativos)
- Los sistemas SMP poseen zonas de memoria dedicadas a paso de mensajes entre procesadores.
- Es importante la cache interna de los procesadores para disminuir los accesos al bus. Sin embargo se necesita comunicar los cambios de cache interna a los otros CPU.
- Limitado a pocas decenas de procesadores
- Fácil de expandir
- Posee tolerancia a fallos (Un CPU Puede ser quitado sin interrupción del sistema)
- Linux SMP Puede operar sobre el concepto “processor Affinity” a partil de kernel 2.6.

Bus de tiempo compartido (2)

El Nº de procesadores no es infinitamente escalable



Memoria Multipuerto



- permite el acceso directo e independiente a los módulos de memoria desde cada una de las CPUs
- Cada puerto de la memoria posee una determinada prioridad
- Se puede configurar memorias como privadas para una determinada CPU
- Es mas eficiente que el sistema de Bus pero requiere una lógica mas compleja.

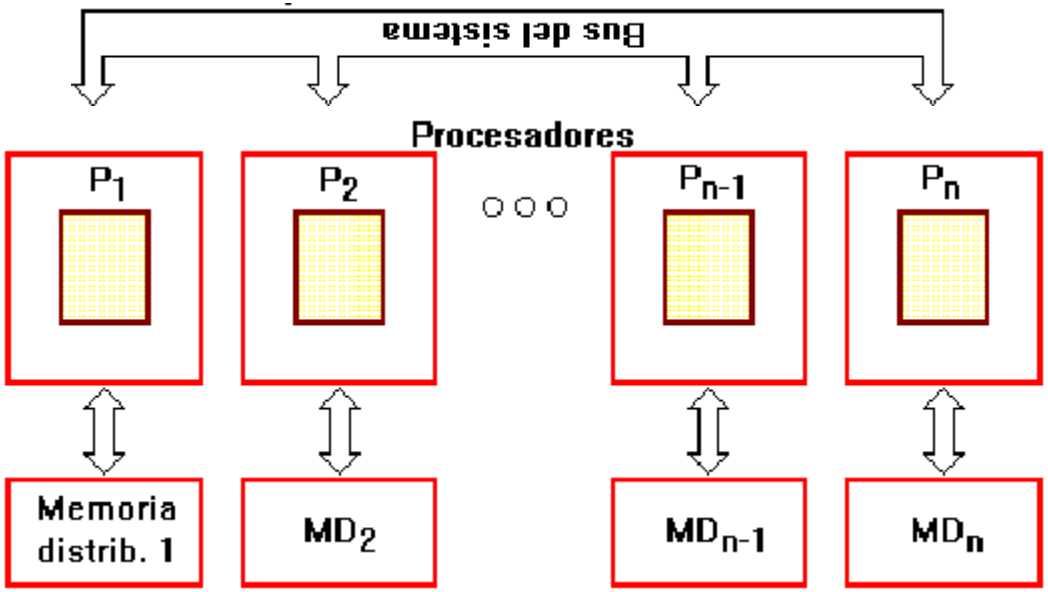
Unidad de control central

- Se trata de una unidad de control que permite la coordinación de las CPU’s
- Multiplexa las distintas secuencias de datos
- Lleva registros de estado de Cada CPU
- Lleva registros de cambios en las Cache´s

Memoria distribuida - multicomputadoras

- Sistema formado por unidades de procesamiento autónomas. (Cada CPU dispone de su propia memoria principal y sus canales de E/S)
- Cada procesador posee su espacio privado en memoria por lo que la coordinación debe funcionar por medio de una red dedicada.
- Mayor escalabilidad debido a que son sistemas autónomos
- Este tipo de sistemas se divide en 2 grandes grupos:
  - MPPs (“Massively Parallel Processors”, Procesadores Masivamente Paralelos). Estos sistemas pueden distinguirse de los anteriores por las propiedades que los hacen Masivamente Paralelos: ; Utilización de interconexiones de muy alto rendimiento ; Capacidad de E/S inmensa ; Tolerancia a fallos.
  - Clusters de procesamiento paralelo

MPP (Sistemas masivamente paralelos)



- Cada procesador posee su propia memoria no compartida – Es decir: Funciona como una unidad computacional
- Existe un bus dedicado para intercambio de mensajes entre procesadores
- El bus de paso de mensajes funciona por contención

Cluster de procesamiento paralelo

- Grupos de sistemas autónomos interconectados por interfaces de red formando diferentes topologías que tienen en común la pertenencia al modelo MIMD
- Se implementa con Hardware de fabricación a gran escala → Disminución de los costos.
- No posee las limitaciones de cantidad de unidades de procesamiento como SMP-
- Facilidad de mantenimiento.
- Velocidad de transferencia mucho menor que SMP (inclusive con FC)
- No todas las aplicaciones están preparadas para trabajar en cluster

Análisis y diseño de software paralelizable

Speedup

- Se define como Speedup a cuanto mas rápido es un programa al ejecutarse en un sistema de multiprocesamiento en relación a su tiempo secuencial

- Sea un proceso que tiene una demora de ts (secuencial) al ejecutarse en una única unidad computacional y sea p la cantidad de unidades computacionales

$$S(p) = \frac{Ts}{Tp}$$

Siempre S(p)<p y esto es debido a :

- **No toda la secuencia de un programa es paralelizable (ej. Cálculos extras para reunificar los resultados parciales)**
- **Tiempos ociosos entre procesadores cuando uno finaliza un proceso antes que otro y debe esperarlo**
- **Tiempos de comunicación en si misma**

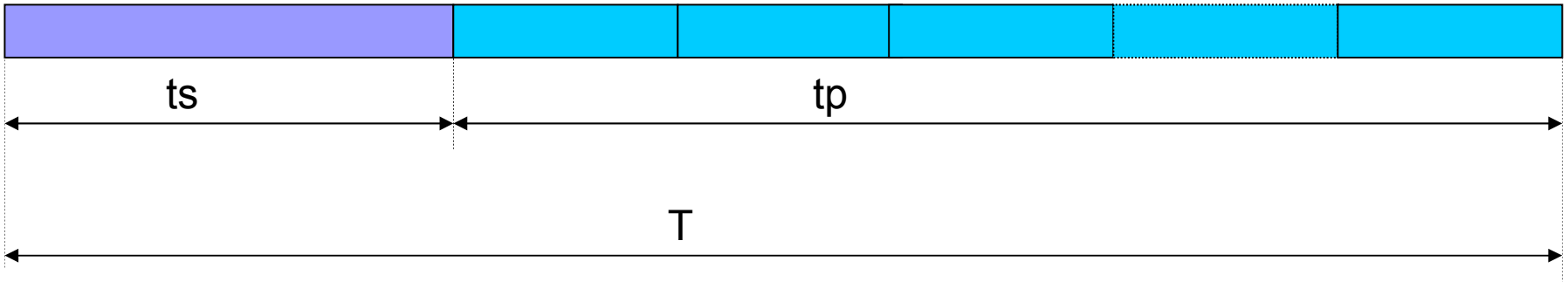
Ley de Amdahl

- No da la pauta de cuanto podemos mejorar nuestro sistema con el aumento de unidades de procesamiento hasta que dicho aumento deja de mejorar el rendimiento.

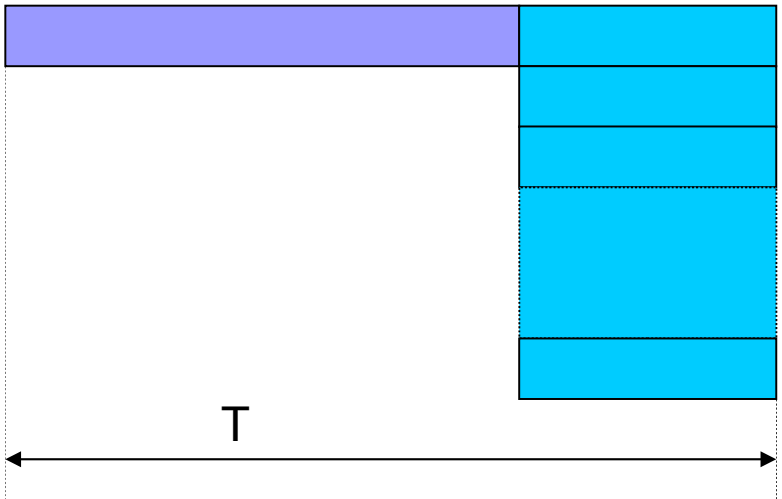
- Esta ley se basa en los siguientes principios:
  1. Existe un speedup teórico independiente de la tecnología de software y hardware utilizadas
  2. El speedup teórico se ve limitado por las secuencias de programa no paralelizables

Ley de Amdahl (2)

Con 1 sola unidad computacional



Con p unidades computacionales

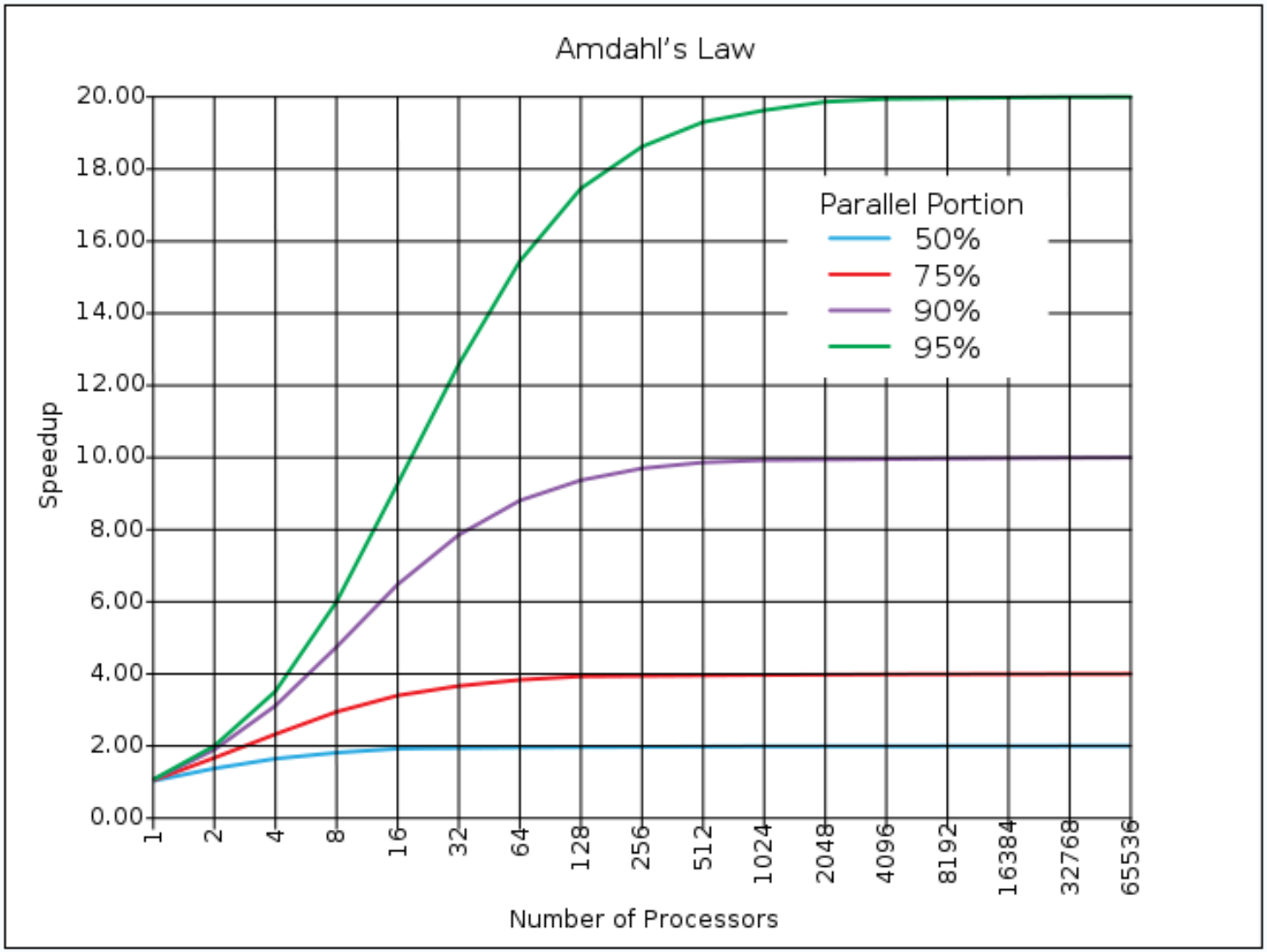


$$S(p,\theta) = \frac{1}{1 - \theta + \frac{\theta}{p}}$$

donde $\theta = \frac{tp}{T}$ Factor de paralelismo

eficiencia $S(p) = \frac{S(p)}{p}\%$

Ley de Amdahl (3) Valores tabulados



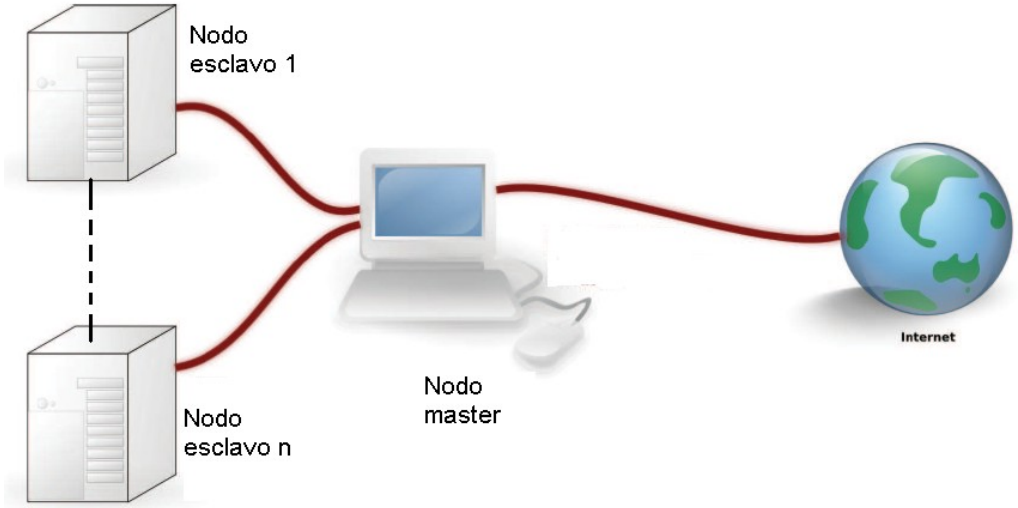
Calculo de tiempos de procesamiento

- Una vez calculado el speedup tenemos lo que se denomina “Tiempo Computacional” Tcomp
- A este tiempo debemos sumarle el tiempo de comunicaciones Tcomm

$$Tr = Tcomp + TComm$$

- El tiempo de comunicaciones debe ser reducido al mínimo
  - No debe existir mas de un salto entre entornos de colisión
  - El contenido del mensaje debe ser mínimo
  - Utilizar funciones no bloqueantes cuando sea necesario

Topología de cluster para procesamiento paralelo - BEOWULF



- Se comporta como un único sistema hacia el exterior
- Programación fuertemente dependiente de la arquitectura. Primero se diseña el modelo de paralelismo y luego se obtiene la estructura física
- No se utiliza memoria compartida entre nodos sino paso de mensajes (memoria distribuida)
- Desde el exterior solo es visible el nodo master
- Se utilizan conexiones de red directas y no por medio de switches
- Sobre esta topología se han desarrollado librerías y facilidades para procesamiento paralelo
  – **Nosotros utilizaremos las MPI**

MPI (Message Passing Interface)

- desarrollada en 1993-1994 por un grupo de investigadores al servicio de la industria, el gobierno y el sector académico.
- Ampliamente aceptado y difundido
- Evolución de las mejores prestaciones de los sistemas previamente existentes como PVM
- Portable – Inclusive puede ejecutar un programa paralelo entre arquitecturas diferentes.
- Comunicación eficiente entre nodos
- Free! http://www.open-mpi.org/

Alcance del standard MPI

- Comunicaciones Punto a Punto
- Operaciones Colectivas
- Agrupamientos de Procesos
- Topologías de Comunicación
- Soporte para Fortran y C

No soportado por el standard MPI

- Comunicaciones de Memoria Compartida
- Soporte de SO para recepciones por interrupción
- Ejecución remota de procesos
- Soporte para threads
- Soporte para control de tareas

Modelo de capas de un cluster MPI

- MPI funciona como un conjunto de módulos del kernel y utiliza los stacks standards de comunicaciones

Programación con MPI

```
#include <iostream.h>
#include <mpi.h>
int main(int argc, char **argv)
{
 MPI_Init(&arc,&argv);
 cout << “Hola Mundo” << endl;
 MPI_Finalize();
}
```

definiciones, macros y prototipos de función necesarios que utilizaremos con MPI

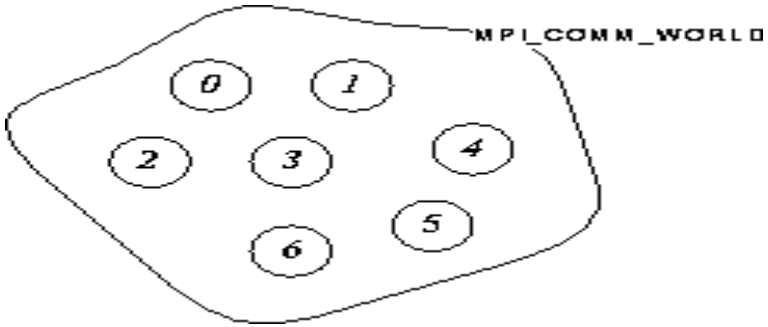
Inicializa las estructuras de comunicación necesarios y una instancia de la clase MPI para el proceso

Siempre debe ser llamada antes de comenzar a utilizar las funciones MPI!!

Complementaria a MPI Init. Debe ser invocada antes de finalizar el proceso

MPI - Comunicadores

- Un comunicador representa una colección de procesos que se pueden comunicar.
- Es un recurso que posee varias propiedades, atributos y funciones que se pueden aplicar al comunicador en si o a los procesos asociados a dicho comunicador
- Especifica un dominio de comunicación que puede ser usado en comunicaciones punto a punto y colectivas.
- Evitan que exista comunicación no deseada
- Contiene un grupo y un contexto
- Cada proceso dentro de un grupo esta identificado por un numero de rango (rank)



MPI (Identificacion de un proceso)

```
#include <iostream.h>
#include <math.h>
#include <mpi.h>
int main(int argc, char ** argv)
{
  Int mynode, totalnodes;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
  cout << "Hello world from processor " << mynode;
  cout << " of " << totalnodes << endl;
  MPI_Finalize();
}
```

Deja en la dirección de totalnodes un entero correspondiente a la cantidad instancias en ejecución

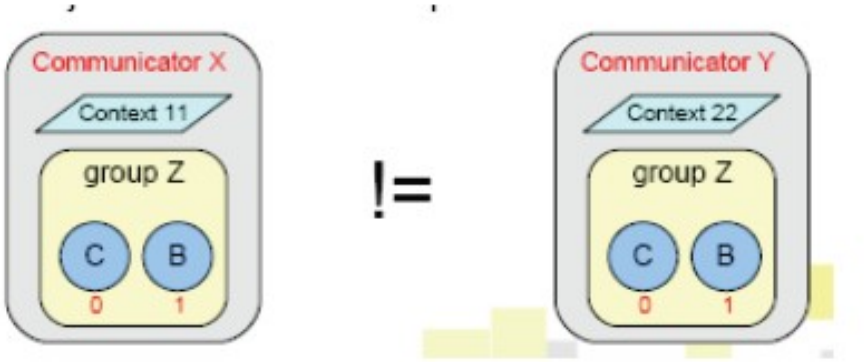
Deja en la dirección de mynode un entero correspondiente al ID MPI de la instancia. Es decir el **rank** en el grupo

Las funciones MPI_Comm_size y MPI_Comm_rank también devuelven un entero para poder determinar si fue exitosa o ocurrió un error

MPI_COMM_WORLD es una cte definida en MPI.h que significa “Todos los procesos que pueden comunicarse en esta colección”

Mas acerca de comunicadores

- **Contexto de comunicación**
  - Los contextos son únicos
  - Generan espacios disjuntos para las comunicaciones



Comunicadores predefinidos

- **MPI_COMM_WORLD** Envuelve a todos los procesos de la ejecución MPI
- **MPI_COMM_SELF** Envuelve solo a los procesos que están en un determinado contexto
- **MPI_COMM_NULL** Ningún proceso
- El usuario puede crear sus propios comunicadores de acuerdo a las necesidades del programa

Algunas funciones útiles

- int MPI_Get_processor_name(char* nombre, int* longnombre)

Devolución de hostname

(El buffer debe ser de un tamaño máximo MPI_MAX_PROCESSOR_NAME

Devolución de strlen(Hostname)

- Double MPI_Wtime()

Nos da el tiempo relativo en segundos desde MPI_Init (Cuando iniciaron todos los procesos)

- Double MPI_Wtick()

Nos da el valor en segundos que existe entre dos muestras del sistema de medición de tiempos

MPI – Comunicaciones entre procesos

- Comunicación punto – punto
  - Se intercambian datos entre 2 procesos solamente
  - El proceso fuente envía el mensaje
  - El proceso receptor tiene que recibirlo (activamente)
  - Se utilizan las funciones *MPI_Send y MPI_Recv*

Paso de mensajes MPI

- **int MPI_Send(void* mensaje, int cant, MPI_Datatype tipo_datos, int destino, int etiqueta, MPI_Comm comunicador)**
  - Mensaje: Puntero al buffer que contiene los datos a transmitir
  - Cant: Cantidad de elementos a enviar
  - Tipo_datos: Unidad a transmitir dentro del buffer Ver cuadro
  - N de nodo destino
  - Etiqueta: Entero para multiplexaxión dentro del nodo o MPI_ANY_TAG para todos
  - Comunicador: Generalmente MPI_COMM_WORLD

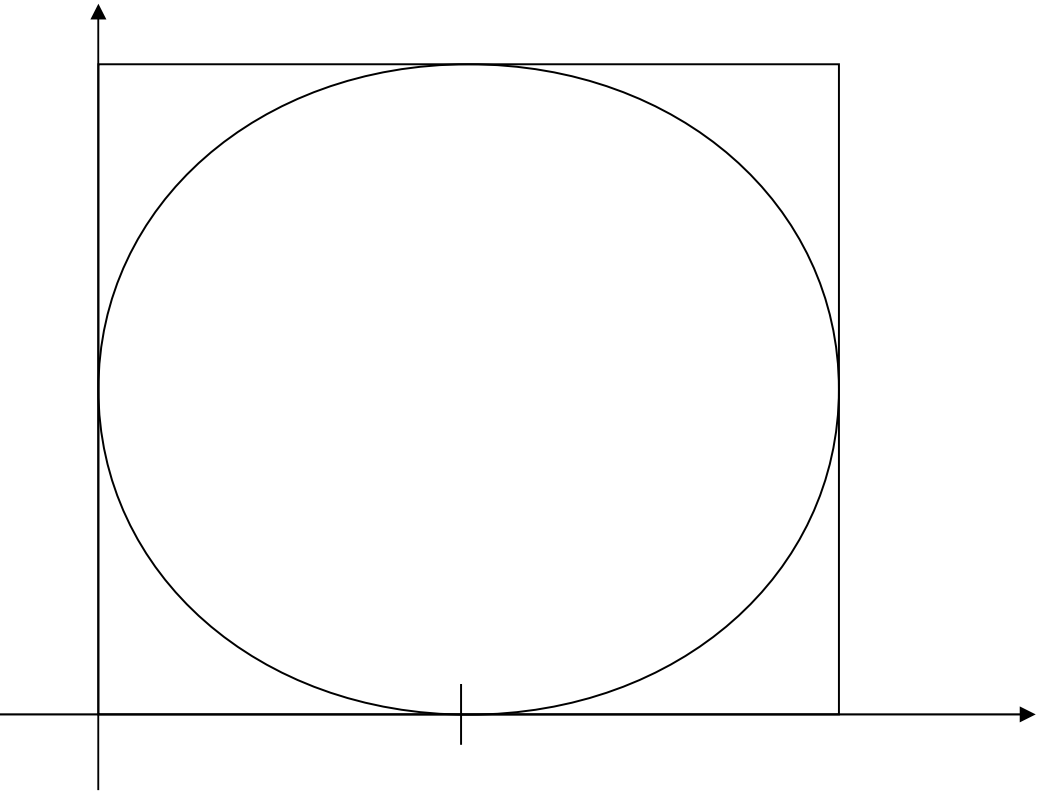
Tipo de Datos MPI	Tipo de Datos C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

•int MPI_Recv(void* mensaje, int contador, MPI_Datatype tipo_datos, int origen, int etiqueta, MPI_Comm comunicador, MPI_Status* status)

•Se puede especificar el nodo origen del cual se quiere recibir el mensaje o todos mediante la cte

•Satus nos da info del mensaje .(Es una estructura MPI_Status)

Un ejemplo de aplicación – Generar ΠΤ con la aprox deseada



1. Sea N una cantidad de puntos aleatorios generados dentro de los limites del cuadrado.
2. Sea n los puntos aleatorios generados que pertenecen a la circunferencia
3. Si N es suficientemente grande podemos escribir la sig aproximación

$$\Pi r^2 = \frac{n}{N}(2r)^2$$

Despejando y ordenando

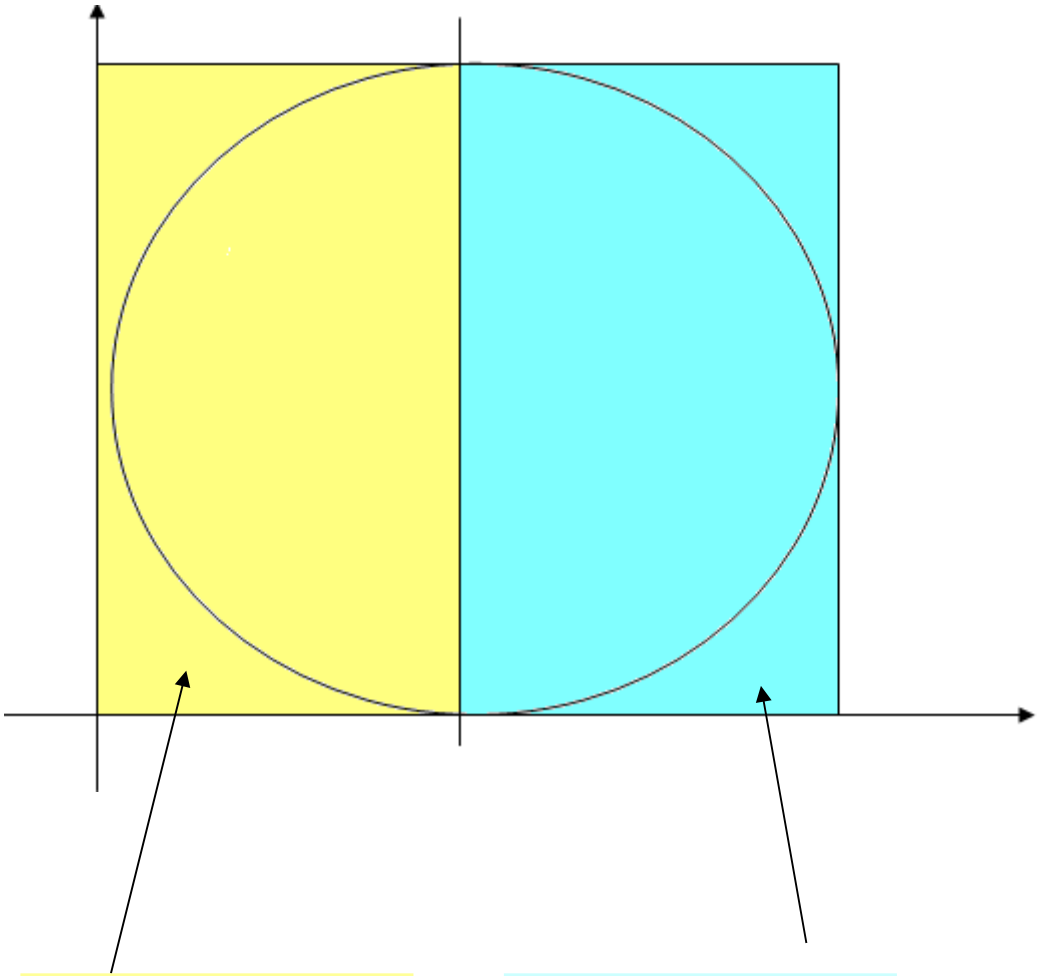
$$\Pi = 4 * \frac{n}{N}$$

Cuanto mayor sea N obtendremos un valor mas exacto de π

Generar π – Algoritmo No distribuido

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

int main(int argc, char* argv[])
{
    int rank, size;
    long long cantpuntos,Nin,i ;
    double starttime,endtime,lado,aproxpi,rndx,rndy;

    lado=atof(argv[1]);
    cantpuntos=atoll(argv[2]);

    MPI_Init(&argc, &argv);
    starttime=MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    Nin=0;
    printf("\nNodo %d Iniciado",rank);

if (rank==0)
      {
       printf("\nProceso %d calculando\n",rank);
       for (i=1;i<cantpuntos;i++)
        {
         rndx=lado*drand48();
         rndy=lado*drand48();
         if(pow((rndx-(lado/2)),2)+pow((rndy-
(lado/2)),2)<=pow((lado/2),2))
            Nin++;

        }
        printf("\nIn: %ld\n",Nin);
        aproxpi =(float)  4*Nin/cantpuntos;
        printf("\nPi: %1f \n",aproxpi);
        endtime=MPI_Wtime();
        printf("\nTiempo de calculo %f
segundos\n",endtime-starttime);

    }
```

Generar π – Algoritmo distribuido (Dividamos el workload)



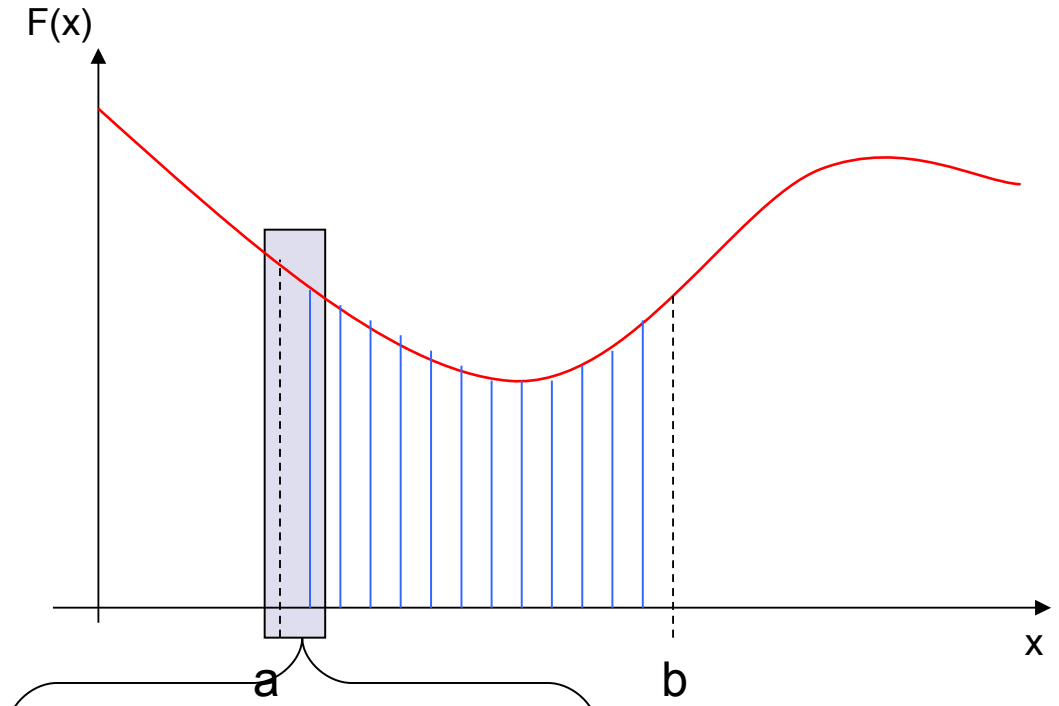
- Necesitamos la mitad de muestras por cada CPU ya que cada una trabaja con la mitad de superficie. De esta manera mantenemos la resolución de nuestro análisis
- Se obtienen n0 aciertos de N/2 intentos en la zona amarilla
- Se Obtienen n1 aciertos de N/2 intentos en la zona Celeste
- El nodo master recolecta los resultados y realiza la adición final para presentar

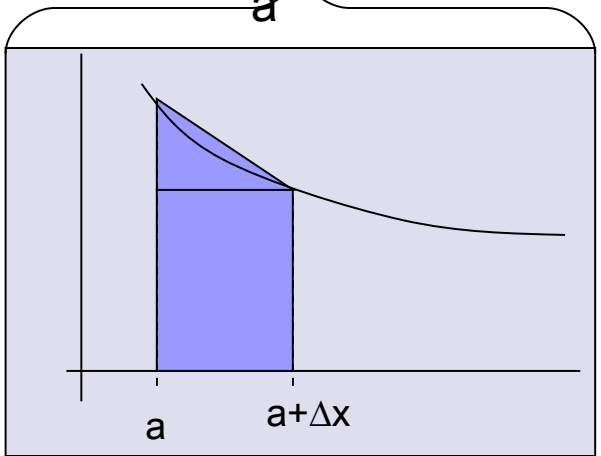
Generar π – Algoritmo distribuido

```
if (rank==0)
    {
     printf("\nProceso %d calculando\n");
     for (i=1;i<cantpuntos/2;i++)
      {
       rndx=lado/2*drand48();
       rndy=lado*drand48();
       if(pow((rndx-(lado/2)),2)+pow((rndy-
    (lado/2)),2)<=pow((lado/2),2))
           Nin++;
      }
     printf("\nEsperando Resultado de nodo paralelo\n");

    MPI_Recv(&NinRemote,1,MPI_UNSIGNED_LONG,M
    PI_ANY_SOURCE,NULL,MPI_COMM_WORLD,NULL
    );
     printf("\nRecibido Valor %ld\n",NinRemote);
     printf("\nIn: %ld\n",Nin);
     printf("\n InTotal: %ld ",Nin+NinRemote);
     resul = 4 * (Nin+NinRemote)/cantpuntos);
     printf("\nPi: %f\n",resul);
     endtime=MPI_Wtime();
     printf("\nTiempo de calculo %f segundos\n",endtime-
    starttime);

    }
```

```
if (rank=1)
    {
      for (i=1;i<cantpuntos/2;i++)
      {
       rndx=(lado/2)+lado/2*drand48();
       rndy=lado*drand48();
       if(pow((rndx-(lado/2)),2)+pow((rndy-
(lado/2)),2)<=pow((lado/2),2))
          Nin++;
      }

MPI_Send(&Nin,1,MPI_UNSIGNED_LON
G,0,NULL,MPI_COMM_WORLD);

    }
```

Integración Numérica – Aprox Trapezoidal





Podemos ver claramente que la superficie del trapecio formado será: Δx [F(a) + F(A+ Δx)] / 2

•Dividimos el área entre a y b en n trapecios de base Δx donde Δx=(b-a)/n

•Podemos aproximar el calculo de la integral mediante la suma de los trapecios

$$A = \sum_{i=a}^{b} \frac{\left(F(Xi) + F(Xi + \Delta x\right)}{2} \cdot \Delta x$$

Comunicaciones Colectivas

- Permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico.
- No se usan etiquetas para los mensajes, estas se sustituyen por identificadores de los grupos (comunicadores).
- Comprenderán a todos los procesos en el alcance del comunicador.
- Por defecto, todos los procesos se incluyen en el comunicador genérico MPI_COMM_WORLD.

Comunicaciones Colectivas MPI

- *Las comunicaciones colectivas involucran a 2 o mas procesos a fin de ejecutar acciones colectivas para:*
  - Sincronizar
  - Mover datos
    - Difundir
    - Recolectar
    - Esparcir
  - Calcular colectivamente

Comunicaciones Colectivas en MPI (2)

- **Sincronizar**
  - Todos los procesos llaman a la rutina.

ierr = MPI_Barrier(communicator);

  - La ejecución es bloqueada hasta que todos los procesos ejecuten la rutina.

Comunicaciones Colectivas en MPI (3)

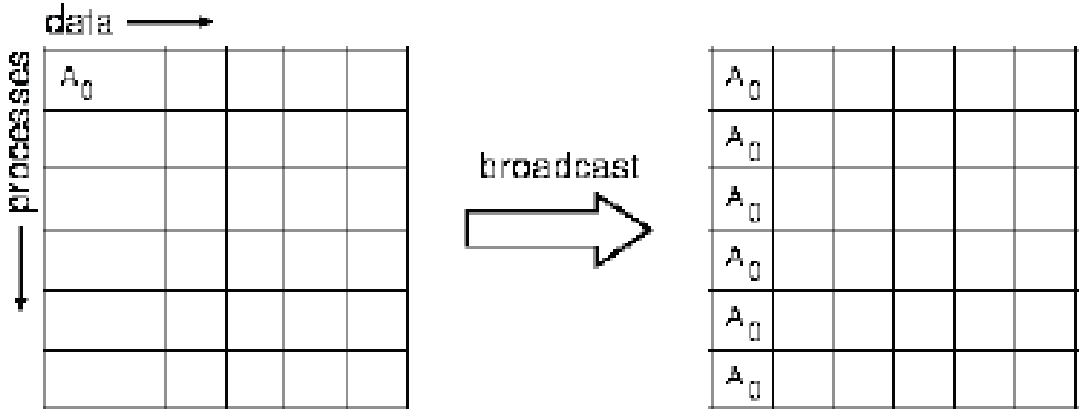
Difundir

Un proceso envía un mensaje (***root***).

Todos los otros reciben el mensaje.

La ejecución se bloquea hasta que todos los procesos ejecuten la rutina. Automáticamente, es decir: **actúa como punto de sincronización**.

ierr = MPI_Bcast(&buffer,count,datatype,
root,communicator);

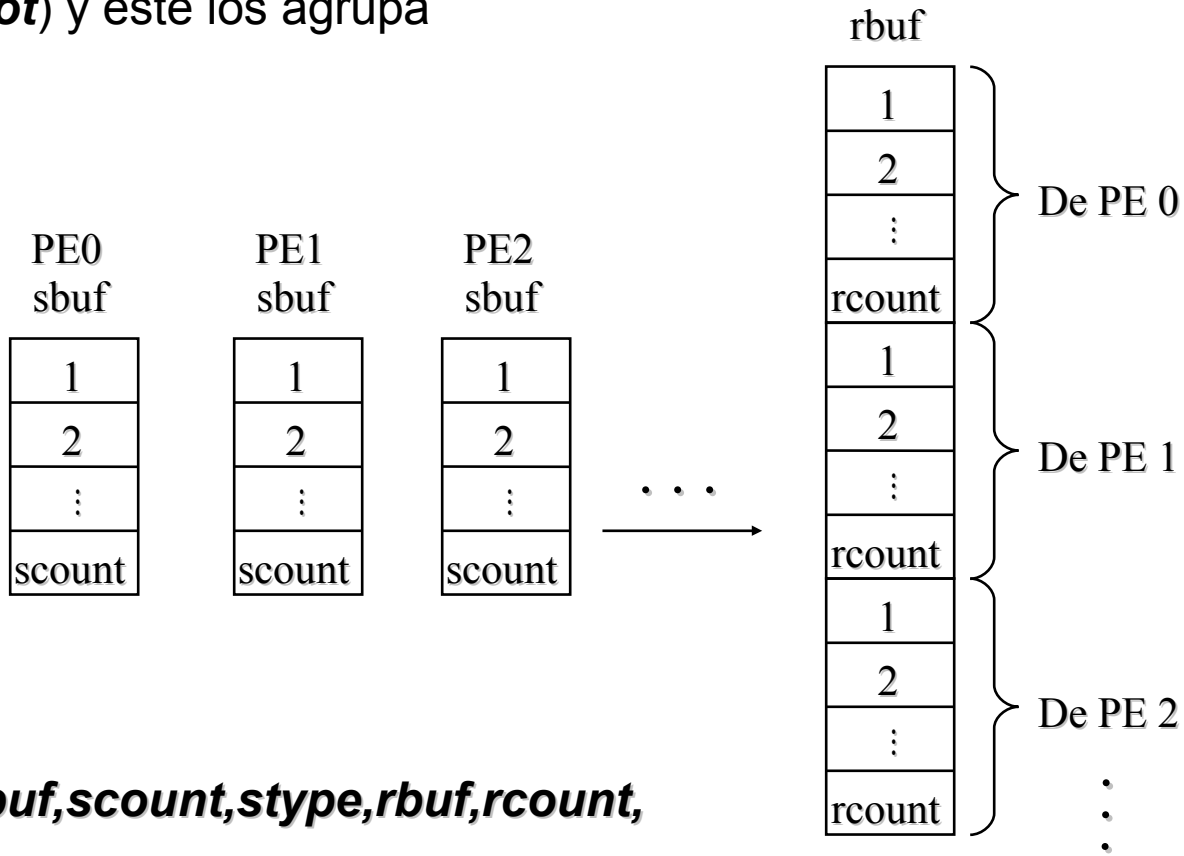


Ejemplo MPI_Bcast

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

int main(int argc, char* argv[])
{
 char msg[100];
 int rank, size;
 FILE *fp;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
     &rank);
if(rank==0)
 {
  printf("\nIntroducir mensaje:\n");
  gets(msg);
MPI_Bcast(msg,100,MPI_CHAR,0,MP
I_COMM_WORLD);
  printf("\n Mensaje enviado a todos
desde %d\n",rank);
 }
 else
 {
MPI_Bcast(msg,100,MPI_CHAR,0,MP
I_COMM_WORLD);
 fp=fopen("/tmp/bcast.log","w+");
 fprintf(fp,"\n Mensaje recibido en %d
desde master\n\n",rank);
 fprintf(fp,"\n%s\n",msg);
 fclose(fp);
 }
}
```

Comunicaciones Colectivas en MPI (4)

- **Recolectar**
  - Cada proceso envía diferentes datos al proceso principal (***root***) y este los agrupa en un arreglo.

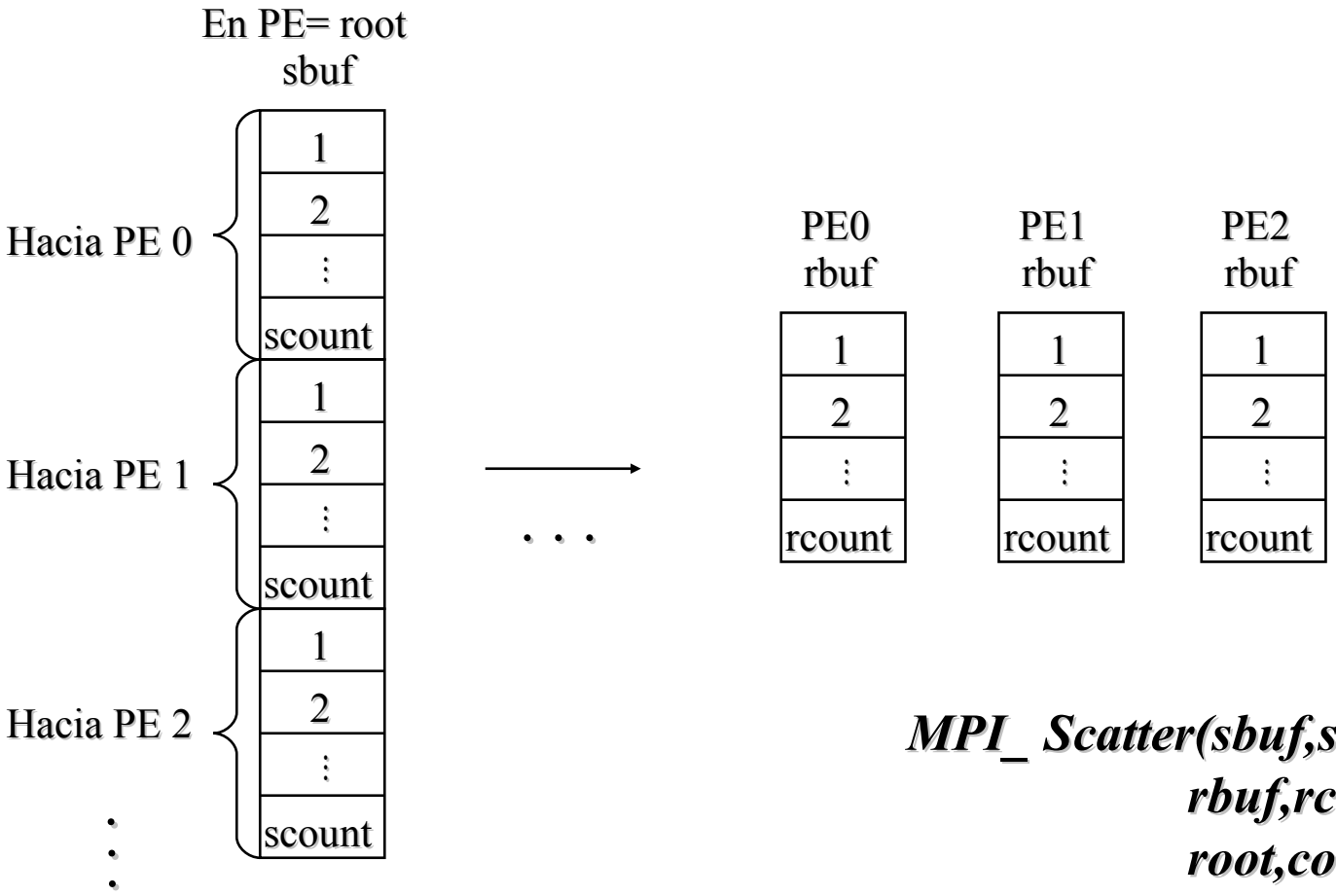


ierr = MPI_ Gather(sbuf,scount,stype,rbuf,rcount,

rtype,root,communicator);

Comunicaciones Colectivas en MPI (5)

- **Esparcir**
  - El proceso principal (***root***) rompe un arreglo de datos y envíe las partes a cada procesador



MPI_ Scatter(sbuf,scount,stype, rbuf,rcount, rtype, root,communicator);

Ejemplo MPI_Gather y MPI_Scatter

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

#define SIZE 100

int Obtener_Mayor(int* Buff,int cantidad);

int main(int argc, char* argv[])
{
    int rank,nproc,a[SIZE],b[SIZE],chunksize;
    unsigned long i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Ejemplo MPI_Gather y MPI_Scatter

```
chunksize= (int) SIZE/nproc;
    if (rank==0)
      {
       for (i=0;i<SIZE;i++)
         a[i]=i+1;
      }
    MPI_Scatter(a,chunksize,MPI_INT,b,chunksize,MPI_INT,0,MPI_COMM_WORLD);
    b[0]=Obtener_Mayor(b,chunksize);
    MPI_Gather(b,1,MPI_INT,a,1,MPI_INT,0,MPI_COMM_WORLD);
    if (rank==0)
    {
     printf("\n");
     for (i=0;i<nproc;i++)
       printf("%d, ",a[i]);
     printf("\n");
    }

}
```

Ejemplo MPI_Gather y MPI_Scatter

```
int Obtener_Mayor(int* Buff,int cantidad)
{
 int i,temp;

 temp=0;

 for (i=0;i<cantidad;i++)
   {
    if (Buff[i]>temp)
     temp=Buff[i];
   }
 return(temp);
}
```

Cálculos colectivos

- Combinación de resultados parciales
  - El proceso principal (***root***) recibe los cálculos parciales y los combina usando la operación indicada.

ierr = MPI_ Reduce(sbuf,rbuf,count,datatype,
operation, root, communicator);

Tipos de operaciones

MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_PROD	Producto
MPI_SUM	Suma
MPI_LAND	Y Lógico
MPI_LOR	O Lógico
MPI_LXOR	O Exclusivo lógico
MPI_BAND	Y bit a bit
MPI_BOR	O bit a bit
MPI_BXOR	O Exclusivo bit a bit
MPI_MAXLOC	Máximo y localización
MPI_MINLOC	Mínimo y localización

Ejemplo MPI_Reduce

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

#define SIZE 100

int Obtener_Mayor(int* Buff,int cantidad);

int main(int argc, char* argv[])
{
    int rank,nproc,a[SIZE],b[SIZE],chunksize;
    unsigned long i,sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    chunksize= (int) SIZE/nproc;
    if (rank==0)
      {
       for (i=0;i<SIZE;i++)
         a[i]=i+1;
      }
```

Ejemplo MPI_Reduce

```
 MPI_Scatter(a,chunksize,MPI_INT,b,chunksize,MPI_INT,0,MPI_COMM_W
 ORLD);
b[0]=Obtener_Mayor(b,chunksize);
MPI_Gather(b,1,MPI_INT,a,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Reduce(b,&sum,1,MPI_LONG,MPI_SUM,0,MPI_COMM_WORLD);
if (rank==0)
{
 printf("\n");
 for (i=0;i<nproc;i++)
   printf("%d, ",a[i]);
printf("\nSumatoria: %d\n",sum);
}
```

Definición de datos en MPI

- Se pueden construir tipos de datos derivados de tipos estándares
- Se permite la comunicación de estructuras de datos mas complejas
- Los mismos pueden ser:
  - Homogeneos → Compuestos de datos del mismo tipo
    - Tipos contiguos
    - Tipos vectoriales
    - Tipos Indexados
  - Heterogéneos → Compuestos de datos de tipo diferente.
    - Tipo Struct

Construcción de un nuevo tipo de datos

- Se declara un objeto para definir el nuevo tipo
  - MPI_Datatype *NewTipe*
- Se construye el tipo con el nuevo objeto mediante la función correspondiente según la clase de construcción deseada seguida de la función:

```
int MPI_Type_commit(MPI_Datatype *datatype);
```

- Se utiliza el nuevo tipo con las funciones de intercambio de mensajes
- Se libera el Objeto

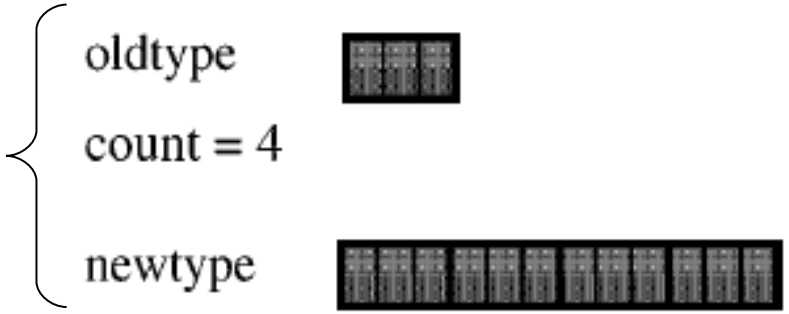
```
int MPI_Type_free(MPI_Datatype *datatype);
```

Tipos Contiguos

•Se trata de un tipo formado por una colección de elementos de un tipo básico.

•Todos ellos del mismo tamaño y almacenados consecutivamente en memoria.

Ejemplo




```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype,
                         MPI_Datatype *newtype);
```

Tipos Vectoriales

- Define un tipo de dato derivado que consiste en elementos que no están alineados consecutivamente sino separados por intervalos reglares

- Esto es posible gracias a que C mantiene los elementos de vectores en forma contigua en la memoria (sea cual sea la dimensión)

```
int MPI_Type_vector (int count, int blocklength, int
stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Parámetros de construcción

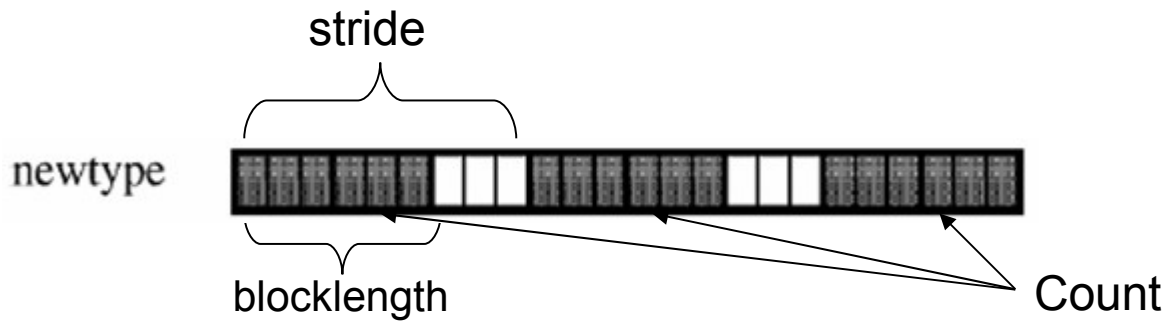
Count=3,

blocklength=2

stride=3

Ejemplo

Sea oldtype  (Tipo Básico)



Aplicación: Manipulación de matrices

- **Tipo: "Columna" (Matriz 4x4)**

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

Count=4

Blocklength=1

Stride=4

- **Tipo: "Diagonal" (Matriz 8x8)**

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Count=8

Blocklength=1

Stride=9

Ejemplo de envió y recepción del tipo “diagonal”

```
#include <stdio.h>
#include <mpi.h>
#define size 10

int my_id, nproc, sizeofint, a[size][size], b[size], i, j;
MPI_Datatype diag;
MPI_Status status;

int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

```
  /* Crea tipo para la diagonal */
  MPI_Type_vector(size, 1, size+1, MPI_INT, &diag);
  MPI_Type_commit(&diag);
  if (my_id == 0) {
    for (i=0; i<size; i++)
      for (j=0; j<size; j++) a[i][j] = i*10+j;
    for (i=1; i<nproc; i++)
      MPI_Send(a,1,diag,i,99,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(b,size,MPI_INT,0,99,MPI_COMM_WORLD,&status);
    printf("Yo soy: %d\n",my_id);
    for (i=0; i<size; i++) printf("%3d ",b[i]);
    printf("\n");
  }
  MPI_Finalize();
}
```

Tipos Indexados

- Elementos no equiespaciados, de modo que las distancias entre ellos son ahora un vector de distancias.

- Cada bloque tiene un numero diferente de elementos por lo que ahora necesitaremos un vector de tamaños de bloque

int MPI_Type_indexed(int count, int *lengths, int *disps, MPI_Datatype oldtype, MPI_Datatype *newtype)

count: cantidade de elementos
length: número de entradas en cada elemento (vector de enteros)
disps: distancias relativas de los elementos desde el inicio (vector de enteros)

Ejemplo

a00 a01 a02 a03 a04
a10 a11 a12 a13 a14
a20 a21 a22 a23 a24
a30 a31 a32 a33 a34
a40 a41 a42 a43 a44

Elementos a enviar/recibir

longs[i] = (N-i); // entradas = {5,4,3,2,1}
desp[i] = (N+1)*i; // distancias = {0,6,12,18,24}

MPI_Type_indexed(5, longs, desp, MPI_INT,&tipo);

Estructuras no homogéneas

intMPI_Type_struct(count, array_bl, array_d, array_t, new)

Count: Número de elementos (*bloques*) del nuevo tipo.

Array_bl: Vector de enteros con las longitudes de cada tipo de la estructura

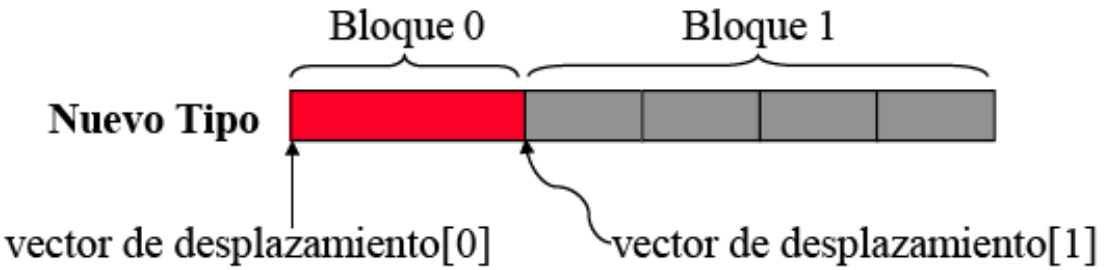
Array_d: Vector con los strides para cada elemento de la estructura

Array_t: Vector con los tipos de cada elemento de la estructura

New: Tipo que se construye

Ejemplo de una estructura no homogénea

```
struct {
  double x;
  int x[4];
} my_struct;
```




```
Int bloques = 2;
int blocklen[] = {1, 4};
int desp[] = {0, sizeof(double)};

MPI_Datatype tipos[]={MPI_DOUBLE,MPI_INT};
MPI_Datatype my_stru;
MPI_Type_struct(bloques,blocklen,desp,tipos,&my_stru);
```

Definición de operaciones

De la misma manera que se pueden definir tipos, podemos definir operaciones para ser utilizadas en funciones como Reduce

1. ***Debemos declarar un objeto "Operación"***
   MPI_Op myOp

1. ***Definimos la operación mediante la asociación del objeto "Operación" a una funcion C***
   MPI_Op_create (Void *CFunc, int Conmutativa, MPI_Op)

1. ***Liberamos el objeto***
   MPI_Op_free (MPI_Op)

Ejemplo para números complejos

```
#include <stdio.h>
#include <mpi.h>
#define size 1000

typedef struct {
  double real, imag;
} complex;

void my_Prod();
int my_id, i;
complex a[size], sol[size];
MPI_Op my_op;
MPI_Datatype ctype;

int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

  MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
  MPI_Type_commit(&ctype);
```

```
  for (i = 0; i < size; i++) {
    a[i].real = (my_id + 1)*i;
    a[i].imag = (my_id + 1)*(size - i);
  }
   MPI_Op_create(my_Prod, 1, &my_op);
   MPI_Reduce(a, sol, size, ctype, my_op, 0, MPI_COMM_WORLD);

   /* el proceso root tiene la soluci□n */
   if (my_id == 0)
     for (i = 0; i < size; i++)
       printf("(%10.0f,%10.0f)\n",sol[i].real,sol[i].imag);

   MPI_Op_free(&my_op);
   MPI_Finalize();
}

void my_Prod(complex *in, complex *inout, int *len, MPI_Datatype *dptr)
{
   int i;
   complex c;
   for (i = 0; i < *len; ++i) {
     c.real = inout->real*in->real - inout->imag*in->imag;
     c.imag = inout->real*in->real + inout->imag*in->imag;
     *inout = c;
     in++; inout++;
   }
```